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

Revised Specification of Case Study Systems

CEC Deliverable No:	**WP4-FHG/FOKUS-040-D5-Int**
Deliverable Type:	**PU**
Dissemination:	**PU**
Deliverable Nature:	**R**
Contractual date:	**April 2002 (as updated in Project Quarterly Reports)**
Actual date:	**Date of submission to European Commission**

Editor:	**Célestin Brou**
File Name	**WP4-FHG/FOKUS-040-D5-Ext.doc**
Contributors:	**WP4**
Version:	**4.0**
Version Date:	**Wednesday, 15 May 2002**
Internal Distribution:	**WP1, WP2, WP3, WP4, WP5 (add/delete as appropriate)**
Deliverable Status:	**Approved**

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
HEL	Partner	United Kingdom
HIT	Partner	Japan
SAG	Partner	Germany
ETH	Partner	Switzerland
FHG/FOKUS	Partner	Germany
IKV	Associate Partner to FHG/FOKUS	Germany
INT	Associate Partner to FHG/FOKUS	Spain
UPEN	Partner	USA

The FAIN Consortium

University College London	(UCL)
Josef Stefan Institute	(JSIS)
National Technical University of Athens	(NTUA)
Universitat Politecnica De Catalunya	(UPC)
T-Nova Deutsche Telekom Innovationsgesellschaft mbH	(DT)
France Télécom / R&D	(FT)
Hitachi Europe Ltd.	(HEL)
Hitachi Ltd.	(HIT)
Siemens AG	(SAG)
Eidgenössische Technische Hochschule Zürich	(ETH)
Fraunhofer-Gesellschaft zur Förderung der angewandten Forschung e.V.	(FHG/FOKUS)
IKV++ Technologies AG	(IKV)
Integracion Y Sistemas De Medida, SA	(INT)
University of Pennsylvania	(UPEN)

Project Management

Alex Galis
University College London
Department of Electronic and Electrical Engineering,
Torrington Place
London WC1E 7JE
United Kingdom
Tel ++44-(0) 207- 679 5738
Fax +44 (0) 207 388 9325
E-mail: a.galis@ee.ucl.ac.uk

Authors

Thomas Becker (FHG) Elisa Boschi (FHG) Matthias Bossardt (ETH) Yiorgos Bouloudis (NTUA) Celestin Brou (FHG/FOKUS) – WP4 Leader (Editor) Yannick Carlinet (FT) Lawrence Cheng (UCL) Jürgen Dittrich (IKV++) Spyros Denazis (HEL)- WP3 leader Ducan Gabrijelcic (JSIS) Christian Garbrecht (DT) Alex Galis (UCL)- WP1 leader Juan Luis Mañas González (INT) Peter Graubmann (SAG) Drissa Houatra (FT) Cornel Klein (SAG)- WP2 leader George Karetsos (NTUA) Chiho Kitahara (HIT) Stamatis Karnouskos (FHG/FOKUS) Antonis Lazanakis (NTUA)	Bertrand Mathieu (FT) Jens Meinköhn (DT) Franci Mocilar (JSIS) Eckhard Moeller (FHG) Tuan-Quoc Nguyen (DT) Yiannis Nikolakis (NTUA) Evelyn Pfeuffer (SAG) Bernhard Plattner (ETH) - WP5 Leader Mehran Roshandel (DT) Lukas Ruf (ETH) Epi Salamanca (UPC) Arso Savanovic (JSIS) Joan Serrat (UPC) Jonathan M. Smith (UPEN) Kiminori Sugauchi (HIT) Toshiaki Suzuki (HEL) Alvin Than (UCL) Christos Tsarouchis (HEL) Mercedes Urios (INT) Julio Vivero (UPC) Christoph Weckerle (IKV++) Ermolaos Zimboulakis (NTUA)

Executive Summary

This document (D5) is the second of a set of deliverables (D3, D5, D8) that reports periodical advancement of the work achieves in WP4. An early version of the deliverable D4 was issued as R14/R15/R16 Internal Reports in early March 2002. This Work Package is responsible for implementing an operator solution and case studies. A Policy-Based Active Network management (PBANM) and Active Service Provisioning (ASP) are being developed the entire project long with continuous refinement of the existing. A 2-tiers based architecture has been identified earlier when designing the two systems as well synergy to realise an integrated active network architecture together with the active node being developed by WP3 team.

The two 2-tiers architecture's assumption adopted in WP4 identified two levels of concern regarding the PBANM and the ASP: elements level functionality and network wide one, which have the same components but different semantics. The element level functionality managed a single active node while the second one deal with FAIN architecture network wide. That has been fully documented in D3 that report initial design of the Management system Architecture. However the architecture has been review as the design evolved. One the major changed is the generalisation of the management delegation from the NIP to the SP and integration of open issues' solutions into the FAIN architecture.

As earlier prototypes focus on an active node management on which it was somehow more natural to resolve efficiently problems raise up by policy based active management and service provisioning, we prioritise to concentrate on the element level management and project solutions network wide. Thus in this report element level components are well designed as well as most of their network peers. However the network level resource manager, the inter-domain management components have been postponed for Y3 for time constraint and high priority of code production for planed milestone. Indeed a lot has been achieved at both levels and will be presented in report R21 that complements D5. A practical organisation of both documents allows avoiding duplication as much as possible of their contents.

Description of Deliverable

D5 contains the revised specification of Case Study Systems achieved since Y1 in FAIN. This deliverable is an extension and consolidation of the D3 – "Initial Specifications of the FAIN Case Studies". It will be updated and consolidated in the year 3 as D8.

It is derived from R14, R15 and R16 describing respectively at the FAIN Element, Network level Management and the Active Service Provisioning Systems for Active Networks. The document focuses on design of the Policy Based Management system initially described in D3.

Following this introduction, a brief overview of the FAIN project is given to review the main contents in D3. Chapter 2 presents the main architectural refinements as well as a mapping of the actual role in the architecture to the FAIN Enterprise Model. Chapter 3 presents a global overview of the management system together with the description of generic components within the 2-tiered architecture. Chapters 4, 5 and 6 constitute the main design of the element-level management, network-level management and the Active Service Provisioning (ASP) respectively.

Keywords

Active Network, Active AN Management, Active Service Provisioning, XML, Policy Based Management, Agent Technology, DPE, QoS Delegation.

Change History

1. Initial Draft Jan 4th, 2002, Version 1.0
2. 2nd draft February 28th 2002
3. 3rd draft B March 11, 2002
4. 4th Draft C April 30, 2002
5. 5th for internal review 14,2002

Table of Contents

Table of Figures

Table of Tables

1 INTRODUCTION

D5 is a consolidation of three other internal reports, namely R14, R15, and R16, which describe the element-level management, network-level management and the Active Service Provisioning (ASP) respectively. This deliverable is an extension and consolidation of the D3 – "Initial Specifications of the FAIN Case Studies"; and as such focuses on the refinement of the previous architecture and design.

Implementation detailed design of the FAIN management architecture are elaborated in the internal reports, R21. In this document we use, Use case diagrams using the Unified Modelling Language (UML) to describe the functionality and motivation for each component.

Our network management approach allows recursive delegation of the management system from the Network Infrastructure Provider to the Customer (*cf*. FAIN Enterprise Model in D1) by offering a restricted instance of the management system. Following this introduction, a brief overview of the FAIN project is given to review the main contents in D3. Chapter 2 will present the main architectural refinements as well as a mapping of the actual role in the architecture to the FAIN Enterprise Model. Chapter 3 furnishes a global overview of the management system together with the description of generic components within the 2-tiered architecture. Chapter 4, 5 and 6 constitute the main design of the element-level management, network-level management and the Active Service Provisioning (ASP) respectively.

1.1 FAIN Overview

1.1.1 Active Networking Issues in FAIN

The FAIN active network architecture defines active nodes, which provide full flexibility to the user for network management and service provisioning. The defining characteristic of an active node is the ability for users to load and manage software components dynamically and efficiently. This can be achieved safely since customers who are sharing the same active node would be provided with a VPN-like resource partitioning.

Packets requiring active processing are marked to allow correct handling by active routers. This allows the discrimination of active and conventional packets and the selection of an active node. Routing and node resources configuration in the active nodes could be achieved by setting policies at the network management level (element and network management nodes). Access to this functionality will be controlled and only possible via a well-defined API.

Figure 1 exemplifies a configuration of an active network and its management nodes.

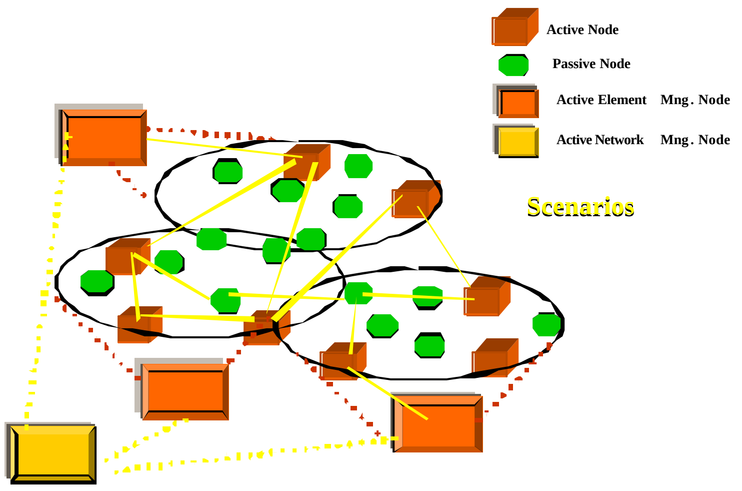


Figure 1: Management of Active Networks in FAIN

1.1.2 Components in the FAIN Active Node

In relation to D4, we provide a summary of issues that are pertinent to WP3. The FAIN Reference Architecture consists mainly of the following entities*:* AA, EE and Node OS.

- *Active Applications/Services* (AA) are applications executed in active nodes.
- *Execution Environments* (EE) are environments where application code is executed. A privileged EE manages and controls the active node and it provides the environment where network policies are executed. Multiple and different types of EE are envisaged in FAIN. A VE is a collection of resources (including one or more Execution Environments) owned by a particular SP. Thus EEs are used through virtual environments (VEs), where services can be found and interact with each other. VEs are interconnected to form a truly virtual network.
- *NodeOS* is an operating system for active nodes and includes facilities for setting up and management of communications channels for inter–EEs and AA/EEs, manages the router resources, provides APIs for AA/EEs, and isolates EEs from each other. Through its extensions the NodeOS offers facilities through the following components*:*
  - *Resource Control Facilities* (RCF). Through resource control, resource partitioning is provided and VEs are guaranteed that consumption stays within the agreed contract during an admission control phase, whether static or dynamic.
  - *Security Facilities* (SF). The main security aspects are authentication and authorisation to control the use of resources and other objects of the node such as interfaces and directories. Security is enforced according to the policy profile of each VE.
  - *Application/Service code deployment facilities* (ASP support). As flexibility is one of the requirements for programmable networks, partly realised as static or dynamic service provisioning, the NodeOS must support code deployment.
  - *Demultiplexing facilities* (DEMUX). As flows of packets arrive at the node, Demultiplexing filters, classifies and diverts active packets to the appropriate VE, and consequently to the destination service inside the VE.
  - *Node Management Facilities* (NM). The main aspects are the initiation and maintenance of VEs, control and management of the RCF and SF, management of the mapping of external to node policies into node resource and security configurations.

The following figure describes the main design features and the components of the FAIN nodes:

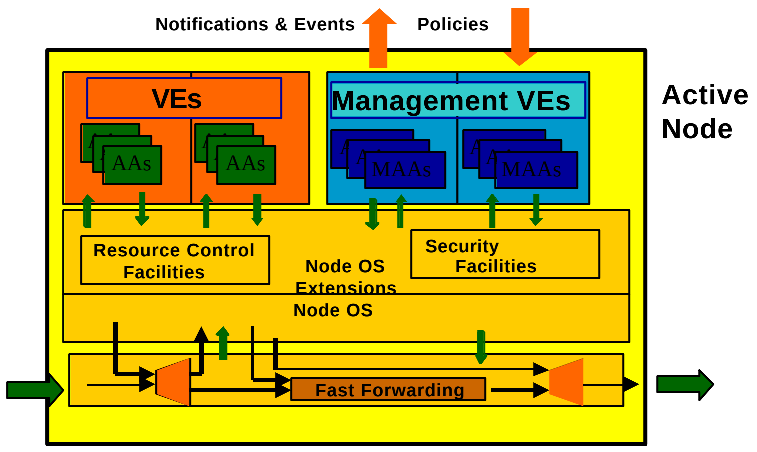


Figure 2: FAIN Active Node

In FAIN, node prototypes that are under development include: a high performance active node, with a target of 150 Mb/s; and a range of flexible and very functional active nodes/servers, with the target on multiple VEs hosting difference EEs

The common part of the prototypes (the FAIN middleware) is the NodeOS with the relevant extensions. Further details and discussions about the active node are provided in the deliverable D4.

1.1.3 FAIN Active Management Components

This deliverable D5 will elaborate on the management approach in the FAIN project, which takes policy-based approach.

We envisage that the management of the active network will require the following features:

- *Policies:* Description of policies required to manage the active nodes and network
- *Node management component:* Design of management components within the active nodes, which will execute policies within an active node and monitor the local node resource usage. The execution of policies means mapping target policies into node resource configurations
- *Management stations:* A set of management nodes that will provide mechanisms to enable network administrators to manage the active networks as a whole, including network policies set-up and processing.

As the delivery of services will require co-operation of a number of active nodes the network providers will need the means of managing the active nodes as a group of nodes and not individual nodes. They will need monitoring mechanisms for checking that correct policies are being defined and used in relation to the network before they are sent to the actual network. It will need to know what policies are currently loaded in the active nodes and what impact these are having on the network. It will also need to protect and monitor the security of the network. Therefore, the network/service provider needs a set of management mechanisms that will enable it to manage the network as a whole.

In FAIN we see the need for two types of management nodes in order to provide these mechanisms:

- Element Management Stations (EMS)
- Network Management Stations (NMS)

The main difference in functionality provided by these two types of management nodes is in the policy types, which they could process and manage, in the sub-networks, which they cover and in the creation of management domains for different types of users, as shown the Figure 1-4

The relationships between the EMS, NMS, and active nodes with regards to the policy flow are shown in Figure 3.

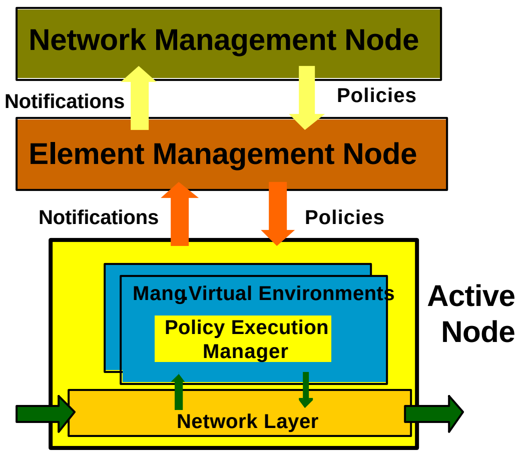


Figure 3: Active Network Management

1.1.4 Design characteristics for FAIN Prototype Nodes

In this section, we extrapolate WP4 efforts towards the milestone demonstrations in WP5, and references the issues highlighted in D6.

The FAIN active node runs active services and contains programmable management, data and control planes. We will develop three versions of this node until milestone *M6* (April 2003) and one version until milestone *M5* (May 2002). While active functions in the data and control plane will be demonstrated at milestone *M5*, active management functionality remains for demonstration at *M6*.

All node types versions will exhibit the similar functionality vis-à-vis services and management components, i.e. they will all support the active service provisioning facilities (ASP) as developed in WP4. They will be different, however, in their respective Node OS architectures and performance characteristics.

We will develop a single version of each of the two types of management nodes listed above. All FAIN management stations can interact directly with FAIN Active Network Nodes.

In the first phase of the FAIN test-bed leading to *M5* (May 2002), we will install, configure and evaluate only Active Network Nodes of type A, and the corresponding Element Management Station and Network Management Station. An initial version of an active node of type C will be developed and demonstrated by HEL. Intel IXP-based Active Network Nodes (type B) and full-featured Hybrid Active Router Nodes (type C) will still be under development at that time and will be ready for demonstration and evaluation as part of the work for *M6* (April 03).

As stated earlier, FAIN is developing two types of management stations, the Element Management Station (EMS) and the Network Management Station (NMS). Both management stations will have the same properties. The stations will be based in PCs with Linux OS. As programming platforms both stations need OpenORB and OpenCCM[1] CORBA platforms over which management components will be build. At the end of the project both management stations will be implemented over FAIN Active Node middleware (*i.e.*, the RCF) in order to be able to limit the amount of management station resources different management instances are consuming.

FAIN currently allows for only one NMS per network. Therefore only one NMS will be operational during the demonstrations; however, partners may set-up their own NMS for testing purposes in their own realm. One EMS may manage multiple active network nodes, which may be assigned dynamically. However, it is anticipated that each partner will run an EMS to be able to locally manage his active network node while testing.

[1] The use of OpenCCM platform is conditional of the availability of a new version of this platform during the project life offering the necessary functionality.

2 ARCHITECTURE REFINEMENT

In the deliverable D3 [8] of this project we presented already the architecture being designed and developed. This architecture is still valid nevertheless some changes have been pointed out as the project evolves. This chapter introduces and explains those changes that affect respectively the whole architecture view, the Management by Delegation and the management bootstrapping process. Finally the mapping of the Architecture and new concepts introduced so far in the FAIN enterprise Model is described in order to help understanding the whole system environment.

2.1 Global Architectural refinement

The management and active service provisioning frameworks are closely coupled in order to manage and maintain the active network infrastructure within the agreed quality of service parameters.

As initially delineated in the previous FAIN Deliverable D3 [8] document both frameworks interact at all layers (i.e. network, element and node). The concrete interaction description between both frameworks was already detailed in D3 and therefore it will not be repeated here. Although some changes have been made since then:

- *The Active Service Provisioning framework will be able to send Service Policies to the management framework.*

  These policies will allow the ASP to determine the best configuration for the Virtual Environment where the service will run, as well as configuring node facilities (such as the ASP or others) in a controlled way. E.g. a service policy may express that service X requests 50% of the CPU and another service policy lets service Y ask the node to download code module Y (z) and replace code module Y (y) with it, if the remaining memory for service Y is less than 100kByte.

  This modification has been included in the second situation exposed in D3, where the management framework was requested to install a service within the active network.

  As a result the function named serviceMap has been changed to:

  void serviceMap(in:ServiceID string, in:Credential Credential, in:ReqResources ResourceList, out: ServicePolicies ServicePolicyList);

  Also the ReqResources parameter has been changed from mandatory to optional.

- *Requests for the installation of active services requiring fast decision and allocation of resources are solved within the active node*

  In case that an active packet arrives to a FAIN Active Node with code, or a pointer to it, that should be installed in the node (third scenario in D3) there might be a latency restriction to process the packet. For example, in the capsules approach the capsules will probably require the fastest possible allocation, execution and forwarding. In those cases, contacting the management system (i.e. the corresponding EMS) to take a decision and allocate resources would be inefficient. The same thing can be said, when the packet does not carry the code, but just points to the code that has to process it. If the code has to be started with certain resources depending on the consumer, associated with the arriving packet (in particular with its credential), we have the same low latency restriction as on the capsules approach, in the sense that, since the code is started with different resources for each consumer, the decision has to be made very fast. Thus, making this decision outside the active node is inefficient (too slow).

The new approach designed to solve this problem is that for services that require fast allocation of resources, the management system will configure the Security Framework appropriately in order to allow the assignment of a certain amount of resources to the consumers. In that way, when the packet (either capsule or pointer) arrives to the node, the ASP will first ask the Security Framework whether the packet can be processed with the requested resources included within the active packet in the form of a node level security polic y. The active packet will also carry the consumer credential information used by the Security Framework for making the final decision. As such, if the Security Framework has been previously configured with the necessary information to accept or reject the request it decides accordingly and the decision is rapidly taken. Otherwise the ASP will contact the management system (i.e. the EMS) to make a decision and allocate resources (if necessary) as described in D3.

In Figure 4 the overall deployment of management and ASP stations for the maintenance and management of an active network.

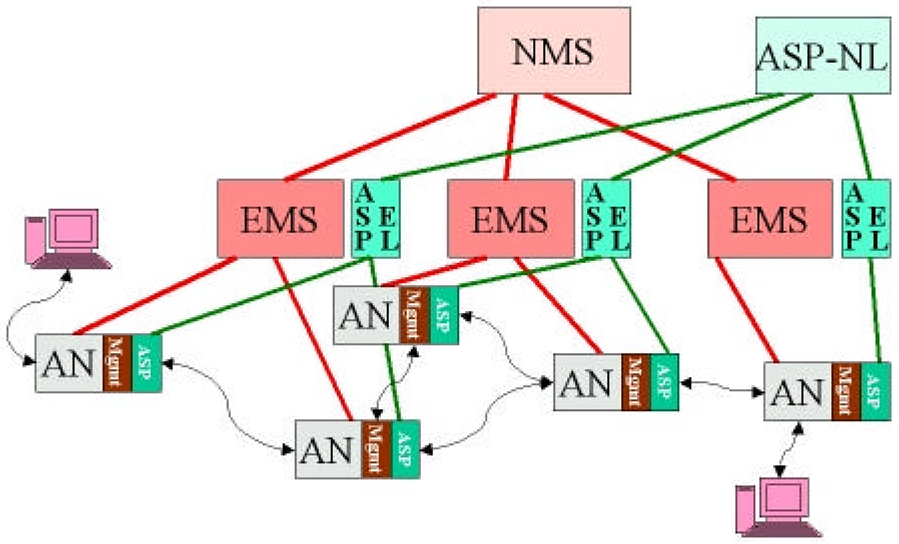


Figure 4 - Overall management and ASP stations deployment

Within this figure there are several aspects that need more careful description in order to ease the complete understanding of the image:

- Inside FAIN Active Nodes there will be certain management and ASP functionality. This functionality serves as helpful basis for the management framework to realise its functiona lity more easily.
- Aside, some components from the management framework, in concrete from the element management system (i.e. the PEPs and PEPDemux components) run within the corresponding Virtual Environment inside the FAIN Active Node. The reason for this is that it reduces the data traffic between the management stations and the active nodes and might even, if necessary, permit fast decision making within the nodes in punctual cases [10]

- In FAIN there will be only one network level management station per administrative domain, and a variable rate of active nodes per element management station with a minimum of one. However, in case that there is more than one active node per element management station, the management station will treat each one of them in a completely independent way (isolated from the others). The reason for this approach is that it allows the infrastructure owner to have a flexible trade-off between a cost-effective solution (several active nodes per EMSs) and a solution simpler, more distributed and more scaled (one-to-one ratio).

The flexible element management stations vs. active nodes ratio, which was already described in D3, altogether with the approach of including PEPs inside the active nodes, impose to the Policy Decision Points (PDPs) the necessity of being able to distribute its decision to the affected enforcement point. The detailed description of this functionality and others from the PDP is provided in subsequent chapters.

2.2 Management by delegation

The Management by Delegation (MbD) concept within the FAIN Policy-based Network Management (PBNM) system has been refined so that the initial PBNM architecture described in D3 was revised.

MbD was conceived to transfer the management logic from the central management system closer to the managed entity. This alleviates the management burden from the central management system [1]. The term "delegation" has been also interpreted in the context of Policy-based management. In [2] it is used to describe the transfer of access rights between subjects by means of delegation policies. Both interpretations of MbD have been incorporated in our revised architecture. Namely, we identify a component whose responsibility is the configuration of security components in order to delegate access rights. On the other hand, Yemini's MbD approach is embedded in the whole framework and it is not realised by a single component.

We extend the concept of MbD in the following sense: we allow multiple management systems to be instantiated With differentiated functionalities. Additionally, these different instances may be adopted by the different FAIN roles, e.g., a Service Provider (SP).

The full-flavoured management by delegation based on policies can be applied to the FAIN enterprise model in its entirety. The delegation of functionality interactions, that take place between the ANSP and the SP, represent a pattern for network and node management and may be repeated at every level of the business hierarchy, namely, between the ANSP and the SP[2], and between the SP and the Consumer.

The SP acquires part of the resources of the ANSP, and in a similar manner the Consumer receives part of the SP's resources. Furthermore, the SP may be presented with a variety of management options, ranging from installing its proprietary management architecture to installing varying instances of the ANSP's PBNM architecture. In the latter case, the SP has the advantage of starting from a specific set of management components that may further be specialised according to the SP's requirements. An advantage of this approach is that the PBNM management architecture may become a product that the ANSP sells to the SP. A similar situation may occur between the SP and Consumer.

[2] For simplifying the management concepts used within FAIN, we have used the simplest mapping of the enterprise model actors to the management system. The complete mapping description of actors to our management framework will be given in section 2.2.

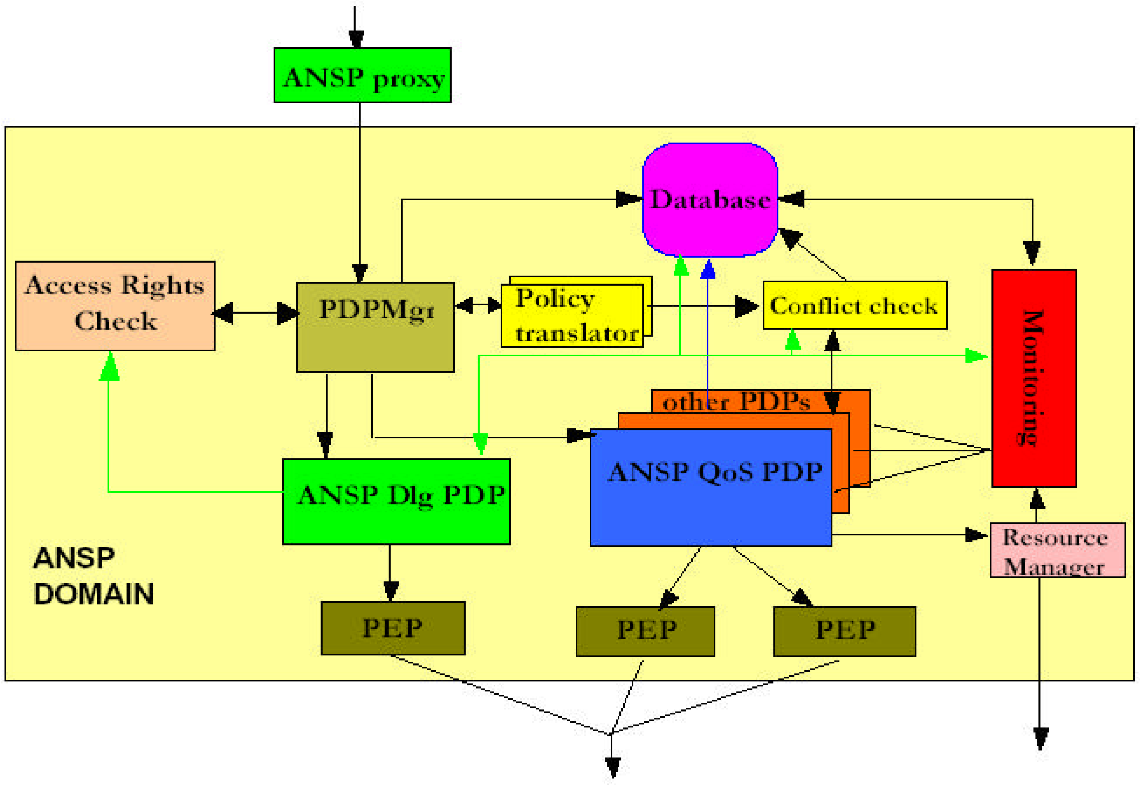


Figure 5: Policy-based Active Management architecture at the NL and EL

The management architecture covers both the network and element management layers, as defined in the TMN [3]. This approach aims to solve scalability problems and to provide a more distributed and autonomous management of active networks [[6],[7]].

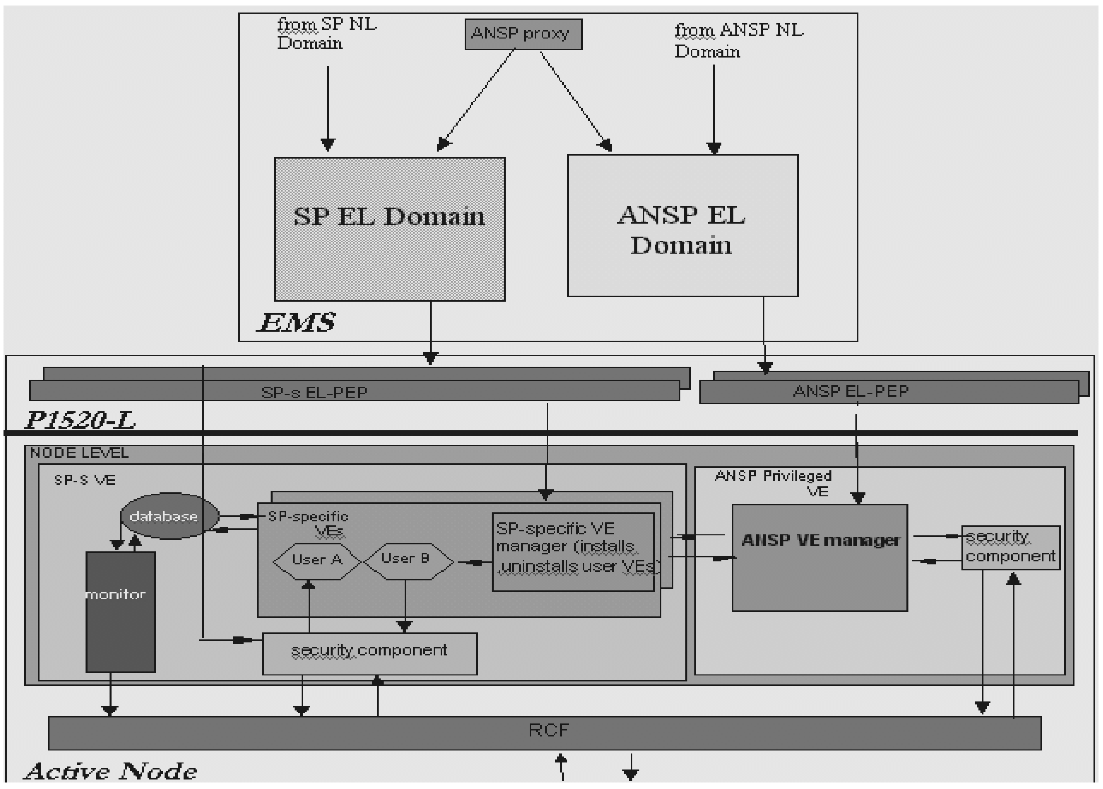


Figure 6. Delegation of management functionality at the Element Level

Figure 6 shows how the delegation of management functionality is realised by the creation of different management instances and its relation with the FAIN Active Node. This figure helps to understand how different management instances and node virtual environments are created. The management instances can be realised both at the network and element management level.

The ANSP management domain is composed of the components in the framework shown in Figure 5. Either the SP's own management components or instantiations of ANSP management components delegated to the SP can compose the SP's management domain.

The "management authority" is the owner of the management system that is delegating functionality to another subject or entity that we call "management instance".

The management installation procedure on the instance starts with functionality transfer, according to the access rights, from the management authority to the instance. That allows the instance owner to use the allocated resources for management purposes.

The management authority may request to control which entity enters the management framework. This requires security checks before any party attempts to enter the management system. For that purpose we have introduced a new component called the "ANSP proxy".

Access rights transfer involves the control of the management system itself. For example, if a policy-based system is used, the management instance should be given the access rights to control the use of the policies. The use of policies refers to the procedure that starts from the policy reception, decision and enforcement, as well as controlling the functionality of the policies, and ends by uninstalling the policies.

An additional access rights transfer from the management authority to the management instance involves the acquisition of rights so that the latter can access the actual managed resources, such as routers, switches etc. These resources initially belong to the management authority and are, then, allocated to the instance creating a Virtual Environment (VE).

In addition, the management authority also provisions physical resources (used for management purposes) to the management instance.

The requirement for instantiating virtual environments as a result of a virtual network deployment implies specific relationships between the ANSP's and SP's management architectures that need to be captured by the overall management framework. Accordingly, within the SP's virtual environment, its management architecture is instantiated by the ANSP, thereby, forming a parent-child relationship. Supporting such relationship requires introduction of an abstraction that we call the management domain depicted in as the ANSP and the SP management domains. Inside such domains the management architectures of the owners of the domain can be deployed/instantiated.

There are a lot of possible interactions that may take place between the ANSP and the SP according to the Service Level Agreement contracted. For example, the SP may use the management functionality of the ANSP as it is. For that purpose, the ANSP will create a new instance and it can maintain the total control of that instance itself (thus creating a new ANSP instance). Total control refers to the ability to control the complete usage of the policies (e.g., which policies can be set, by whom, and when should they be uninstalled...). Alternatively, it can delegate the control of the policy usage to the SP (thus creating a SP instance). It must be noted however that even when the SP has the control of the management logic, the ANSP still maintains the control of the management instance itself. For example if that instance is implemented in the form of a thread, the ANSP can "kill" that thread if the SP performs an illegal operation. Moreover, in order to make the whole system more security robust, the ANSP may keep the control of who is entering the management system at any time with the use of the ANSP proxy. In any case, in the SLA, the entity that controls the access rights both for entering the management framework as well as for controlling the use of the policies, should be clearly defined.

Consider another case when the SP wants to install its own management system. For this reason, the ANSP creates a SP instance where the SP can install its own management code. The ANSP can still delete or recreate the SP instance. Again in this case, it is important that the detailed interactions and the relationship between the ANSP and the SP should be explicitly defined in the SLA constructed between both parties.

2.2.1 Inter-PDP policy conflict resolution

Another area that has received a lot of attention since D3, is the inter-PDP conflict resolution. By inter-PDP we mean between policies enforced by different PDPs within the same administrative domain, e.g. the SP-domain.

Inter-PDP policy conflicts occur when two or more policies are eventually destined to access the same resources in a node from different PDPs conflict. If there is no co-ordination between these PDPs as far as conflict resolution is concerned, eventually there will be a conflict during the enforcement time.

With the conflict resolution mechanism at the network level we can resolve some of the conflicts at deployment time, which is of course far better that waiting until the enforcement time to capture a conflict.

We will try to avoid this kind of conflicts by defining a good information model that contains both complex and simple policies. In case this is not enough for avoiding conflicts we will solve possible appearing conflicts implementing one of the next two approaches that follow:

1. Based on policy translation
2. Based on a group of checkers

The first solution is based on the policy translation from the PDP-specific format of the policies to another one more generic that is common to all PDPs, which might conflict. This policy translation is realised by the PDPs themselves, and once the policy is translated, it is send to a generic conflict check component, that will check this *generic* policy against all previous policies (in the *generic* form) enforced before.

The second solution opts for a slightly different approach. Each set of possible conflicting PDPs will form a primary domain. Each primary domain will have a conflict check component. When a policy arrives to a PDP that might conflict with other policies in another PDP, this PDP will forward this policy to the corresponding primary domain conflict check component. This component will understand all possible conflicting policies from its related PDPs and will realise the conflict checking of the new policy against all previously enforced. When a new PDP is downloaded to the system that deals with policies that might conflict with others in one particular primary domain, the conflict-checking component of that primary domain should also be extended so that it can understand and check the *new* policies from the *new* PDP.

We ended up using one generic conflict check component, mainly because we wanted to avoid the direct communication between several PDPs for policy conflict resolution. The latter would most certainly lead to an increase in the complexity of the interfaces exported by the PDPs, and would also incur scalability problems by introducing a large number of messages that need to be exchanged between the PDPs. However, the use of a generic conflict resolution mechanism has drawbacks as well, as the scalability concerns are still not resolved.

2.2.2 Resource manager

The task of the resource manager (RM) is to assess the resource utilisation information that it has registered to receive from the monitoring system. This evaluation will drive short-term or long-term decisions for admission control, traffic re-routing, resource re-allocation etc.

So far we have designed the resource manager module only at the network level. However, we found out that it should also be located at the element level as well. This is because both the network level and the element level management system must work in tandem for resource management. Since the resource management algorithm runs inside the RM component, this algorithm can be distributed between the network-level RM and the element-level RM. The RM component will also be located in the ANSP and the SP both domains.

2.2.3 Management bootstrapping process

The bootstrapping process at every level of our management system, namely the network, the element and the node level mainly consists of two parts:

a) Deployment and loading of the code that creates the management logic and

b) Configuration of this logic so that for example, the ANSP PBNM will allocate resources for its own use.

The former falls within a wider scope of the ASP, whereas the ANSP PDPs captures the latter. In other words, the ANSP is entirely responsible for installing and initialising the management system in every part of the network needed. On the other hand, when the SP is assigned virtual resources, it is the SP that is responsible for the configuration of those resources (possibly with the association of the ANSP).

Accordingly, in a bootstrapping scenario we assume that the ANSP functionality is somehow deployed and we only need to describe how it is configured.

2.3 Mapping the Management Framework and FAIN Enterprise Model

The complete understanding of the management framework, and particularly of the delegation of management concept, would not be possible without a clear mapping of this framework to the FAIN Enterprise Model defined in FAIN Deliverable D1 [9]. The figure below along with the explanatory notes attached, describe this mapping and its main properties. Nevertheless, more detail regarding the actual architecture components designed to fulfil this functionality would be given in following chapters 4 and 5.

Figure 7 also eases the understanding of the management system use cases presented in the next sub-section.

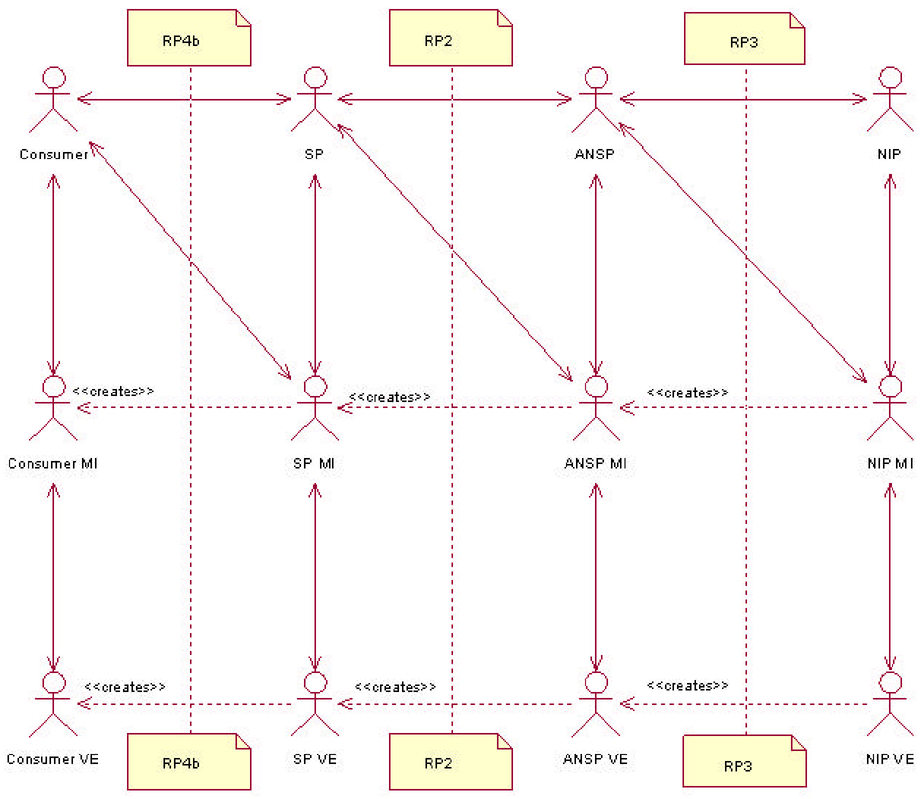


Figure 7 – Framework mapping to the FAIN Enterprise Model

In Figure 7 we can see the generic and complete mapping of the management framework and its implications (i.e. the delegation of management functionality approach) with the FAIN Enterprise Model defined in FAIN Deliverable D1[9]. The Network Infrastructure Provider owns the infrastructure that is, the programmable nodes (without Execution Environments) and the management framework to control the infrastructure.

After a Service Level Agreement with one or more Active Network Service Providers (ANSP), the NIP might partition its infrastructure between those ANSPs creating virtual networks over which they can install one or more execution environments. The access to this virtual infrastructure is offered to the ANSPs through their Virtual Environments (VE) created by the NIP (relation represented by the <<creates>> arrow in the figure at the Virtual Environment level). In any case the NIP always keeps a privileged Virtual Environment and management instance so as to have complete control over its infrastructure.

In order to allow the ANSPs to manage this virtual infrastructure the NIP also creates for the ANSPs new conveniently restricted instances of the management framework (represented by the <<creates arrow at the management instance, "MI", level). Although in some very special cases, the ANSP would also be able to manage its infrastructure using the NIP management instance (relation represented by the diagonal arrow in the figure).

All the possible interactions between the Active Network Service Provider and the Network Infrastructure Provider are contained in the FAIN Reference Point RP3. As shown in the figure these interactions can occur at different planes (business, management, node, etc.). It is also remarkable to note that the interactions realized between systems owned by the same actor (e.g. the ANSP MI and the ANSP VE) fall within the scope of internal interfaces of that partic ular actor.

In a similar way the Active Network Service Provider can partition its virtual infrastructure into smaller ones and offer them to different Service Providers that will use this smaller virtual infrastructure to install active services, which they will offer afterwards to consumers. The interactions that allow this new delegation of resources between the ANSP and the SPs are almost the same that in the previous step between the ANSP and the NIP. The only difference is that in this case there is the obvious limitation that the maximum resources and access rights that can be allocated to a SP by an ANSP are all the resources and access rights this ANSP has obtained from the NIP.

Again, all possible interactions between a Service Provider (SP) and an Active Network Service Provider are contained within the FAIN Reference Point RP2.

Finally, the same process can be repeated with the Service Provider creating an even smaller virtual infrastructure to its consumers. In case that happens the interactions between these two actors would be those contained in FAIN Reference Point RP4b.

Up to this point, we have described the generic mapping of the management framework to the FAIN Enterprise Model. As derived from the explanatory text above, this generic mapping is not straightforward. However, if we take into account some considerations we will soon realize that the mapping is not as complex as it seems.

First of all, it is quite likely that most of the times the Network Infrastructure Provider decides to install by itself one or more Execution Environments over its virtual infrastructure and act directly as an ANSP. If that happens the ANSP and the NIP will be the same actor and therefore the first delegation "level" which was between the NIP and the ANSP disappears. We would have just three actors and two delegation levels.

Now, let us consider that it is not really advantageous that a consumer obtains a virtual infrastructure from a SP, except for some very concrete services and highly prioritised consumers. We will find ourselves in a situation where the mapping between the most common enterprise model and the management framework is much more straightforward and easy to understand. It involves just three actors (NIP-ANSP, SP and C) and just one delegation level between the ANSP and the SP.

3 OVERVIEW OF THE MANAGEMENT SYSTEM ARCHITECTURE

This chapter gives an overview of the whole management system being built. In D3 we have already identified the three main sub-subsystems that composed the management system, namely PBANM, PBENM and ASP. Out of that we identified already a core policy based functionality that overlap both Network level and element level that have been abstracted as 2-tiers Policy Based Architecture. The following section outlines these sub-systems and their dependencies. The next section will describe the abstract functionality that will be inherited and specialize by the Network and element levels sub-systems in order to avoid redundancy when describing those sub-systems in relevant chapter.

3.1 Management System sub-systems

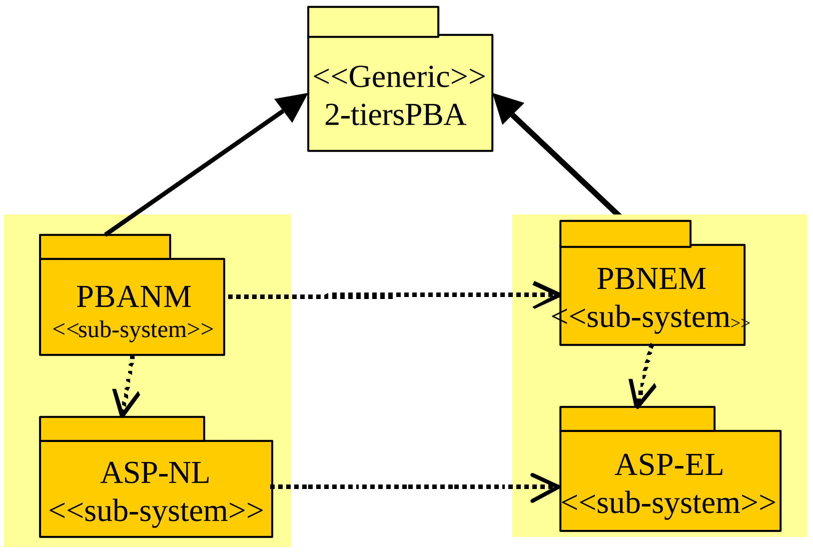


Figure 8 – Management framework system relations

The management framework designed in FAIN is a policy-based management framework at two levels: the network and the element level. Both this levels are specialization from a generic policy-based management framework adapted in FAIN, represented by the 2-TiersPBA Package in Figure 8.

The network level management sub-system (PBANM-NL) is the core of the whole management infrastructure, distributed in several element level management systems (PBANEM).

Both network and element management sub-systems are using the ASP sub-system that provides the code mobility management functionality completing, in the whole FAIN management infrastructure.

3.2 The 2-tersPBA Use Cases

This section describes the main generic use cases of the management system, which are shown in the Use Case Diagram of Figure 9.

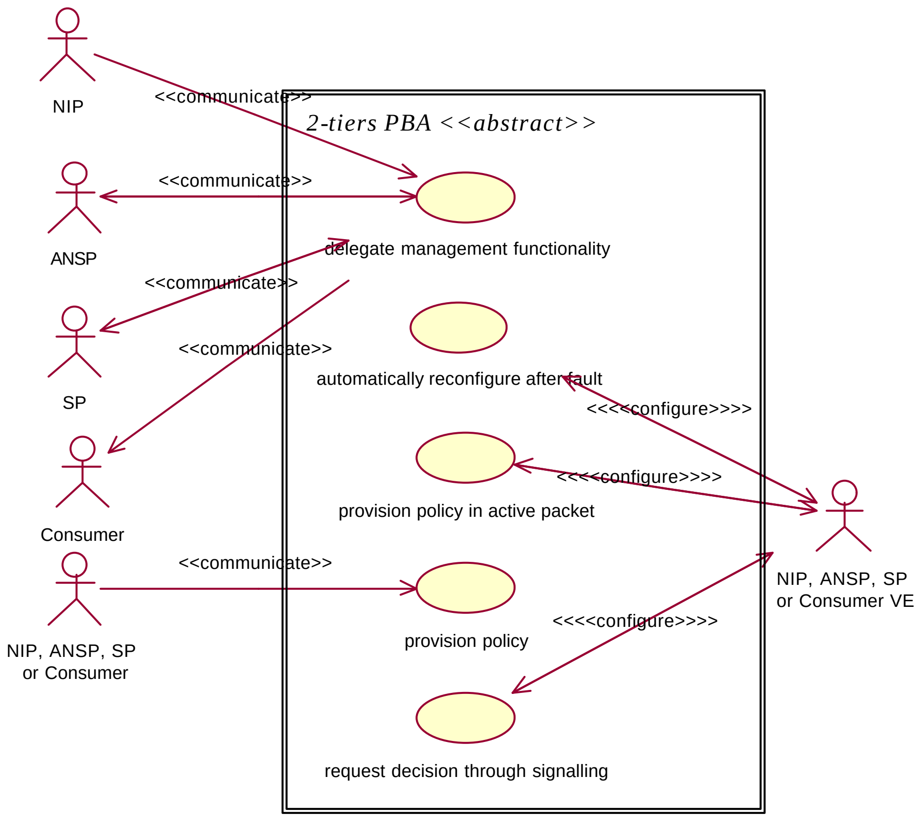

NIP
ANSP
SP
Consumer
NIP, ANSP, SP or Consumer
NIP, ANSP, SP or Consumer VE
2-tiers PBA <<abstract>>
<<communicate>>
<<communicate>>
<<communicate>>
<<communicate>>
<<communicate>>
delegate management functionality
automatically reconfigure after fault
provision policy in active packet
provision policy
request decision through signalling
<<<<configure>>>>
<<<<configure>>>>
<<<<configure>>>>

Figure 9 – Network Management Framework generic Use Case diagram

For simplicity reasons, use cases that are related either with all NIP, ANSP, SP and Consumer actors have been grouped in a single actor named “NIP, ANSP, SP or Consumer”.

As can be immediately deduced from the figure all use cases, and thus the functionality they represent, are supported by the generic policy-based management framework, and therefore by both the PBANM-NL and PBANEM systems that have already been introduced in the previous section.

Provision policy

This is probably the most important use case for a policy-based management framework. Together with the signalling use case they represent the basic functionality for policy processing in a policy based management system.

The provision policy use case encloses all functionality realised in our management framework every time a policy is introduced in the system.

The activity diagram in Figure 10 shows the main functionality within the provision policy use case.

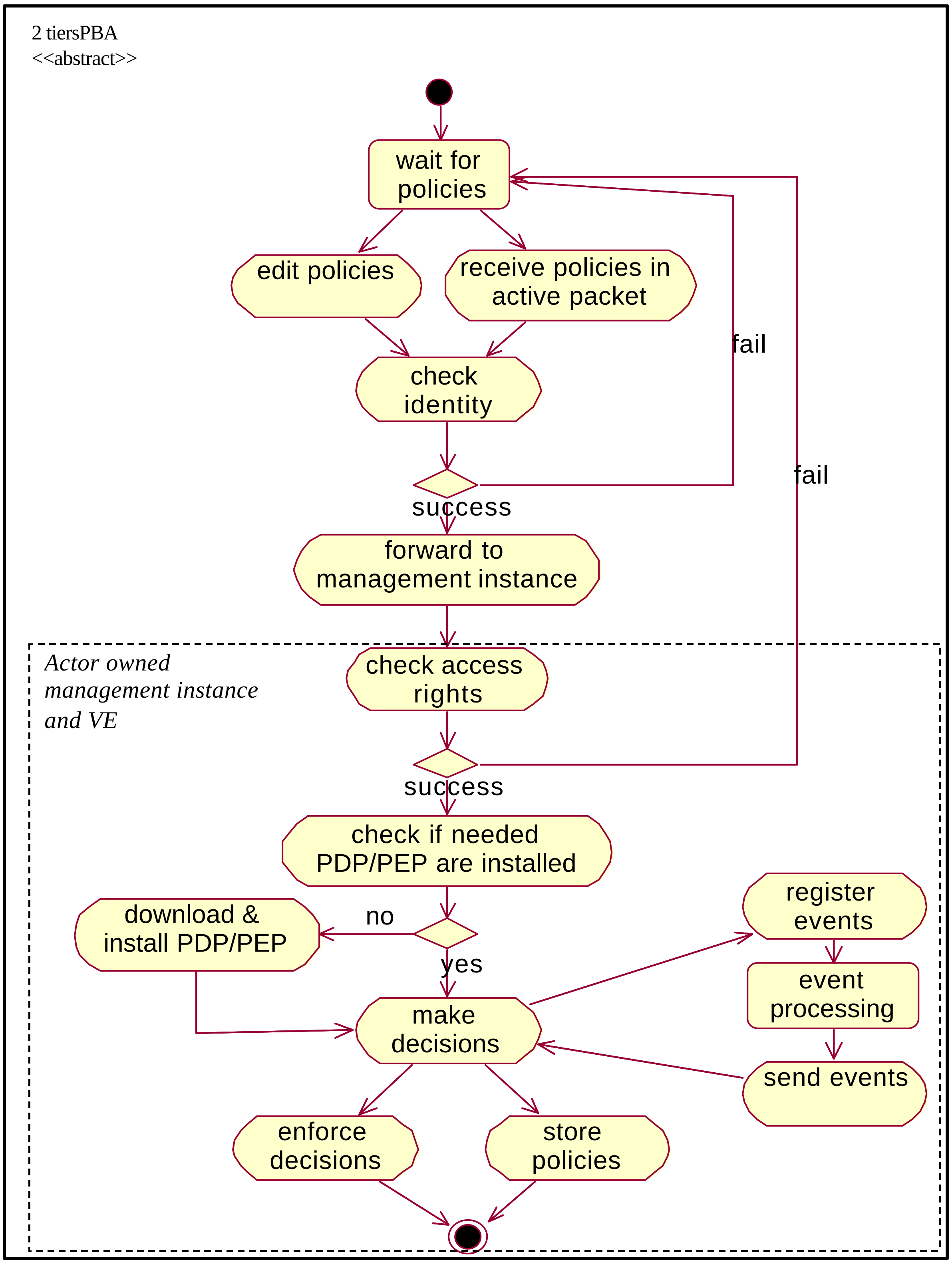


Figure 10 – Provision policy Activity Diagram

First, the pre-processing functionality is realised outside all management instances and realises: policy edition, checking of the identity through the credentials of the actor that pretends to use the management system, and demultiplexing of the policy to the corresponding management instance.

Once the policy enters the particular management instance the functionalities that will be realised[3] are mainly:

- Checking the access rights in the management instance of that actor (this functionality will only be developed in very special cases).
- Also in some cases, it might be necessary to extend the management functionality through the downloading of new components to correctly process the policy.
- Finally, the core policy logic functionality. This functionality encloses the most usual functionality of a policy based system: policy storage in the repository, making decisions on when a policy should be enforced based on events received through the event processing functionality and, finally, the enforcement of decisions. As a refinement of the latter, just at the network level specialization of the framework, the policies should be distributed to the appropriate element level systems.

Usually the result of the decision enforcement functionality ends in a set of configuration changes on the virtual environment owned by the same actor as the management instance that requests those changes.

[3] In case the actor chooses to install and use its own management functionality within its management instances the functionality realised might not be the one described here

Request decision through signalling

Another basic functionality of a policy-based system is that which covers the signalling approach; where the managed device requests through the policy enforcement point a set of resources to the decision point. Based on the resource consumption status, and on the policies available in the system, the policy decision point has to decide whether this request is accepted, and thus the resources are allocated, or rejected.

In Figure 11 we can see the main functionalities contained within the signalling use case.

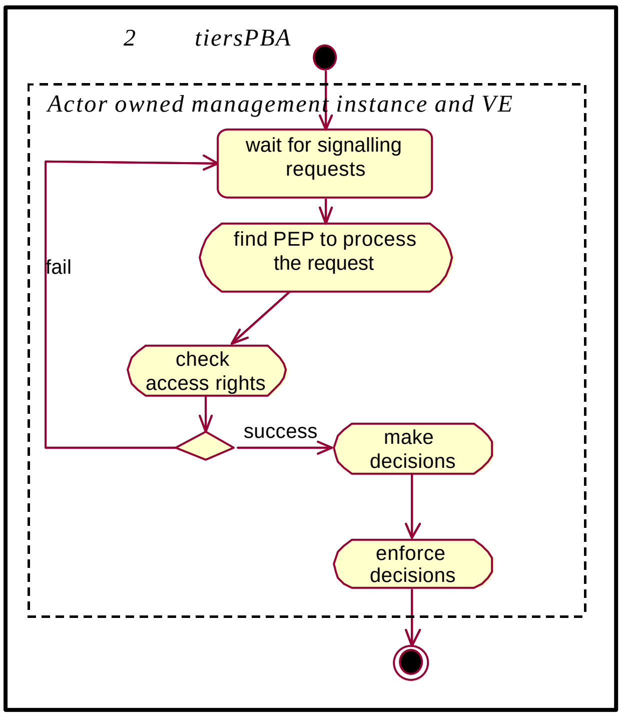

2 tiersPBA
Actor owned management instance and VE
wait for signalling requests
find PEP to process the request
fail
check access rights
success
make decisions
enforce decisions

Figure 11 – Request decision through signalling Activity Diagram

Since we might have several enforcement points within the same virtual environment of an active node, the first functionality to be realised, is the demultiplexing of requests to the appropriate enforcement point.

From there on, the functionality for processing the request is initiated. It contains basically two main functionalities: first, the access rights checking of the actor that is sending the request, and second, the concrete policy logic functionality related with signalling processing.

Finally, the latter is composed by the decision making functionality based on the resource status and on the policies installed within the system, and also by the enforcement of the decision which, obviously, includes the forwarding of the decision notification to the entity that made the request, whatever the decision is.

Delegate management functionality

The delegate management functionality use case is conceptually almost the same than the provision policy use case. The only difference is that, in this case, the provisioning actions are the creation of a virtual environment and a management instance for a new actor with certain access rights.

However, although being very similar to the already explained provision policy use case, the importance of the delegation concepts within our framework makes necessary the introduction and description of the delegate management functionality use case.

In Figure 12 we can see the main functionalities included in the `delegate management functionality` use case.

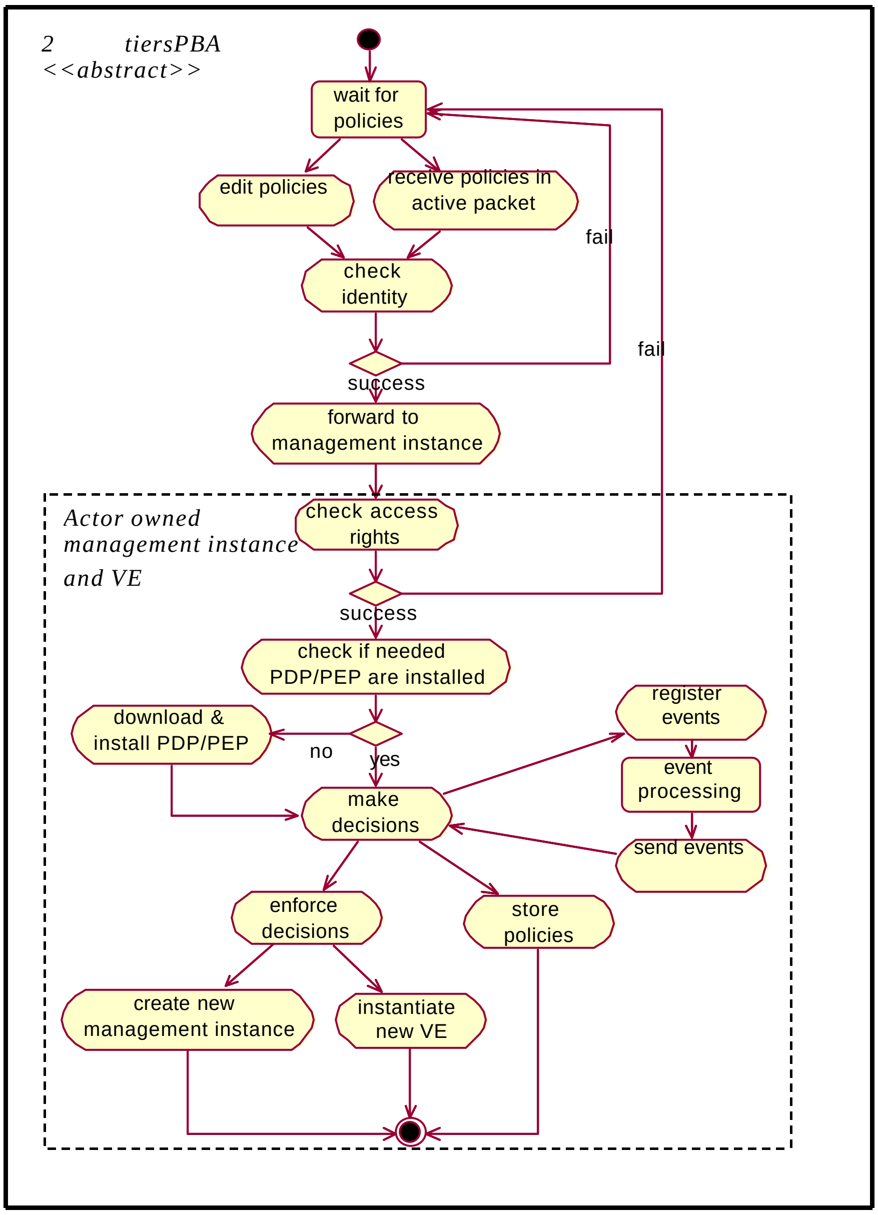


Figure 12 – Delegate management functionality Activity diagram

As said before, most of the functionalities are equal to the ones defined for the provision policy use case. For that reason we are not going to repeat them here. Nevertheless, we will try to highlight the main particularities of the delegation of management functionality use case.

As stated in the use case diagram, only the NIP, ANSP or SP actors can in theory realise this use case, not the Consumer since it cannot delegate management functionality to any other actor.

For the same reason, the result of the enforce decisions functionality are configuration actions over the virtual environments of the NIP, ANSP or SP only, since this configuration actions request the creation of the delegated virtual environment with the appropriate access rights.

Also, the enforce decisions functionality, in this particular use case, includes the functionality for creating the new delegated management instance within the management system for the actors that obtains the delegation of management functionality. Finally, those actors are notified with the handlers that they need to access both their virtual environment and management instance.

Provision policy in active packet

A new functionality we have introduced within the FAIN management framework is the one reflected in the provision policy in active packet use case. This functionality takes advantage of the fact of being managing an active network infrastructure in order to make even more flexible the distribution and introduction of policies in the appropriate management stations.

In this use case the policy is included within an active packet and forwarded to the nearest management station each time a policy arrives to an active node where it should be applied. The functionality reflected in the use case is that covered since the policy arrives to the management station.

Since the nearest management stations to active nodes are the element management stations, although the use case is in theory possible at both levels of the framework, it is in practice applied only at the element level.

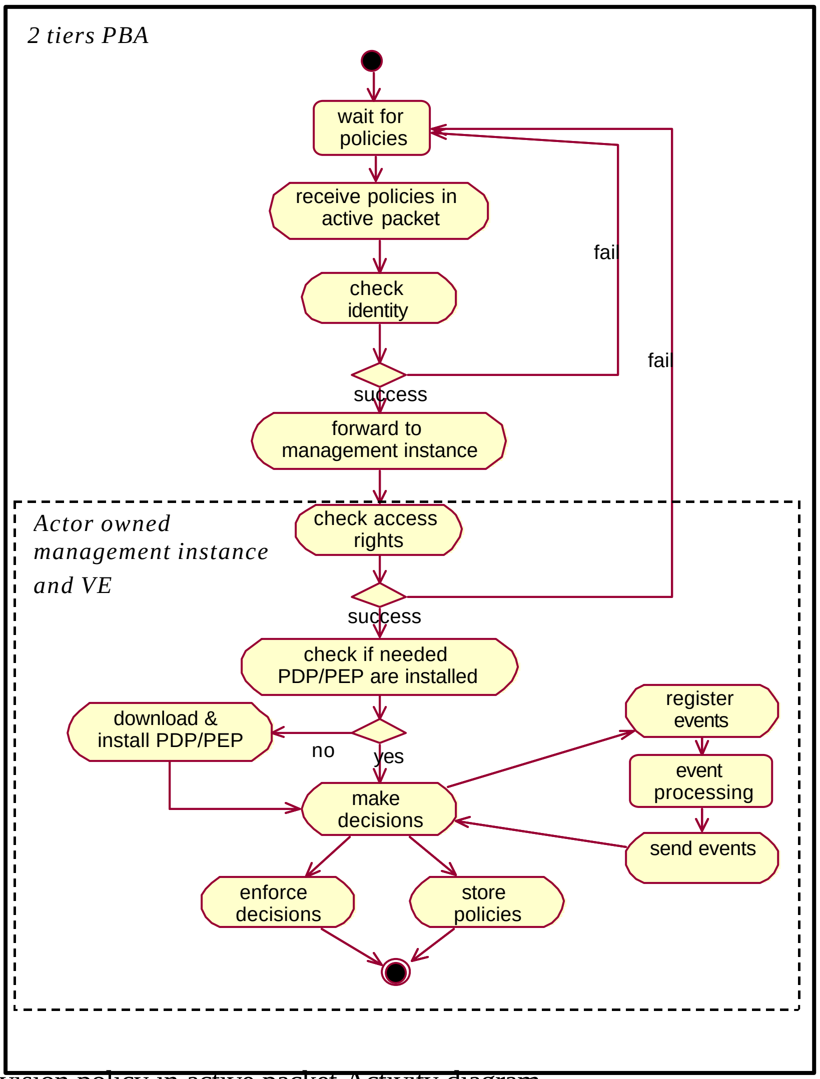


Figure 13 – Provision policy in active packet Activity diagram

The functionality contained in this use case is very similar to the one already described in the provision policy use case. There are basically two differences. The first is that the use case is initiated by the actor's virtual environment that forwards the policy to its management instance in the nearest management station. The second is as previously explained, in practice this use case can only happen at the element level.

Automatically reconfigure after fault

The functionality included within this use case is quite original if we compare it with features of others policy-based systems. This functionality copes with the necessity of readapting the active node and network configuration when a fault occurs.

The management framework, after the reception of the alarm warning of the fault occurred, will determine which are the policies that should be applied in order to correct the faulty situation. In that way we achieve an autonomous, distributed and fast resolution of problems and faults occurring in the active network infrastructure.

Figure 14 below illustrates the main functionalities included within this use case.

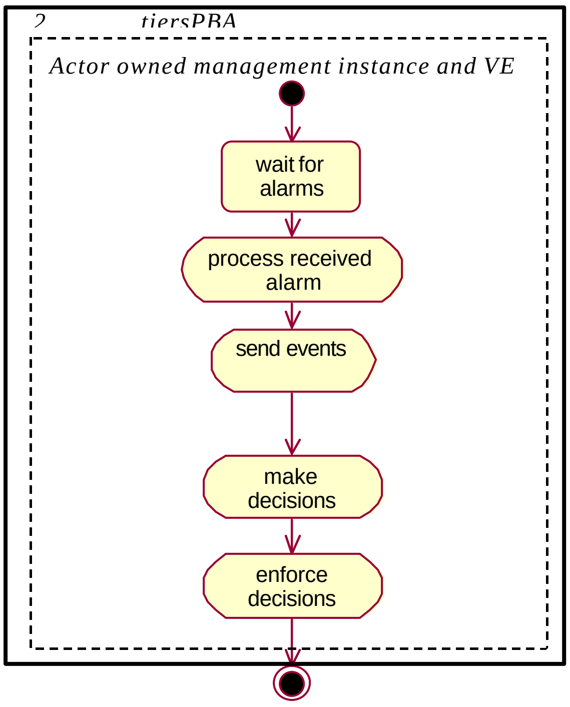


Figure 14 – Automatically reconfigure after fault Activity diagram

In this use case, the alarm processing functionality monitors the resources. When an abnormal or faulty situation occurs it creates an alarm event, which is then forwarded to the event processing functionality that communicates the alarm to the decision-making functionality. Based on the alarm event it decides which policies should be applied, and requests their enforcement.

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 necessity to store any new policy in the policy repository.

3.3 Components Overview

In this section we will briefly introduce the main components of the FAIN management framework relating them to set of functionalities they realise from all those we have seen in the previous section.

The main packages within the FAIN management framework have already been introduced in previous sections. Nevertheless, in this section we will map them to the functionalities they realised from the ones we discovered in the previous use cases so as to progressively introduce the reader to their capabilities and to the complete understanding of the architecture. A more accurate description of the components, the functionalities they cover, and how do they realise it will be given later on this document.

Through the description of the main use cases we have introduced several functionalities or smaller use cases. Many times these functionalities are repeated through several use cases (e.g. the make decision functionality appears in the five use cases). In order to make a much more simpler and comprehensible mapping of functionalities and components we will show which component realises which functionalities in the table below. This is possible, instead of on use case after another describing all functionalities and mapping them to the package that covers them, since the component that realise a particular functionality is the same independently of the use case where the functionality is found.

Functionality	**Component**
Edit policies	Policy editor
Check identity	ANSP Proxy
Forward to management instance	ANSP Proxy
Check rights	Access rights check
Dynamic management functionality extension	PDPMgr
Store policies	Policy Database
Make decisions	PDP (e.g. delegation PDP, QoS PDP, fault management PDP, etc.) and with the support of the Resource Manager and the Monitoring system components
Event processing	Monitoring system (i.e. the event channel)
Distribute policies	PEP (i.e. the PEP components at the network level)
Enforce decisions	PEP
Find PEP	PEP Demux
Create a new management instance	PDPMgr
Alarm processing	Fault management PDP in close coordination with the monitoring system

Table 1 – Functionality Vs. Component mapping table

4 R14 EMS (PBANEM) DESIGN

4.1 EMS 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, which are going to be covered, are the signalling, the policy within active packet and the fault-triggered management reconfiguration.

All three use cases have already been introduced in previous sections of this document. Nevertheless, we recover their description here in much more detail and particularized for the Element Management System (EMS), the PBANEM system.

Although the amount of functionality contained in these use cases is high, the majority have already been described before. Therefore, our main focus in this section will be to introduce the new functionalities advancing a new step towards the complete comprehension of the management system presented. The new functionalities introduced are either specific for the element management system or more concrete, and less important but necessary, compared against those presented before.

In the figure below a use case enclosing the three use cases presented in this section and their relations is presented.

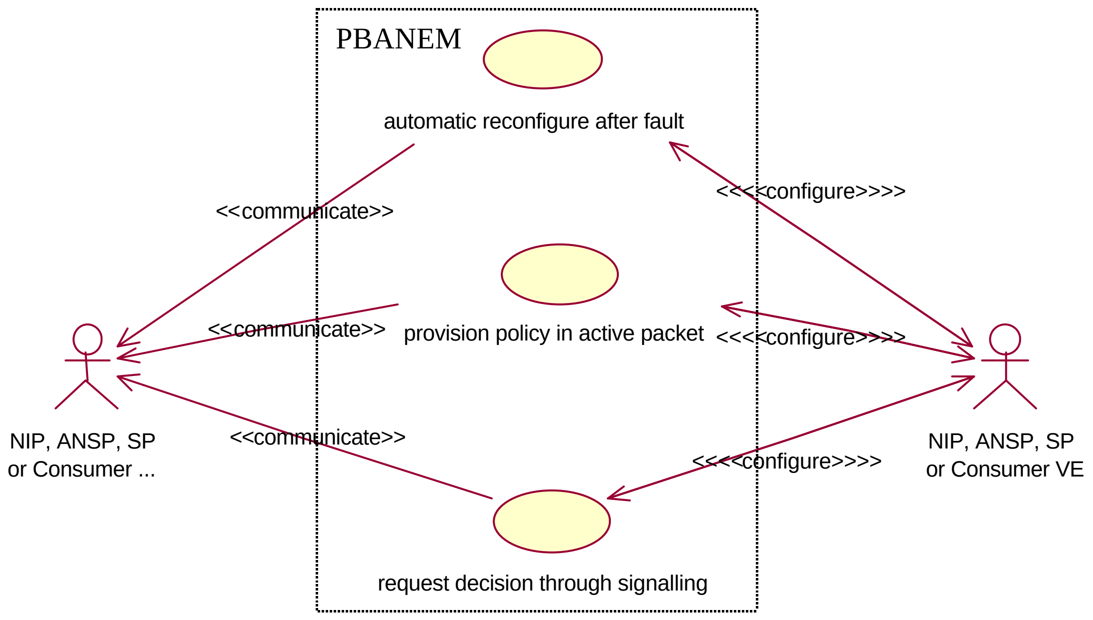

PBANEM
automatic reconfigure after fault
<<communicate>>
<<<<configure>>>>
<<communicate>>
provision policy in active packet
<<<<configure>>>>
NIP, ANSP, SP or Consumer ...
<<communicate>>
<<<<configure>>>>
NIP, ANSP, SP or Consumer VE
request decision through signalling

Figure 15 – General EMS Use Case diagram

Before proceeding with the actual description of the use cases, I will just note that the NIP, ANSP SP or Consumer MI at NL actor, stands either for the Network Infrastructure Provider, Active Network Service Provider, Service Provider or Consumer Management Instance at the network level. 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 in order to keep it informed of what is happening at the element level. In this way the network level always has a general view of the network resources and can act accordingly.

Request decision through signalling

The signalling use case particularized to the element management level includes three new functionalities that extend those described before. As can be seen in Figure 16, the new functionalities included are the demux decisions to PEP, dynamic conflict checking and notify configuration changes to NL MI.

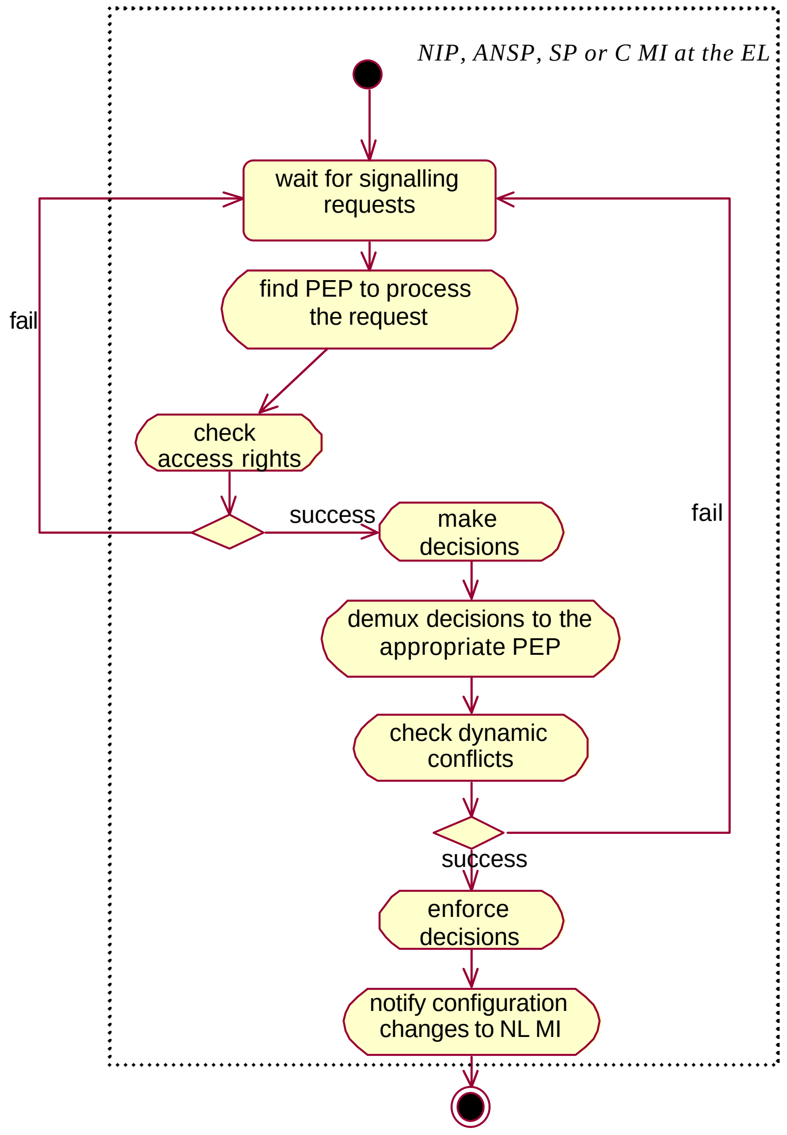

NIP, ANSP, SP or C MI at the EL
wait for signalling requests
find PEP to process the request
check access rights
fail
success
make decisions
demux decisions to the appropriate PEP
check dynamic conflicts
fail
success
enforce decisions
notify configuration changes to NL MI

Figure 16 – PBANEM Request decision through signalling Activity diagram

The demultiplexing decisions to PEP functionality extends the make decisions functionality previously described in the following sense. At the element level the policy enforcement points are located within the active network node, particularly within the virtual environment 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 ratio between decision points and enforcement points. Hence, the component which realises the make decision functionality at the element level should be extended with the functionality necessary to find the appropriate enforcement point to where this decision should be forwarded.

The second new functionality included in the diagram is the dynamic conflict checking functionality. This functionality is the responsible for checking possible conflicts between different policies in the precise moment where a policy should be enforced. The need of this functionality in policy-based system is justified in [11].

The dynamic conflict checking functionality is realised in part when the decision is made, and in part when the decision has to be enforced, that is the reason why it is in the middle of both functionalities in the diagram. The goal of the FAIN project is first to try to dynamic conflicts as much as possible making a clear and efficient allocation of resources, and keeping different allocations completely isolated from each other. However, if there are still dynamic conflicts the checking functionality will be realised in the this way: when a policy has to be deployed, it will be checked for conflicts, within the PDP, against other policies in that PDP, and if any conflict is found, it will be solved with policy priorities. If no conflict with other policies is found in this first step, the element management system will keep its normal process and will try to enforce the policy in the node. In this case, it might happen that the enforcement point finds out that there are not enough resources, that is, it detects a dynamic conflict. Then, 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.

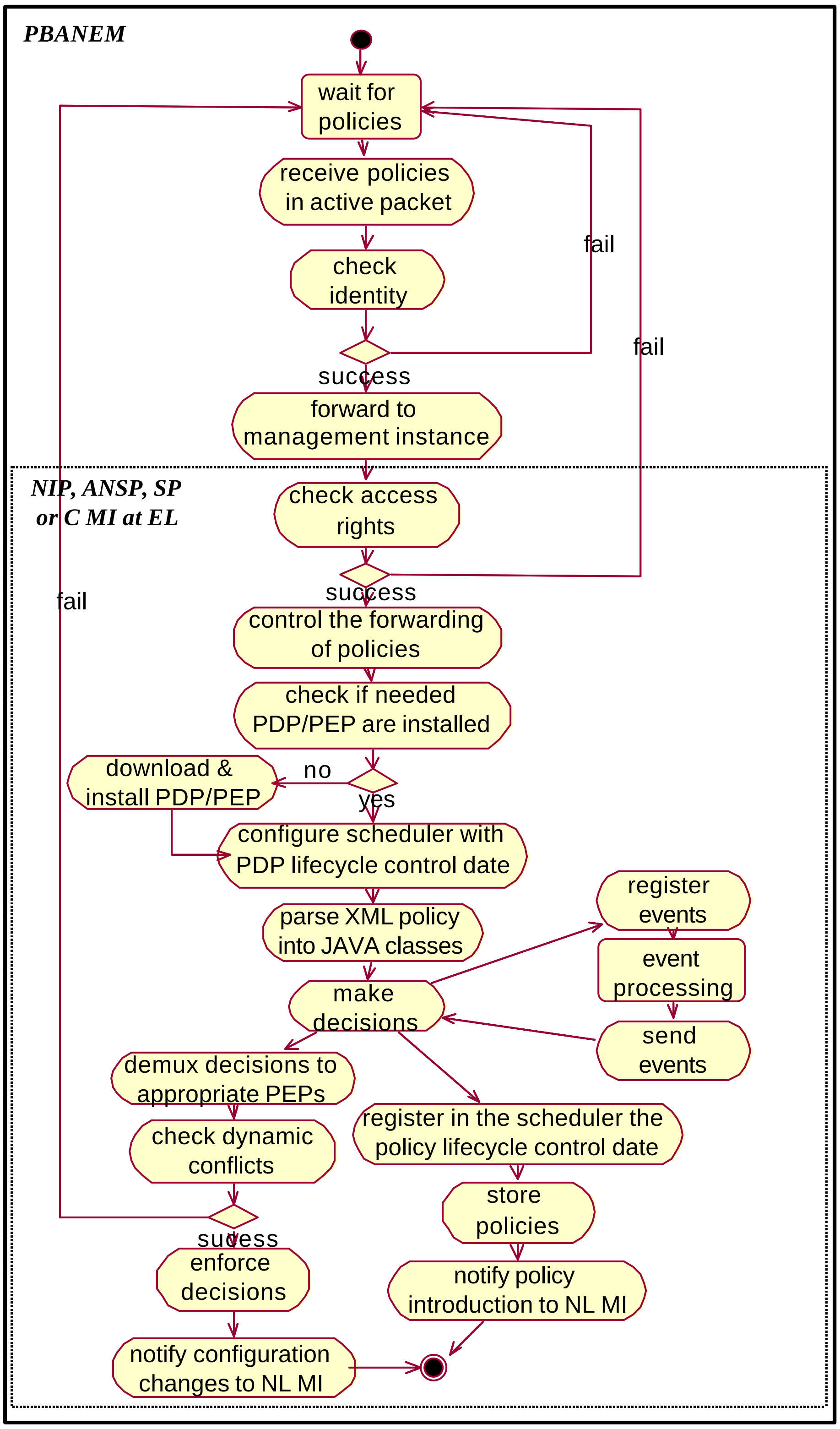
Finally, the last functionality included in the signalling use case at the element level being described, is the notify configuration changes to NL MI. This functionality is partly related with the previous one in the sense that when a reservation is made freeing resources of other actors because of a dynamic policy conflict, as introduced above, this functionality would be the responsible of sending the notification of the resource allocation and the freed resources to the network level, which will in turn forward it to the actors affected.

However, the above described will be the less common use of this functionality, usually this functionality will be used in order to inform the network level management instance, owned by the same actor as the notification originator instance, the configuration actions realised on the managed resources so as to keep the network level informed and allow it to have a general view of the managed resources.

Provision policy in active packet

The provision policy in active packet use case, despite of what the diagram shown below might induce, does not change substantially when particularized to the element management level. There are just some small new functionalities, which should be specifically added to the element management level, and some others, general for both the network and the element level, which are introduced now in order to provide some more information about the management framework functionality.

At the element management level the provision policy in active packet and the provision policy use cases are identical, except for the actor which initiates the request: the network level in the provisioning and the corresponding virtual environment in the policy in active packet use case. In consequence, all functionality description in this section applies as well to the policy provision use case at the element management level, which we are not going to describe explicitly.



PBANEM
wait for policies
receive policies in active packet
check identity
fail
fail
success
forward to management instance
NIP, ANSP, SP or C MI at EL
check access rights
success
fail
control the forwarding of policies
check if needed PDP/PEP are installed
download & install PDP/PEP
no
yes
configure scheduler with PDP lifecycle control date
register events
parse XML policy into JAVA classes
event processing
make decisions
send events
demux decisions to appropriate PEPs
register in the scheduler the policy lifecycle control date
check dynamic conflicts
store policies
sucess
enforce decisions
notify policy introduction to NL MI
notify configuration changes to NL MI

Figure 17 – Provision policy in Active Packet Activity diagram

In the diagram above we can see several functionalities not included in the generic provision policy in active packet use case described in a previous chapter. However, from all these functionalities only six of them are really new (i.e. the scheduler, policy forwarding control, the PDP lifecycle control, policy lifecycle control, parsing and notify policy introduction to NL MI). The functionalities notify configuration change, dynamic policy conflict and demux to PEP have just been described in the previous section. From the six new functionalities introduced only `the notify policy introduction` is specific for the element management system, the rest apply at both levels.

The scheduler functionality is just time trigger functionality. It is used by the PDP lifecycle control and policy lifecycle control to be triggered when either a PDP or a policy valid period expires. To do so it has, obviously, to be previously configured with the trigger time and the component to be triggered. Although not reflected in the diagram it might also be configured to trigger the make decision system when a time condition is met.

The FAIN policy-based management framework supports the introduction of groups of policy rules in the form of a policy set. The policy rules within a policy set might need to be applied atomically, sequentially, independently, etc. The policy forwarding control functionality is in charge of controlling the forwarding of policy rules into the management framework based on the forwarding property in the policy set.

The PDP lifecycle control is in charge of the maintenance of the policy decision points. It basically monitors the number of policies each policy decision point is treating. When a policy decision point is not processing any policy the lifecycle control functionality will remove it from the system freeing the resources that it was consuming.

The policy lifecycle control functionality is similar to the functionality above but it controls the lifecycle of policy rules within the management framework. Each policy rule has a “policyRuleValidityPeriod” property [12] that indicates the date when a particular policy expires. The policy lifecycle control system will register this date in the scheduler, and when triggered it will remove the policy from the element management system.

As described in the FAIN Deliverable D3 policies are expressed using XML language. In order to ease the manipulation and processing of policies these should be parsed afterwards to a JAVA class, this functionality is the parsing functionality in the diagram above.

Finally, the notify policy intro functionality aims to keep the network management level informed of what policies are being processed at the element management stations. Each time a policy rule is stored at an element level policy repository, this functionality is in charge of sending a notification to the corresponding network level management instance. In case the policy had been sent by the network level this notification would just act as confirmation, otherwise it informs that a new policy, with its main properties, coming from the virtual environment has been introduced in the element level management instance.

Automatically reconfigure after fault

The last use case we are going to describe in this section is the automatically reconfigure after fault use case. It contains four new functionalities specific for the element management level (PBANEM system). Three of them have already been described in previous chapters (i.e. the demux decisions to PEP, dynamic policy conflict and notify configuration change), only the send alarm to NL MI functionality shown in the diagram below needs to be described still.

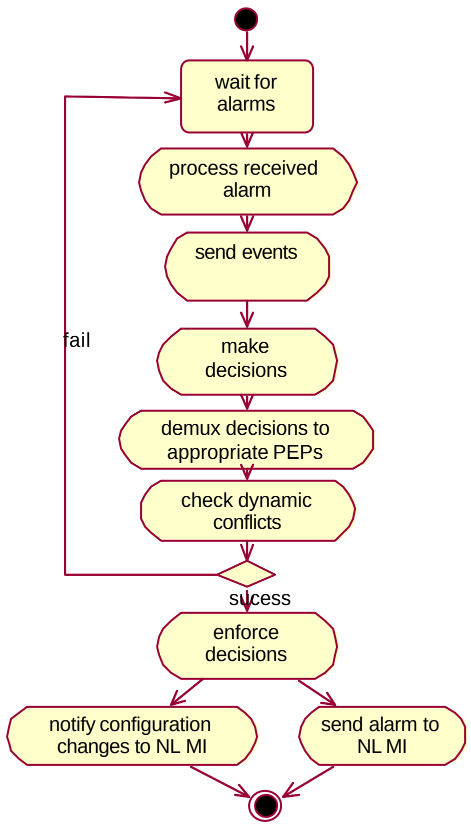


Figure 18 – Automatically reconfigure after fault Activity diagram

Both the send Alarm to NL MI and the notify configuration changes to NL MI can happen but never at the same time. In case a fault occurs within an active node virtual environment an alarm will be raised which will be captured by the corresponding element level management instance. The management instance will try to solve the problem, and in case it succeeds then the configuration changes realised 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. The functionality of creating the alarm notification and sending it to the corresponding network level management instance is the one included within the send alarm to NL MI bubble.

4.1.1 Components overview

In this section we will provide a table mapping the new functionalities introduced in the above described use cases with the components of the policy-based active network element management (PBANEM) system within which that functionality will be enclosed. All components listed have already been introduced in previous sections and they will be described in more detail later on this document.

The table below shows the mapping between the new functionalities described and the element management system (PBANEM) components that cope with them. This table complements the one given for the mapping between generic functionalities of the whole management framework (both element and network levels) and the components of the generic framework.

Functionality	**Component**
Demux decisions to PEP	PDP
Dynamic conflict checking	Part in PDP and part in PEP
Notify configuration changes to NL MI	PDP
Scheduler logic	PDP
Policy forwarding control	PDPMgr
PDP lifecycle control	PDPMgr
Policy lifecycle control	PDP
Parsing	PDP
Notify policy introduction to NL MI	PDP
Send alarm to NL MI	Fault management PDP

Table 2 – Functionality Vs. Component at the element level mapping table

4.2 EMS Components description

4.2.1 ANSP Proxy Component in Element Level

The ANSP Proxy at the element level works as a dispatcher of the policy data from the network level PEPs to the EL PDP Manager.

4.2.1.1 Use cases

When the ANSP Proxy receives the policy data from a PEP, it checks the parameters included in the policy data, such as VE id, and then finds an appropriate PDP Manager passing this VE id as well as a name, which indicates the domain. The ANSP Proxy also analyses reports from the PDP managers and may also create and send summarized reports to the operators. This reporting could be done directly to each SPs or through the ANSP in the NMS.

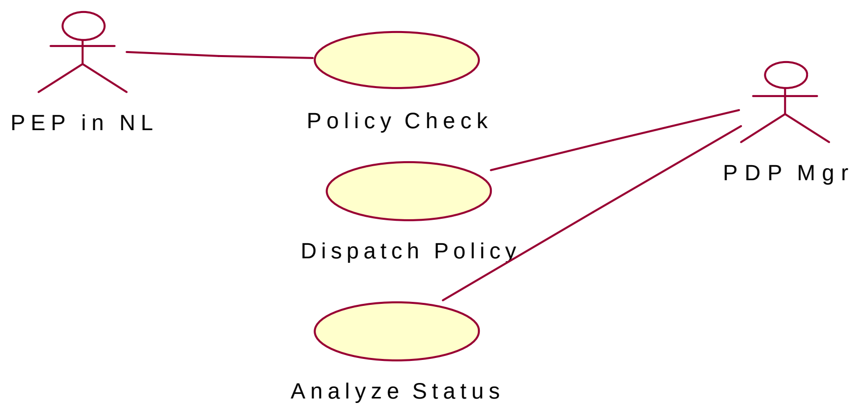


Figure 19: Use cases of the ANSP Proxy at Element Level

4.2.1.2 Class Diagram

The ANSProxyImpl class provides two methods: forwardPolicy() and setReport(). When a PEP in the network level sends policy data to the EMS, forwardPolicy() is called. After the deployment of policy data, the NL PDP may send reports to the ANSProxy with the setReport() method.

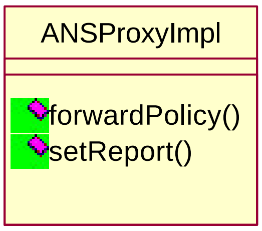


Figure 20: Class Diagram of the ANSP Proxy in Element Level

4.2.2 PDPMgr Component

The main functionality of this component is to demultiplexing received policies into the corresponding Policy Decision Point. Other important functionality are the PDP lifecycle control, controlled forwarding of policy sets and the PDP installation when an arriving policy needs to be processed by a non-installed PDP.

In the actual design the policy lifecycle functionality is realised registering in a database the latest caducity date of all policies enforced by that PDP.

In Figure 21 we show the general use cases of the PDPMgr component:

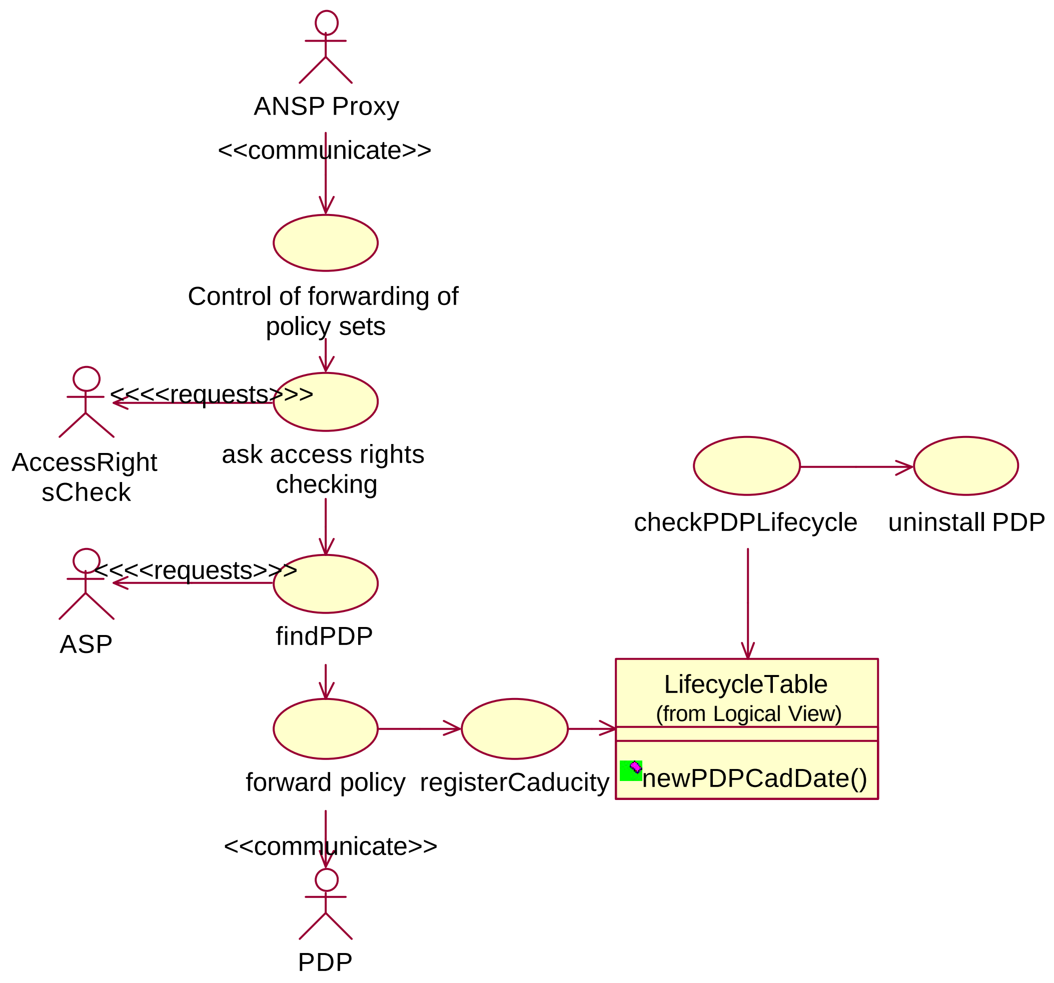


Figure 21 –PDPMgr Use Cases

The class diagram Figure 22 we have designed for this component is based on PForwControl and PDPmgrImpl classes, which realise most of the functionality helped by the other classes.

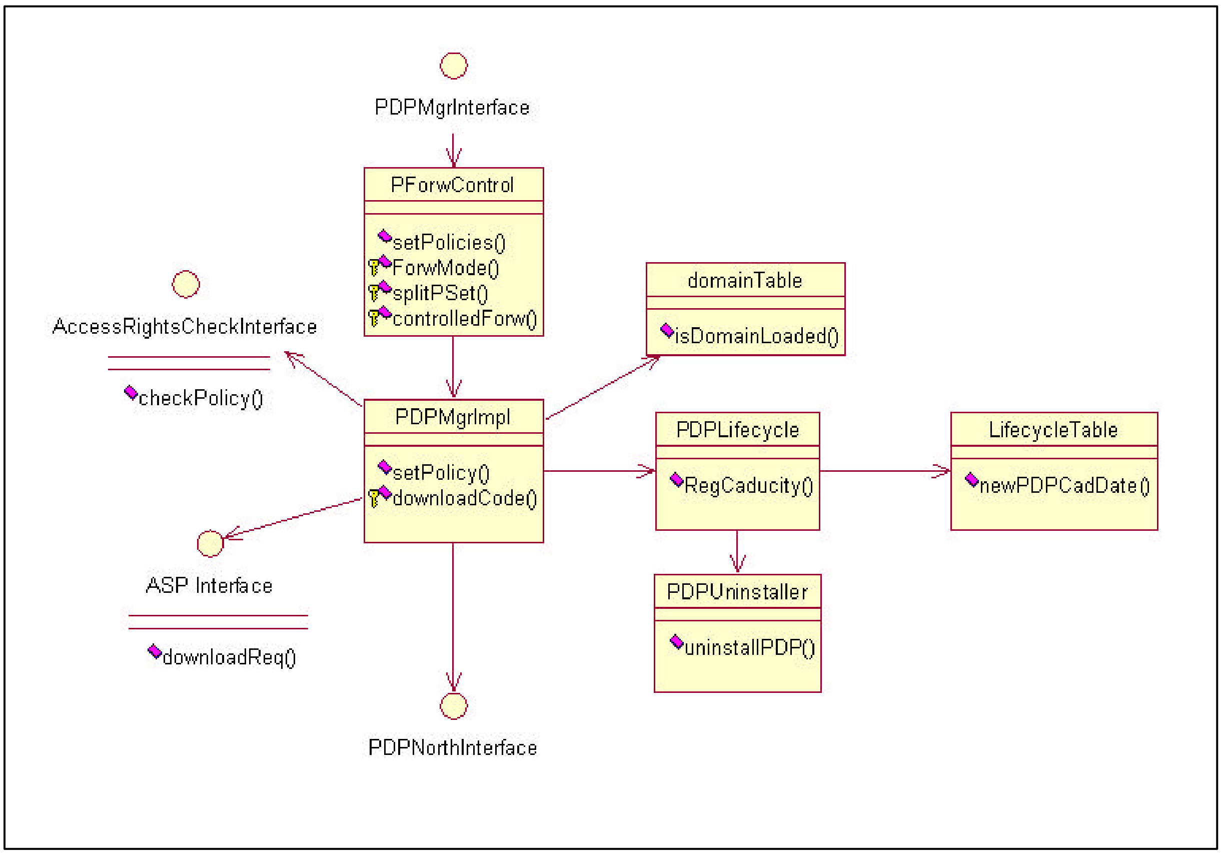


Figure 22 – PDPMgr Class Diagram

The PForwControl class implements the functionality of the "*control of forwarding of policy sets*" use case. When a policy set arrives, it checks whether it is just one policy or a policy set. in case of Policy set, it splits the set in individual policies, checks the forwarding mode of the policy set, and then forwards the individual policies accordingly.

The PDPMgrImpl co-ordinates the core behaviour of the PDPMgr. The mapping of classes with use cases is the next one:

· The PDPMgrImpl uses the AccessRightsCheck component in order to realise the "ask access rights checking use cases"

· It uses the domainTable class in order to realise the findPDP use case and if the PDP is not found, the PDPMgrImpl class uses its downloadCode private function in order to request the PDP downloading to the ASP component.

· The PDPMgrImpl class is also the responsible of forwarding the policy to the PDP once it is installed.

· After the policy forwarding it realises the register caducity use cases using the PolicyLifecycle class, which in his turn uses the lifecycleTable.

· Finally the PolicyLifecycle class periodically checks the lifecycle table in order to realise the check PDP lifecycle use case.

· If the PolicyLifecycle detects that a PDP has expired, contacts the PDPUninstaller in order to remove it fulfilling the uninstall PDP use case.

4.2.3 QoSPDP Component

The PDP component is the main component in a policy-based management architecture. Its main functionality is to check possible syntactic and semantic conflicts in policies (sometimes, even try to solve these conflicts[4]). Another piece of functionality of the PDP is to decide when a policy should be enforced. In order to realise this functionality the PDP needs to receive information from the monitoring system. The third important functionality is to forward decisions to PEP components in order to be enforced. Also, support for signalling requests; that is, realise the above-mentioned processes but for a signalling request. Finally, the PDP needs also to control the policy-validity period of policies in order to uninstall expired policies.

In Figure 23 we show the main use cases of a PDP component.

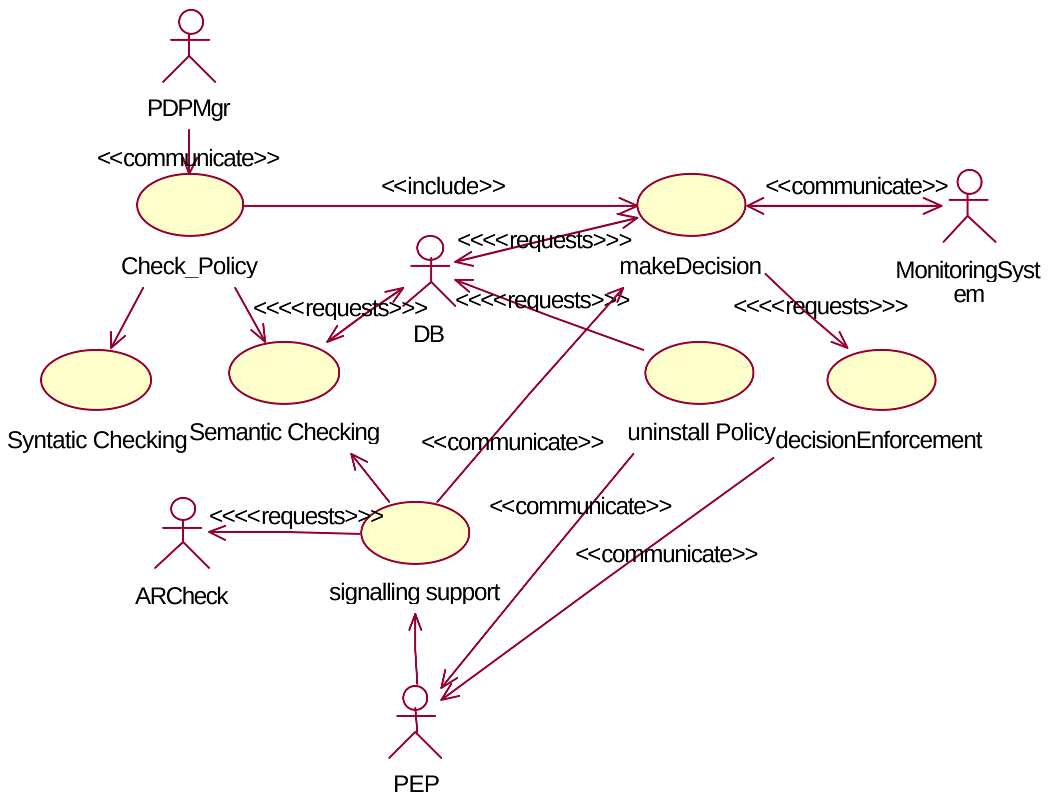


Figure 23 – PDP Component use cases

The class diagram we have designed for the PDP has its core in two classes the pdpQoSOpsImpl and the Evaluation class.

The mapping of use cases to the classes that realise that functionality is the following:

- Check policy: it is the pdpQoSOpsImpl class that after receiving the policy checks whether there are syntax conflicts (i.e. using its private function checkPolicySyntax). After that check it contacts the SemanticConflictCheck class to realise the semantic conflicts checking.

[4] We have not yet considered any semantic conflict resolution in our design, although in future versions we will consider it.

· makeDecision: this is the most complex use case. It is realised mostly by the Evaluation class but highly co-ordinated with the Condition_intepreter, EventRegister and EventInterpreter classes. It also uses the DBInterface and Scheduler classes in order to fulfill the functionality of this use case. When a policy arrives to be evaluated the Evaluation class forwards the policy to the Condition_interpreter in order to evaluate the conditions and know which information is needed in order to make a decision. If no information is needed a decision is made, if the policy has to be enforced it is forwarded to the ActionInterpreter class, and it is stored in the database as well as its validity date is registered in the scheduler.

The second possibility is that some information is needed in order to take the decision. Then the Condition_interpreter class will either configure the Scheduler (if it is a time condition) or the EventRegister (in case monitoring information is needed), and the Evaluate class will store the policy in the database and register the validity period of the policy in the Scheduler. Afterwards, the Event Interpreter will receive all registered events from the event channel, and map them to a java class format. Then, it will contact the Evaluate class, to re-evaluate the affected policies, and the whole process starts again.

· decisionEnforcement: we have already briefly described this use case before. The Evaluate class, when a decision has to be enforced contacts the ActionInterpreter class which builds the command that should be forwarded to PEPs and forwards this command to the command_demux class which finds and forwards the command to the correspondent PEPs (e.g. when we have one EMS per several nodes, and PEPs inside the nodes).

· uninstall: the Scheduler that contacts the PolicyUninstaller component when a policy has expired in order to uninstall it starts the functionality of this use case. If some actions are needed on the node to uninstall this policy the PolicyUninstaller component contacts the ActionInterpreter in order to realise them.

· Signalling support: this use case is reflected in signallingComp class. This class will create, with the help of the parser, the appropriate XML policy, contact the access rights checking to see if the requester is able to realise that functionality, and finally contact the pdpQoSOpsImpl class to continue with the rest of the process. It will forward the result of the policy processing to the PEP.

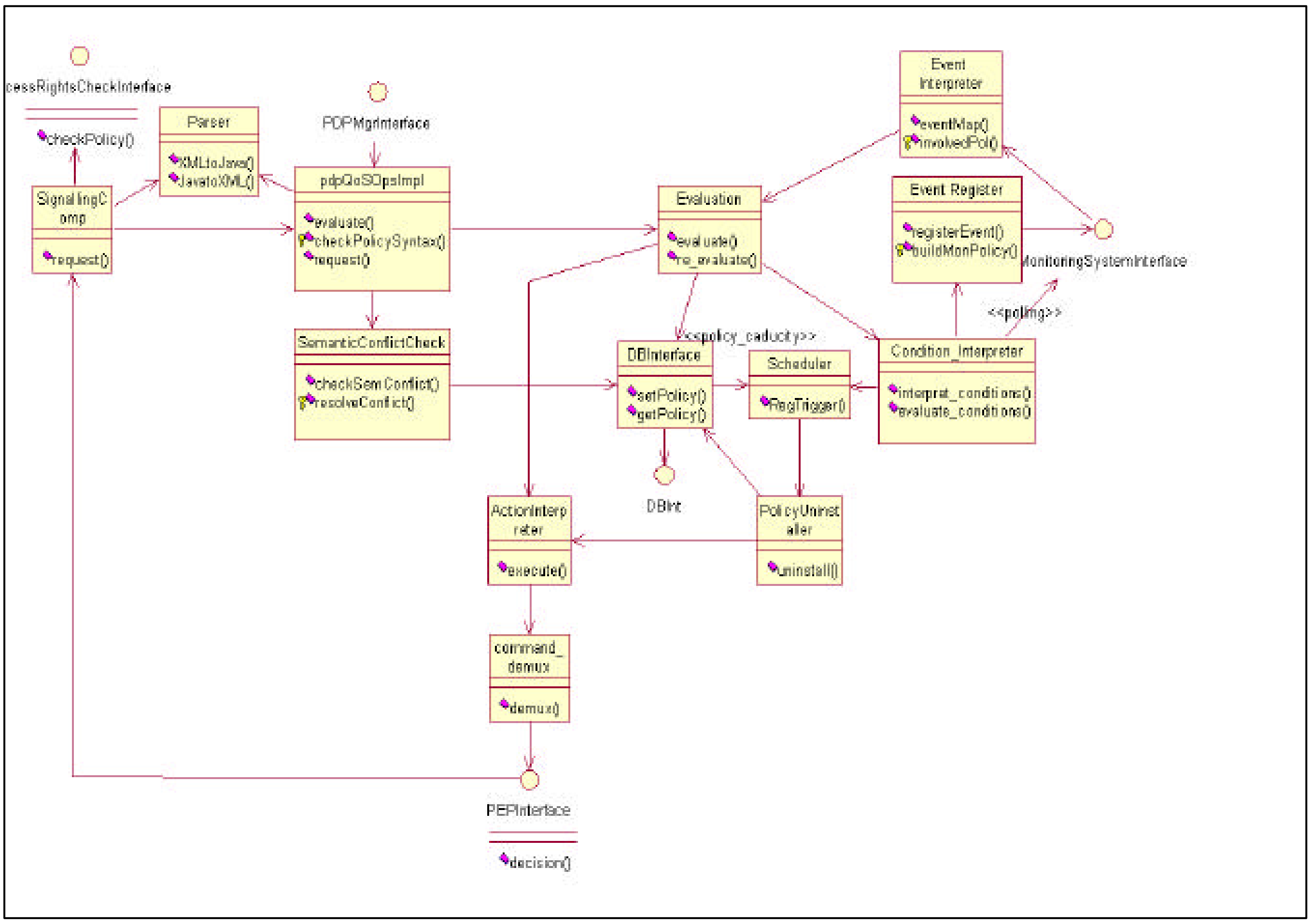


Figure 24 – PDP Component Class Diagram

4.2.4 QoSPEP Component

The PEP (Policy Enforcement Point) component is also a very important one in 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).

Figure 25 shows the main use cases of the PEP component.

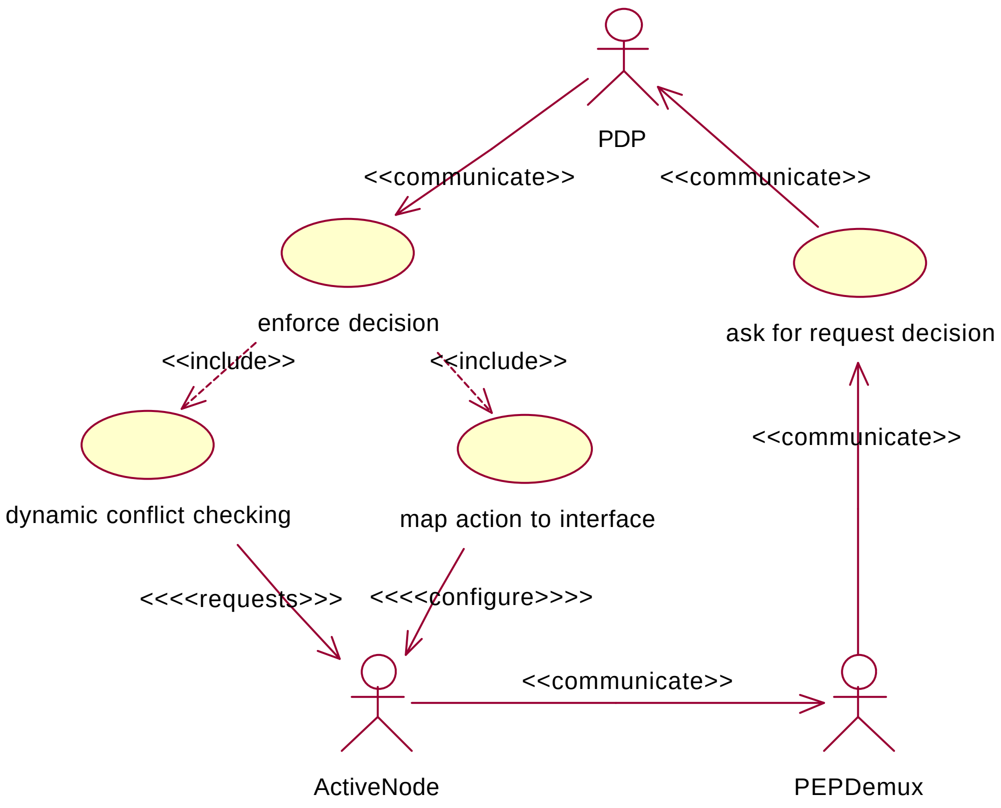


Figure 25 – PEP Component Use Cases

Figure 25 shows the classes that develop the PEP functionality. The main class for the decision enforcement is PEPImpl, while for signalling request forwarding is SignallingReq class. The mapping of the use cases to the classes that realise this functionality is the next one:

- Enforce decision: The functionality of this use case is developed by the PEPImpl class, which uses the classes DynCheck and IntMapper for realising the dynamic conflict checking and map action to interface use cases respectively.
- Dynamic conflict checking: The class DynCheck makes the dynamic policy conflict verification. For doing this job it might need to access the node interface.
- Map action to interface: The IntMapper is responsible for translating the decision into the actual commands understandable by the active node interface.
- Ask for request decision: The functionality of this use case is basically realised by the SignallingReq class which receives the requests from the node interface, parses them to the format understandable by the management system with the ReqParser class, and then forwards them to the PDP.

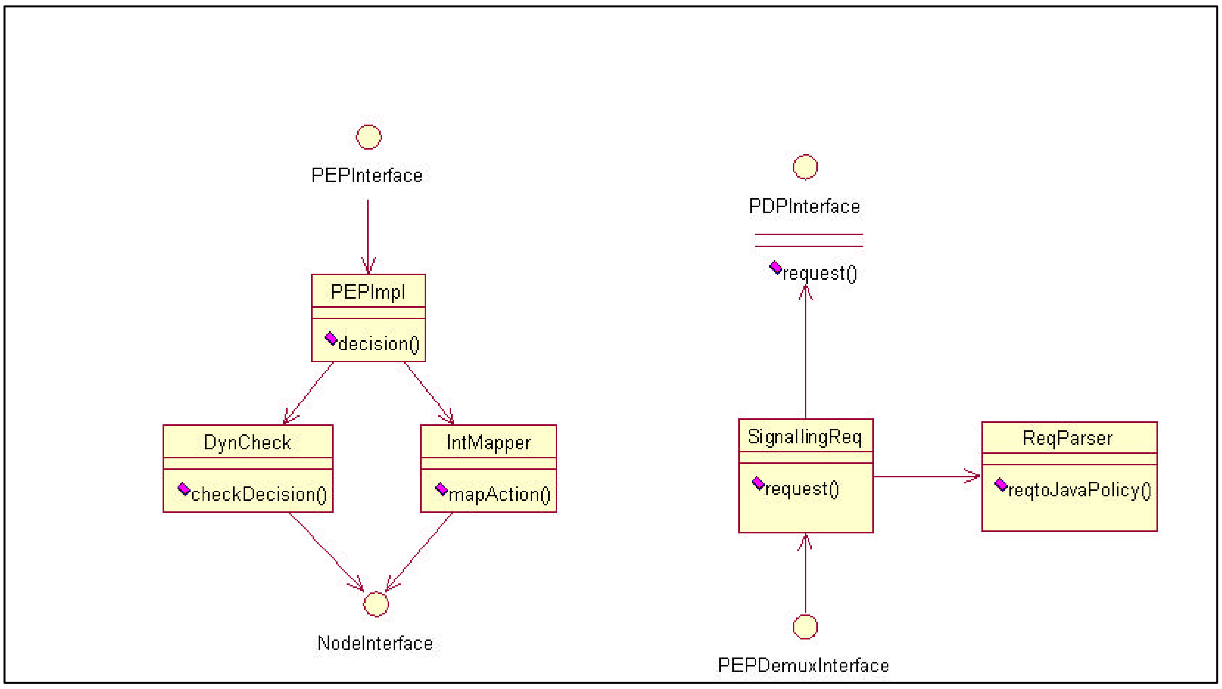


Figure 26 – PEP Component Class Diagram

4.2.5 PEPDemux Component

The functionality of this component is to demultiplexing active node signalling requests to the appropriate PEP. It would be just one PDPDemux component per active node and per management system instance.

The use cases for this component are shown in Figure 27.

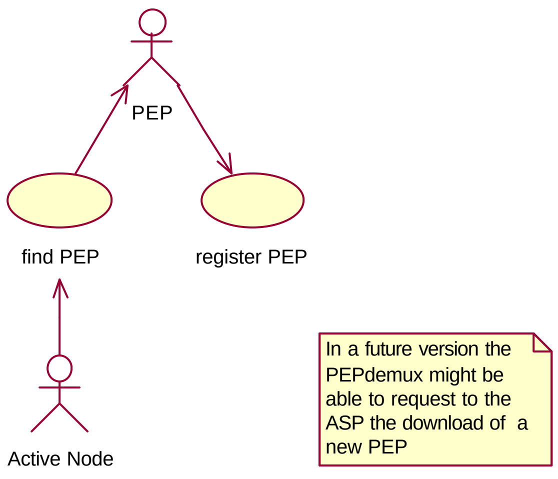

PEP
find PEP
register PEP
Active Node
In a future version the PEPdemux might be able to request to the ASP the download of a new PEP

Figure 27 – PEPDemux Use Cases

In Figure 28 we can see the class diagram for this component. The mapping of these classes with the use cases is indeed quite straightforward. In fact, the Demux class with the help of the PEPfinder class mainly develops the functionality of the findPEP use case, while the registerPEP use case is realised again by the same two classes.

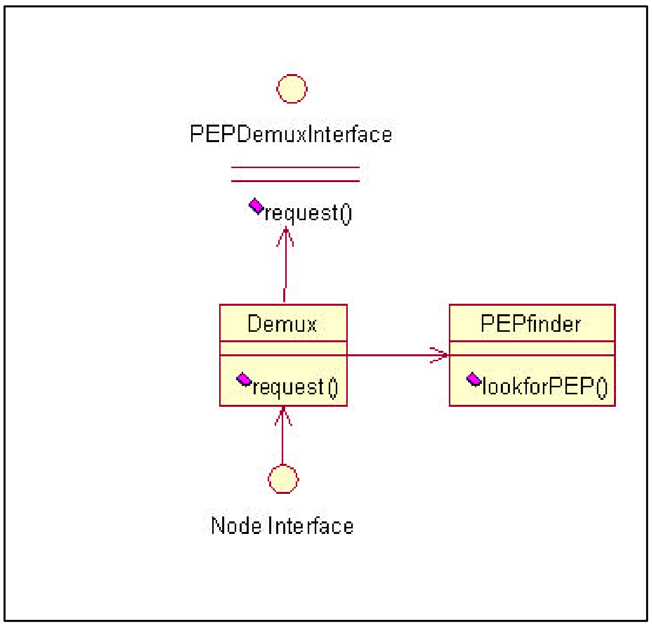


Figure 28 – PEPDemux Class Diagram

4.2.6 Delegation PDP Component

As we described before, Management by Delegation (MbD) has two types of main functions, 1) The delegation of management functionality, 2) the delegation of access rights. Delegation PDP is one of the key components in PBANEM system and is in charge of controlling access rights attribute of active nodes and of decision for what kinds of management domain is prepared for the customers (SPs). Both functions are achieved by interacting with PDP manager, monitoring system and delegation pep.

4.2.6.1.1 Delegation of management functionality

When a SP wishes to use the exact management system of an ANSP, the ANSP delegates management's functionality to the SP in one of the two ways described above.

The delegation of management functionality is not realised by a single component in our framework. The PDP manager is actively involved in the whole procedure, since it is the component that co-ordinates the other PDPs.

4.2.6.1.2 Delegation of access rights

The delegation of access rights involves the configuration of a security component. This task is done by the delegation PDP with the use of delegation policies. For example, the ANSP may want to release specific parts of its node management interface to the SP. Eventually, the element level Delegation PDP will configure the node-level **security components** that exist in every active node that the SP wants to manage. The security component, based on the value of an VE will grant or deny the access rights.

4.2.6.1.3 Definitions of the Delegation Types

The following options are provided by an ANSP for a SP to choose during the SLA process. These options could be chosen several times and are mapped into policy definitions, and then are translated into node readable parameters.

(1) Superuser

This allows a user to own his own management area. Access attributes to this management area are READ/WRITE. Expiry period is defined as well. This user can also delete another management areas. Usually this type is only assigned to ANSPs.

(2) High

This allows a user to own his own management area and to use/copy the super-user's management functionality's and tailor it to its own needs. The access attribute to this management area is READ/WRITE. Expiry period is defined as well.

(3) Monitor only

This allows an SP to access somebody's management area. The access attribute to this management area is READ only. This type could be used by a user who intends only to monitor (e.g. a billing service provider).

4.2.6.2 Access Rights Control

Delegation PDP is in charge of mainly controlling the access rights of the active nodes collaborating with the security framework.

4.2.6.2.1 Access Rights Definition

The access rights attributes define what type of accesses are allowed for the customers (ANSP, SP) on the active node interfaces. These access attributes will be verified by the security (SEC) component in order to avoid malicious accesses.

4.2.6.2.1.1 Attributes

(1) ReadWrite

A customer can configure the interface of an active node (RCF, Demux) and monitor the information of an active node (RCF, Demux) through the interface.

(2) ReadOnly

A customer can only monitor the information of an active node (RCF, Demux) through the interface.

(3) Disable

A customer has no access rights. While the attributes are disable d, the customer cannot configure or modify the active node interface nor can he monitor the information.

Also each component (RCF, Demux) of an active node may provide an access right attribute explicitly, so a customer may choose the attribute with the following matrix:

Node Component	RCF	Demux
ReadWrite	✓	
ReadOnly		✓

- RCF access (Write)

This allows a customer to access the RCF interface in order to configure and modify the resource allocation. The configuration of the RCF would be done by the QoS PDP.

- RCF access (Read)

This allows a customer to monitor the status of the resource consumption using the RCF interface.

4.2.6.2.1.2 Assigned groups

An access rights attribute is assigned for the VE. Moreover, additional definition could be supported. For instance, one VE involves several active nodes (see pictures below) and the customer of this VE may need to assign distinct attributes for each active node. So the Delegation PDP may support the following definitions:

(1) VE

An access rights attribute can be assigned only for each VE. In this case, all of the active nodes, which are involved in a certain VE, have the same attributes.

(2) Group of active nodes

Some active nodes could be categorised in groups and this group may have the same access rights attributes. We propose two groups of active nodes, namely edge node group and core node group.

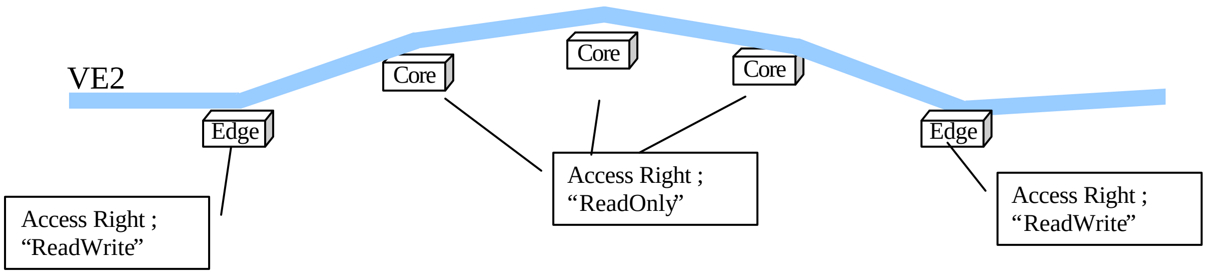
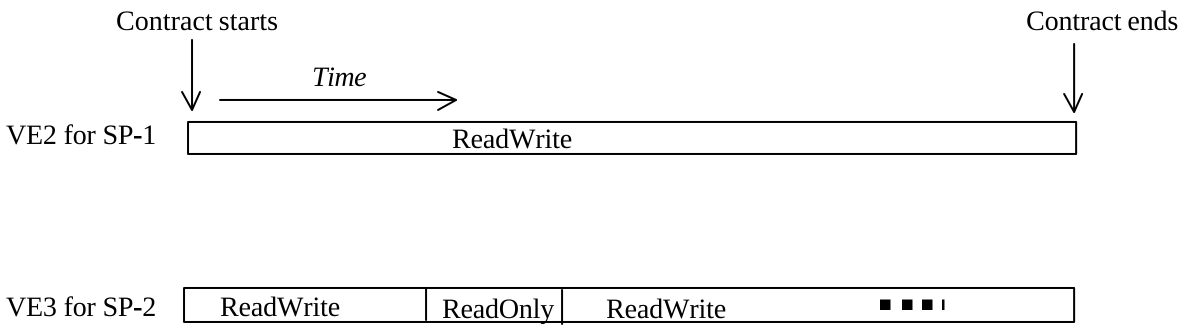

VE2
Edge
Core
Core
Core
Edge
Access Right ; "ReadWrite"
Access Right ; "ReadOnly"
Access Right ; "ReadWrite"

Figure 29: Access Rights for Nodes

4.2.6.2.1.3 Period of access rights

The Customers configure the access rights attributes during the contract period. For instance, the attributes may not be changed until the contract has expired. However, the customers may wish to configure the access rights based on a time-table:



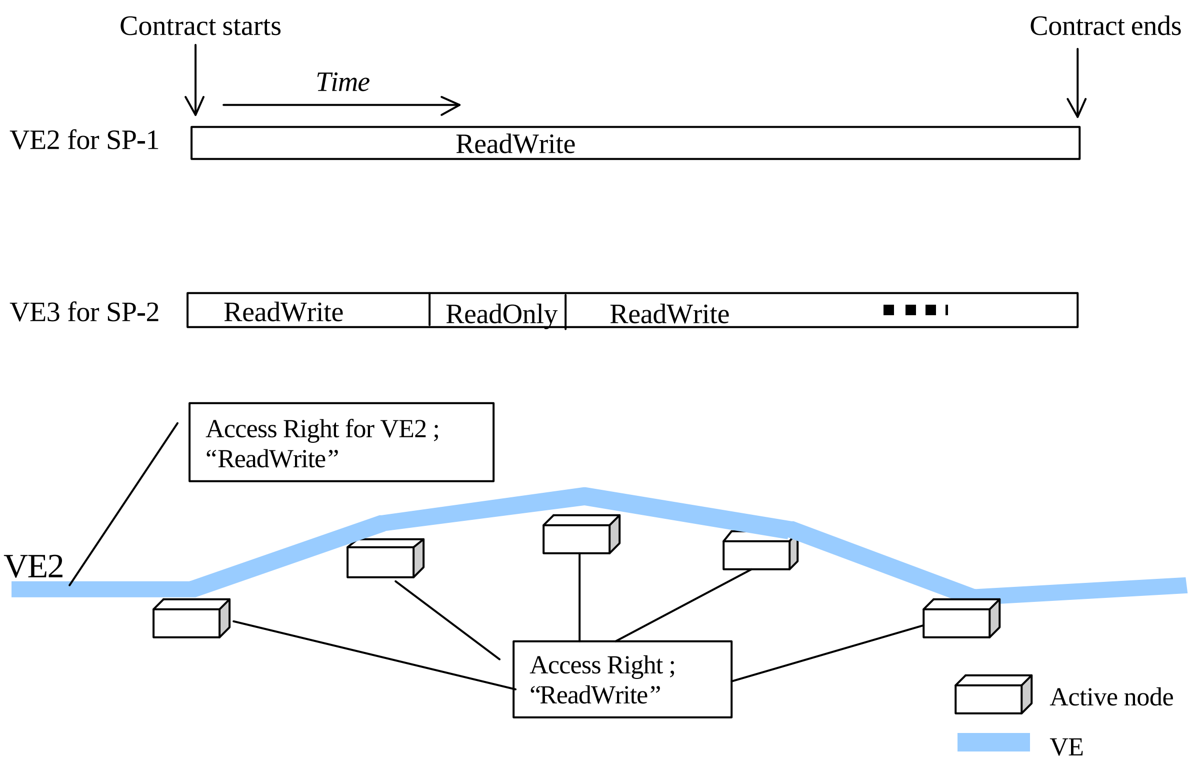


Figure 30: Period of Access Rights

(1) Weekly

The Customers can configure the access rights attributes by indicating days of a week.

	Mon	Tue	Wed	Thu	Fri	Sat	Sun
VE2	RW	RW	RW	RW	RW	RW	RW
VE3	RW	RW	RW	RW	RW	RO	RO

(2) Days

The Customers can configure the access rights attributes by indicating specific days. For instance, a customer may configure "VE1 attribute = Read/Write from 1st January to 31st March"

(3) Hours

The Customers may need to configure the access rights by indicating specific hours, however this function is not supported in this version.

4.2.6.2.2 Super-user

The right of a super-user allows a customer to disable the access rights that belong to other customers. Normally this right would be owned only by an ANSP.

4.2.6.3 Delegation PDP use-cases

The Main use cases of the Delegation PDP, Configure, Operate and Reconfigure are depicted in Figure 31.

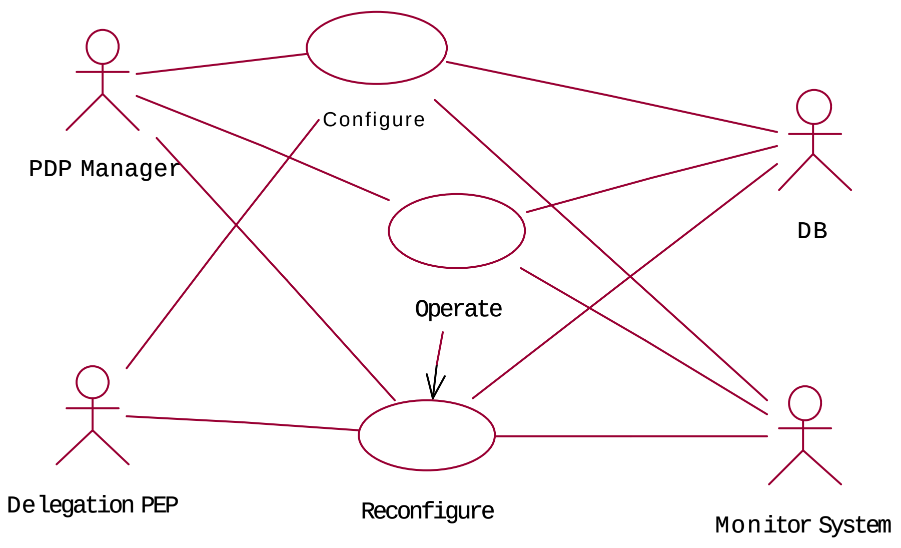


Figure 31: Main Use cases of the Delegation PDP

Configure when a new customer joins, this use case starts. Configuration of access rights in active nodes will be done.

Operate: When configuration is completed for a new customer during Configure steps, this use case starts working until the policy for this customer is expired.

Re-configure: when some configuration changes are needed, this use cases starts. For instance, a customer may need to reconfigure the access right attributes or if an error has occurred, a modification is necessary to avoid further problems.

We would mainly focus on the use cases "Configure" and "Operate" in the following

4.2.6.3.1 Configuration

When a new customer joins one ANSP, a configuration of the active nodes should be made. Configuration data are written in the policy rules, therefore the policy data need to be checked first, and then distributed to the proper active nodes through the Delegation PEP. The detailed use cases are depicted below:

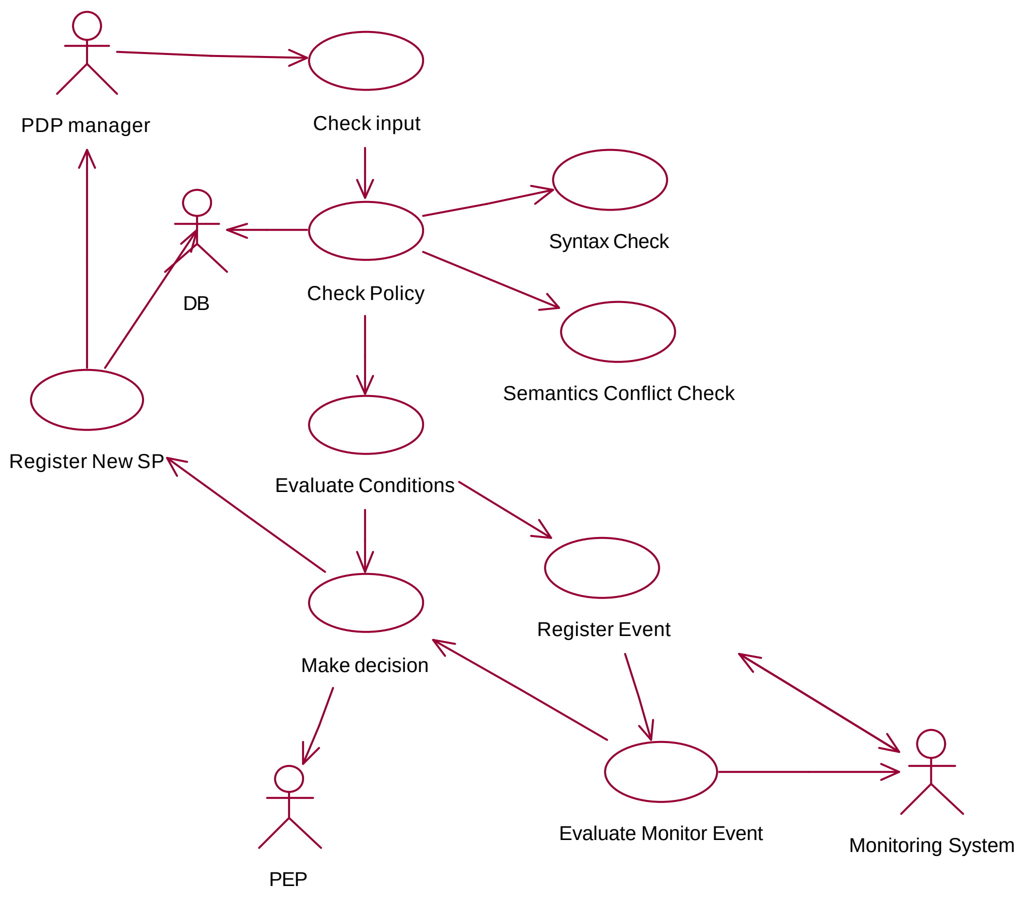


Figure 32: The detailed Use-cases of “Configure” in Delegation PDP

4.2.6.3.1.1 Use Case of "Check Input"

When the policy data is dispatched from the PDP manager, this use case starts. The Delegation PDP checks the type of policy data that is received from the PDP manager. If the policy is not a Delegation one, it notifies the PDP manager. The Delegation PDP stores the newly received policy data in a data table locally. This data table is located in the local memory assigned to the Delegation PDP when instantiated. At the same time, other information, which comes inside policy data from PDP Manager, is checked as well.

4.2.6.3.1.2 Use Case of "Check Policy"

The Delegation PDP checks the newly received policy data as far as the syntax and the semantics are concerned. The syntax check will verify that the delegation policy is syntactically correct, and the semantics check will verify that this policy is not going to conflict with other policies already installed. If there are no errors, the policy data is stored in the database. If there are errors, it notifies the PDP manager.

(1) Syntax Error

(2) Semantics Error

In the case that a definition of time condition may have invalid values or invalid variable combination, this is determined as a semantic error.

4.2.6.3.1.3 Use Case of "Register Monitor Event"

There may be the case when the monitoring system is required for the evaluation procedure to be concluded. Thus, an event is registered to the monitoring system. For example such an event may be the access rights status check on an active node. Also, the Delegation PDP may periodically check the access right status on the active nodes in order to verify that the SLA between the SP and the ANSP for instance, is not breached. The monitoring system is again used for this purpose.

(1) register information

The following information is sent to the Monitoring System and used to check the status of the access rights in the active nodes.

A) report interval

The PDP receives the report from the Monitoring System in this interval. Defined in milliseconds (e.g. every 5 ms a report is given from the monitoring system.)

(2) Error

If the registration in the Monitoring System fails, an error notification is returned to the delegation PDP (monitoring registration error).

4.2.6.3.1.4 Use Case of "Evaluate Conditions"

The delegation policy condition is evaluated with time period conditions such as dayofWeek. Then the “make decision” use case is executed. A delegation policy example is shown later.

4.2.6.3.1.5 Use Case of "Make Decision"

If the delegation policy condition is met, the “Make Decision” use case starts and the delegation PEP is asked to enforce the delegation policy. It then waits for notification of the results from the delegation PEP. A scheduler function is used to detect the exact time to enforce the policy in the case that the policy data include a time period. An example of this case is shown below:

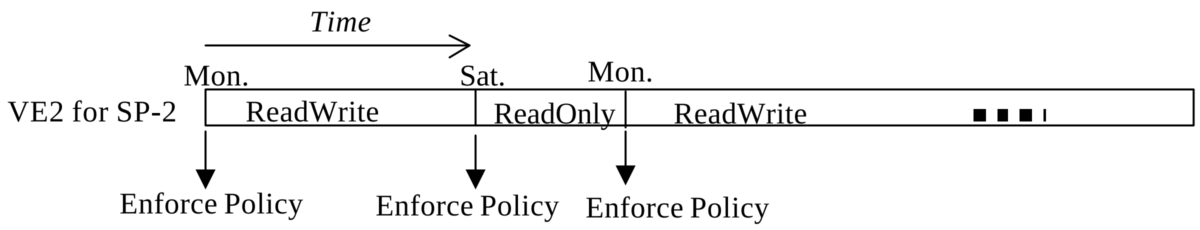


Figure 33: Policy Enforcement

(1) Information

To distribute the policy to the appropriate active nodes, the PDP passes the policy action data that contain the node list to the PEP.

4.2.6.3.1.6 Use Case of "Register/Unregister new SP"

If the delegation policy for a new SP is enforced successfully and no error is returned from the delegation PEP, a positive reply and a report is sent to the PDP manager.

Then the Delegation PDP stores the policy data of the new customer (SP) in the DB with a Java format. The "current_status" is set as stand by, which means that a policy is not being enforced at the moment.

If the operation is removal of a customer, the PDP removes the policy entry from the DB.

(1) Current_status of policy data. It indicates the status of the related policy data. This variable has the following values.

A) stand by

B) operating

C) halt

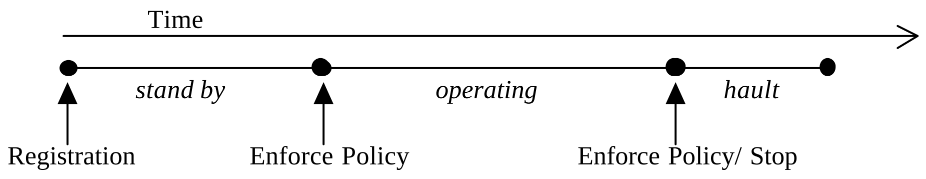


Figure 34: status of policy data delegation

(2) Error

If the registration has failed, an error is returned to the PDP Manager (registration error).

When the SP registration has finished, the operation is returned to the PDP Manager with a report. Then the next processes continue asynchronously.

4.2.6.3.2 Operate

After configuration, Delegation PDP deals with the access rights in active nodes, with checking status, reconfiguring and reporting to other components, e.g. PDP Manager, as necessary.

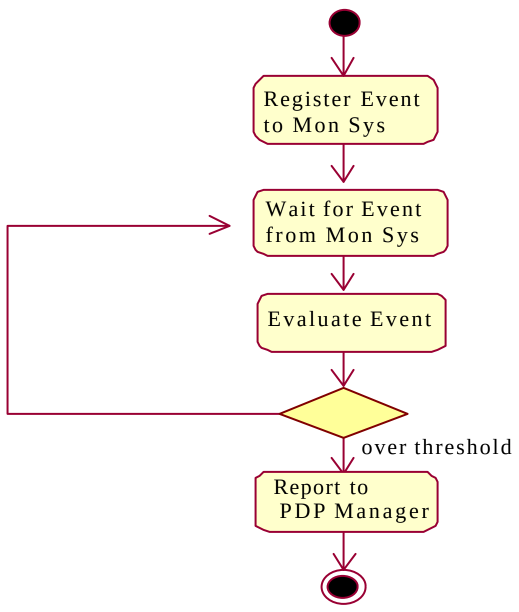


Figure 35: Active Diagram of "Operate"

4.2.6.3.2.1 Status Evaluation

Delegation PDP needs to check the overall status in order to honour the SLA of the customers and to discover potential malicious usage. The Monitoring System is required to achieve this with the following information. The Delegation PDP evaluates the information (event), which is given by the Monitoring System periodically as follows: the PDP accumulates a number of errors in a given interval, then compares the number with a predefined threshold.

(1) variables for evaluation

If the number of malicious accesses or the number of errors exceeds a threshold, an alarm report is prepared. For this purpose, the following value and information are used in the PDP.

(a) interval

(b) threshold

(c) error level definition

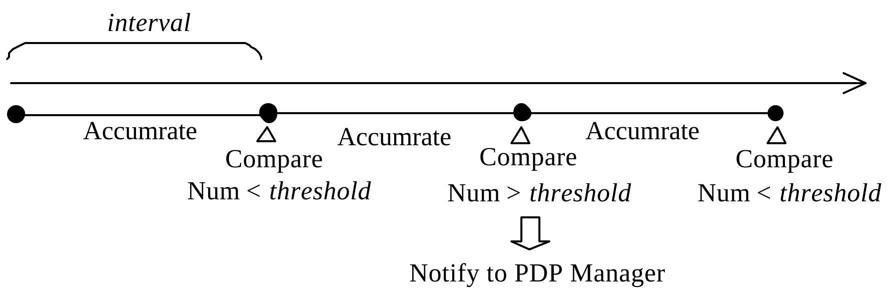


Figure 36: Evaluation

(2) Information from Monitoring System

Delegation PDP needs to obtain the number of invalid access with a VE id and active nodes ids from the Monitoring System.

4.2.6.3.3 Reconfigure

During normal operation, configuration may be needed. We can assume the following cases:

1) Some customers may reconfigure their policies for accommodating their services or consumers,
2) The ANSP may need to maintain the active nodes and thus change the customers configuration temporarily,
3) In the event of a malicious attack is detected in specific active nodes, the ANSP may compulsory configure these active nodes. "Reconfiguring" functions need to handle negative conditions in order to keep the access rights status stable.

The functions in "Configuration" section may support also the cases of 1) and 2).

4.2.6.3.4 Policy Example

Each policy data is controlled based on VE id individually. The example shows delegation policy for a certain VE which involves multiple active nodes (node id = 1, 2, 3, 4, 5). Let’s imagine here that the Customer (SP) may want to configure the access right differently for active nodes and frameworks. For instance this SP may do some maintenance on weekends and forbid access to active nodes.

IF (dayOfWeek= Monday, Tuesday, Wednesday, Thursday, Friday, Satuday, Sunday) AND

(frameWork= Demux) THEN (accessRight= ReadWrite for Demux)

IF (dayOfWeek= Monday, Tuesday, Wednesday, Thursday, Friday) AND

(frameWork= RCF) THEN (accessRight= ReadOnly for RCF)

IF (dayOfWeek= Monday-Sunday) AND

(frameWork= Demux) THEN (accessRight= ReadOnly for RCF)

The activity diagram of the Delegation PDP is presented below:

4.2.6.3.5 Delegation PDP class diagram

In this section the class diagram that corresponds to the Delegation PDP is presented.

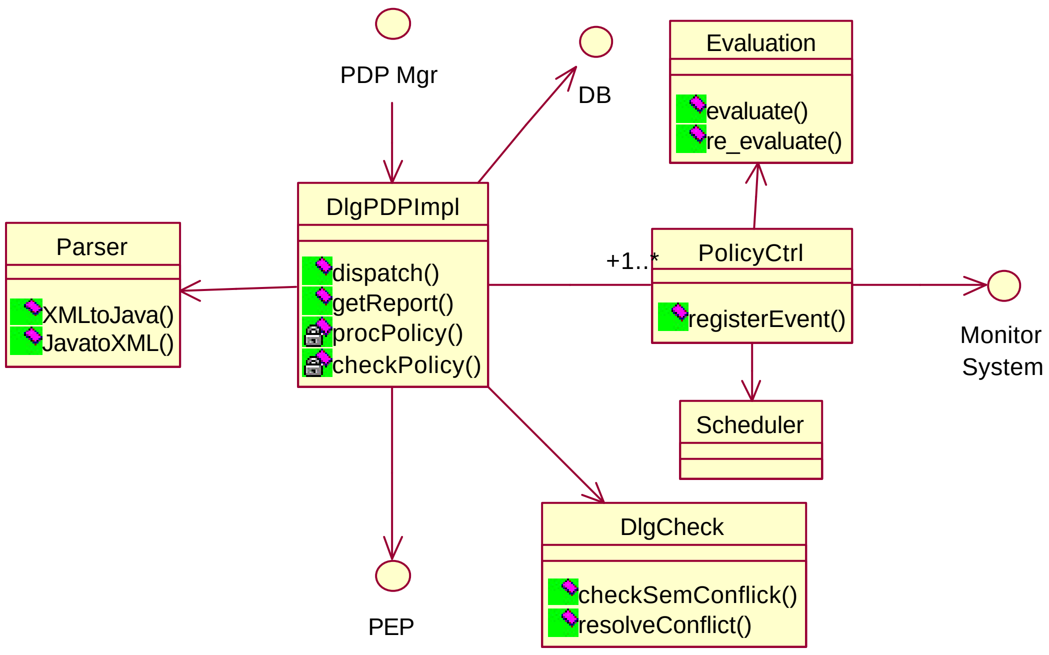


Figure 37: Class Diagram of the Delegation PDP

After the delegation policy is dispatched to the Delegation PDP, the latter checks it for syntactical errors via the checkPolicy() method of the DlgPDPImpl class. If an error is encountered, an exception is raised and sent to the PDPmanager through the PDPManagerInterface. If the policy is syntactically correct, it is checked for conflicts by the checkSemConflict() method of the DlgCheck class. If a conflict is detected it will be resolved by the resolveConflict() method. However, the conflict cannot be resolved an exception is raised and the PDP manager is notified, via the PDPManagerInterface.

If the PolicyCtrl needs to register an event to the monitoring system in order to check status, it does that by accessing the MonitoringSystemInterface. When the event arrives from the monitoring system, it is delivered to the PolicyCtrl class. When the conditions are met, the policy is delivered to the ActionInterpreter class to be enforced. The validity period of the Policy is set in the Scheduler class. And the PDPManager is notified about the policy deployment.

4.2.7 Element level Delegation PEP component

The functions of the delegation PEP at the element level are the following: It receives policies from the delegation PDP and translates them into commands being in a node readable format. Then it enforces the commands to the active node.

We can show the functionality of the element level delegation PEP with this sample high-level policy: A SP sends the following policy: “For the Edge nodes of my Virtual network that exist in the United Kingdom, I want medium security, and for the Core nodes I need high security”.

In the network level delegation PEP, the “high security” is resolved into RO (read only) and the “medium security” is resolved into RW (Read/write). Moreover, the Core and the Edge routers are now routers with specific IP addresses, due to the translation done by the Network level Delegation PEP.

At the element level, timing conditions are put into play. Specifically, the Delegation PDP decides when the time is right to enforce the delegation policy. When this time comes, it passes the action to the delegation PEP in order for the latter to enforce it.

In the case that we find ourselves inside the SP management domain, which means that the SP is responsible for deploying and enforcing the policies, the SP's ability to enforce the particular policy will be questioned. The EL Delegation PEP will create a restricted schema and store it in the Schema repository, where it will be collected by the Access Control Check (ACC) component. The Access Control Check component will check if the SP is able to enforce a particular policy by comparing this policy against the restricted schema. If the outcome is positive, the PDP manager will send the policy to the delegation PDP for further processing.

There is a special type of policy coming from the Delegation PDP that dignifies that a new user wants to instantiate management components. In that case the PEP will use the instantiateDom() method offered by the PDP manager interface. The parameters passed to the PDP manager should be the name of the entity that wishes to instantiate (e.g SP) and the components that the entity wishes to be instantiated. The PEP will receive the result of the instantiation procedure.

4.2.7.1 Use cases of the element level Delegation PEP

The following use cases diagram captures the above iterations:

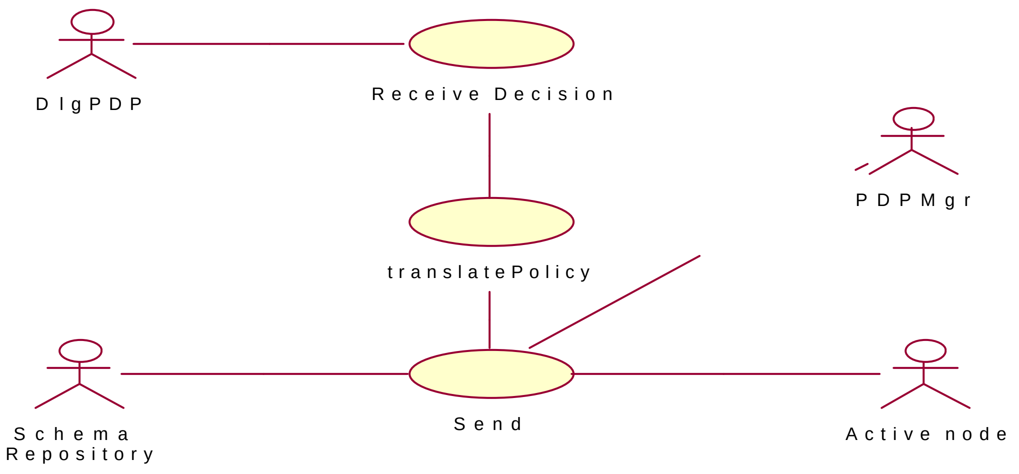


Figure 38: Use cases of EL Delegation PEP

4.2.7.2 Class diagram for the EL Delegation PEP

The class diagram for the delegation PEP at the element level is the following:

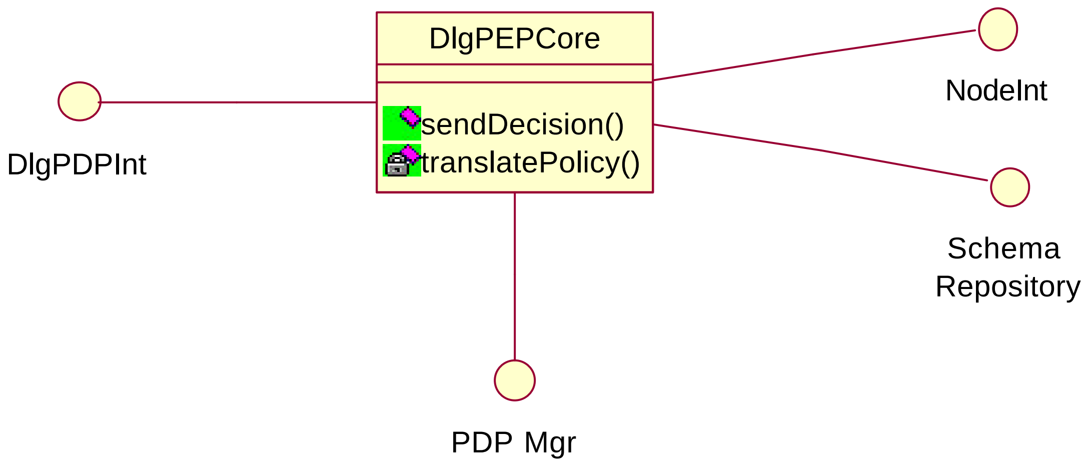


Figure 39: Class diagram of EL Delegation PEP

The sendDecision() method is used by the delegation PDP in order to pass the policy to the delegation PEP for enforcement.

The translatePolicy() method is used internally by the Delegation PEP in order to translate the policy action into a node understandable format.

4.2.8 Conflict Check Component

The functionality of this component, which appears in figure 2 both at the network and element levels, has already been introduced in section 2.2.1 and it will not be repeated here.

In section 2.2.1 two solutions were presented to fulfil the functionality needed in case we are not able to avoid the necessity of this component appropriately defining the FAIN policy information model. Nonetheless the actual election between one of the two solutions as well as, the design of such a solution has been left for the next deliverable document, when we will have evidences of whether this component is needed or not.

4.2.9 Monitoring Component

4.2.9.1 Monitoring Component Use Case Diagram

Figure 40 describes Monitoring component of FAIN management system use case diagram. This diagram summaries services (subscribe, add a probe, information retrieval and being informed of events) provided to authenticated users (subscriber).

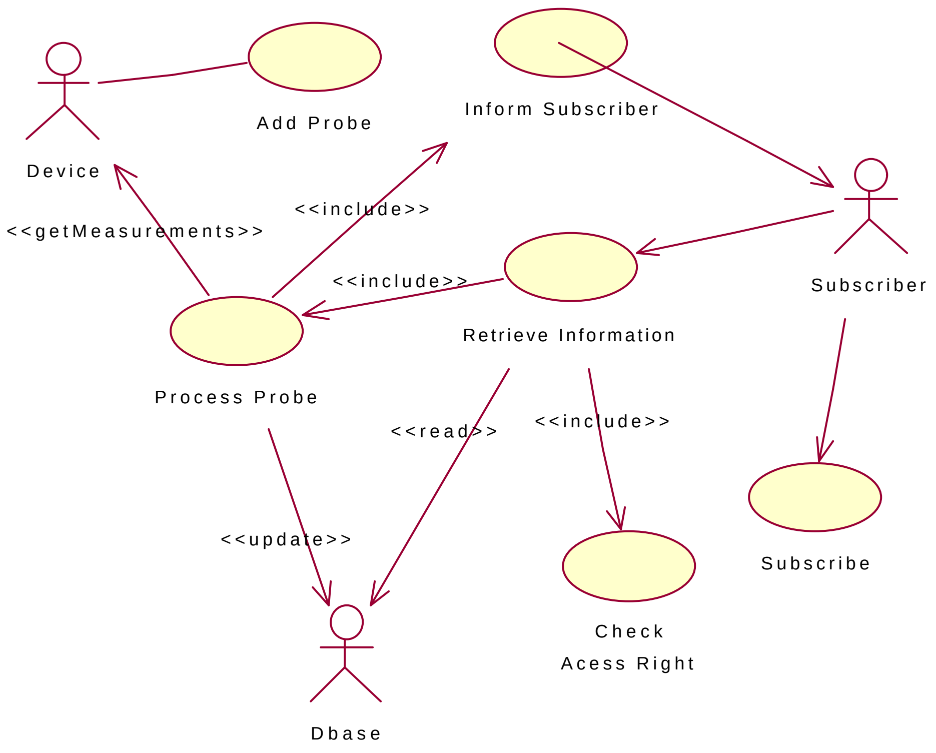


Figure 40: Monitoring system Use Case Diagram

Actors description

Actors taking part to the monitoring component use case diagram are any component, system or software unit that interacts with him/her. The following describes actors identified in the monitoring system use case diagram*:*

- The *Device*,

is an abstraction of either the FAIN Active Node, software unit or device that interact with the monitoring system. The FAIN Active Node represents infrastructure provider that offers shared resources via its Resources Control Function (RCF) to the Management system. Others devices controlled by the management system or software unit provided by some customers might furnish to the monitoring system any interfaces in order to access their internal probe

- *Dbase*,

is the Database Management System being use to store information for users of the monitoring system

- *Subscriber*,

Subscribers to the monitoring component are PDPs included the Resource Manager and network level Monitoring system that register events for which they are interested in after having subscribed to the monitoring system usage.

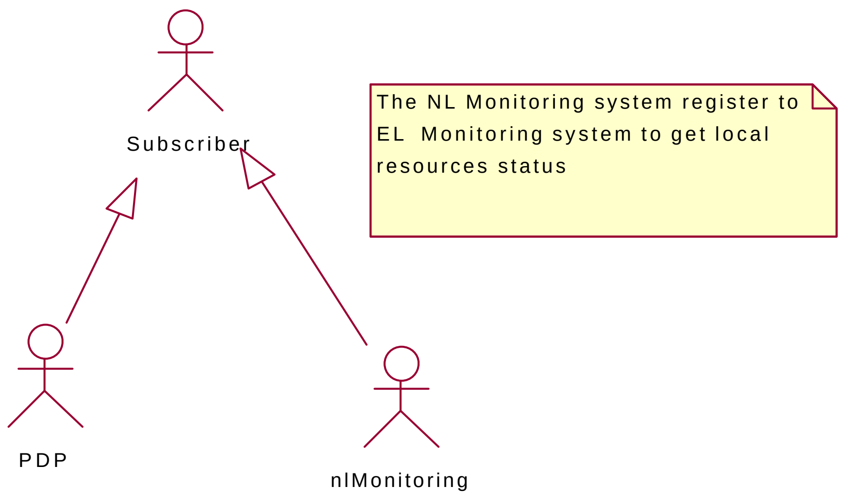

The NL Monitoring system register to EL Monitoring system to get local resources status
Subscriber
PDP
nlMonitoring

Figure 41: Subscribers to the monitoring component

Use Cases Description

Subscribe

This use case allows the monitoring system users to subscribe to the service and register events to be monitored. The monitoring component affects them an Id that identifies them for notifying them or used for future event registration

Retrieve information

This use case provides for users the ability to ask for any information or data registered in the monitoring system. It consists of a synchronous delivery of the event based on the pull model of a notification service.

Inform Subscriber

When a given threshold is triggered, this use case generates the adequate event to the concerned users asynchronously.

Process Probe

This use case is responsible for translating policies introduced by user into managed components' resource structure being measured

Check Access Right

This use case checks authenticated users.

Add Probe

This use case allows to introduce application specific probes into the monitoring system to extend its capability on behave of users.

4.2.9.2 Monitoring System Design

The monitoring system can be considered as a set of traditional metering blocks that are enhanced by the use of policy based control mechanisms and improved distribution channels. Although it is foreseeable that the active node will be accessed through P1520 interfaces, the design should be flexible enough to support other network interfaces as well.

As the figure below shows, the monitoring system will be distributed among the active nodes and the element management node. The P1520 objects and probably certain high performance monitoring PEPs will be hosted in the active node, whereas the main components and control logic will be placed in the management node.

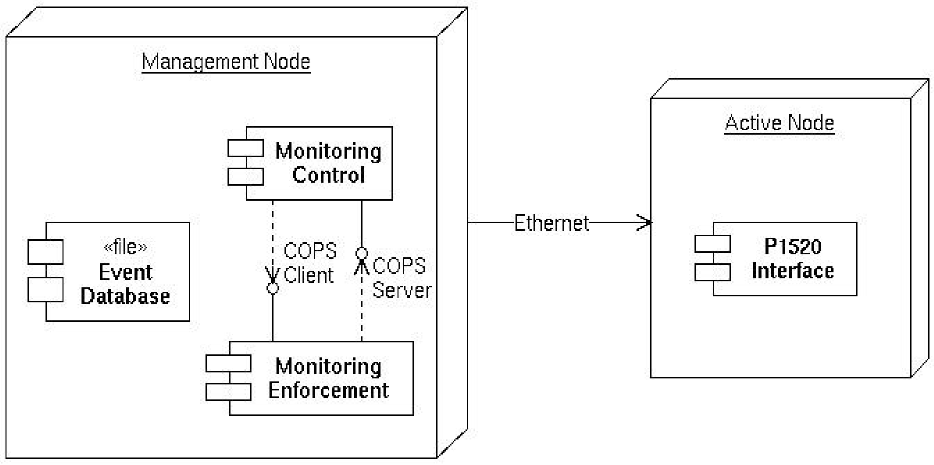


Figure 42. Monitoring system deployment diagram

The diagram corresponds to a three-tiered design in which a CORBA Component Model will be adopted as the base for the co-operation among the distributed objects. Also, the monitoring enforcement component and the monitoring control components will communicate through COPS interfaces. These two components are associated to the PEP and PDP respectively, although they contain additional objects and functionality that extend their basic behaviour. An event database completes the basic monitoring infrastructure in the management node.

A set of P1520 objects running in the active node will be accessed by the monitoring components. To model this interaction, a P1520 interface component has been included in the *active node*, which should define the interfaces that allow accessing monitoring information from an external module and presenting them to their clients.

Finally remark that the management node and the active node could be connected using a local area network in the case a management node is planned to be set up for each active node[5].

4.2.9.2.1 Monitoring System Packages

The monitoring system will contain a set of traditional metering blocks, an additional part for policy-based control and a distribution part required to support multiple PDPs. The different packages that have been identified reflect this structure. The diagram in Figure 43shows a draft of the system package decomposition. We have tried to minimize the number of packages by determining the expected functional areas and reducing the number of dependencies between packages. Thus, each package should group the classes with closer relationship.

[5]Other scenarios could be possible if a management node were to manage several distant active nodes.

Some of the packages, as in the case of the Protocol Handler, group other packages that are specialization for the different protocols. For example, it would include handlers for SNMP, COPS, LDAP, etc. The diagram does not include those packages embedded in other systems (such as the name service package), although they will be accessed as part of the normal operation of the system. The packages have been grouped into three functional layers: the control, distribution and acquisition layer.

The control layer is fulfilling the PDP role in the monitoring system while both distribution and acquisition layers realize the PEP role. The policy-based monitoring has been already presented in D3 at section 2.4.2.2.3 and 2.4.3.1.4.

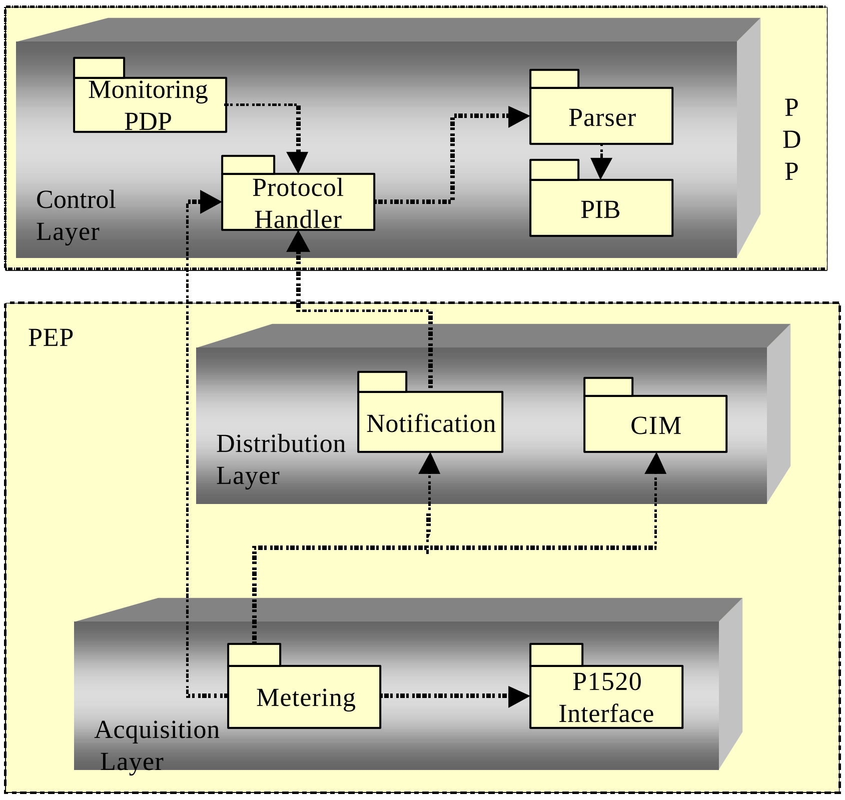


Figure 43: Monitoring System Packages

The CIM (Core Information Model)[6] package is intended to contain the data structures associated to the Policy. The notification package is in charge of distributing events to consumers.

Notification

The Notification Package is the main package of the acquisition layer. It provides extended capabilities around the basic CORBA event and notification service in order to deal with authenticated users and policies efficiently. It contains patterns that allow simultaneous treatment of several notifications in parallel.

[6]Internal Note: The policy classes could include XML serializing and deserializing methods. This seems to be an appropriate design decision since in XML, further meta-information is required to process the information correctly. Thus, since such meta-information cannot be obtained just by analysing the neither data type nor value of the data item, it would be difficult to proceed with the serialization in external classes. If possible, it would be interesting to implement any existing interface that allows us to link our specialized code with the existent code developed in general XML parsers. (To be done)

The extended notification service allows a simplification of the process of connecting to the event channel and the distribution of complete information on the consumer event subscriptions. This would lead to a more precise configuration of event suppliers that is currently achievable.

The CORBA notification channel has shown itself insufficient to transfer configuration information to the data acquisition layer. As a consequence, it has been necessary to extend the basic service to include the required new capabilities. In this way, while the notification service only announces the event type and domain name the consumers are interested in, managing its proper distribution, the extended notification service will also provide information on the filters being set up for each of the events. This information is broadcast to every interested entity.

This behaviour will facilitate the creation of entities being in charge of configuring the monitoring sensors and probes.

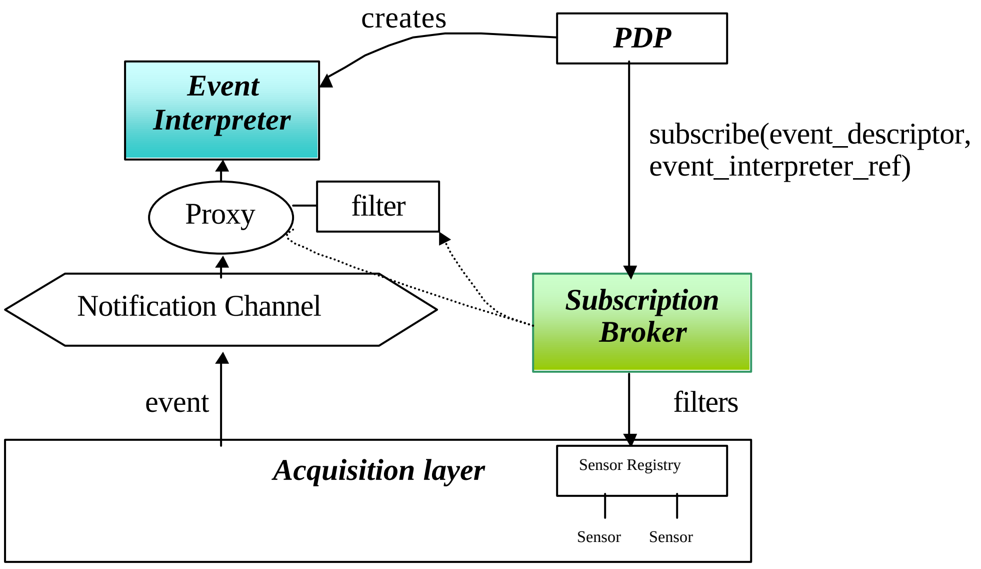


Figure 44 Extended Notified Service Architecture

The Subscription Broker

In order to enable the transference of event filter information from consumer to suppliers, a subscription broker has been defined. The subscription broker is responsible for connecting the consumers and suppliers to the event channel, controlling the way the subscription process is realized. As part of its mediation, the broker performs a delivery of filtering information to a series of interested entities. Such entities should provide the handlers (notifiers) required realizing additional configurations on the event sources, based on the fields contained in the event filters.

The subscription broker highly simplifies the process of connecting and disconnecting to/from the channel, hiding the peculiarities of the CORBA event channel. Bellow a part of the Broker interface description:

```
readonly attribute CosNotifyChannelAdmin::EventChannel channel;


ProxySupplier subscribe(     inout EventDescriptor eventDescriptor,
                             in CosNotifyComm::NotifyPublish subscriber)
                             raises (InvalidSubscriber);


void unsubscribe(     in EventDescriptor eventDescriptor,
                              in CosNotifyComm::NotifyPublish subscriber);

// Methods related to the event suppliers.
ProxyConsumer offer(        in EventDescriptor eventDescriptor,
                            in CosNotifyComm::NotifySubscribe notifier)
                            raises (InvalidNotifier);

void withdraw(in CosNotifyComm::NotifySubscribe notifier)
            raises (WithdrawalFailure);
};
```

The channel attribute contains a reference to the CORBA event channel being used by the subscription broker, so that it can be retrieved whenever an administration operation, not implemented by the broker, is required.

The subscribe method is invoked by the subscriber to ask for interest in events. The connection to the event channel is implicitly realised. The type of the subscriber should correspond to the type of events that are requested. That is, if the requested event is of type StructuredEvent, then the subscriber should be either a StructuredPushConsumer or a StructuredPullConsumer.

The unsubscribe method is called when the subscriber no longer needs the reception of events. The subscription broker releases the resources, which had been associated to such subscriber.

In a similar way, two symmetric methods are available for event suppliers: the offer method, which allows an event's source entity to offer a handler to manage, to configure the notifiers according to the filters defined by the event consumers.

The withdraw method withdraws an offer. Thus, the notifie r will no longer send events to the channel. The subscription broker proceeds to disconnect the event supplier from the event channel.

The information regarding the event is included in a common structure named EventDescriptor. The IDL definition of this structure is the following:

```
typedef sequence<CosNotifyFilter::Filter> FilterSeq;
typedef sequence<octet> encapsulated;

union DescriptorBody switch (short) {
      case 1: FilterSeq filters;
      case 2: NotifyConfigure interested_entity;
      case 3: CosNotifyFilter::FilterIDSeq filterIDs;
      default: encapsulated content;
};

struct EventDescriptor {
      CosNotification::EventType event_type;
```

```
        string event_name;
        DescriptorBody body;
};
```

Each event descriptor contains the event name and type, together with and additional field that may include either a filter sequence, in case a subscription is to be performed, or an event handler for the considered event. This handler would be the one to receive the filters defined for each event type. The interface is open to new possibilities by the use of an encapsulated field.

If it is considered that the same subscriber may be interested in the reception of several event types, it would be necessary to be able to identify the concrete event we are interested in dropping from the interest list. It is therefore required to be able to name the filters we want to remove from the channel, so that the subscription broker can control them. This is the reason why a sequence of filter identifiers has been included as one of the possible fields being contained in the descriptor body.

This sequence will be returned by the offer function in the event descriptor (defined as an inout parameter), and might be used during the subscription removal.

The Extended Notify Server

The extended notify server is in charge of obtaining the reference to the ORB and the notification channel, connect the subscription broker to the ORB and transfer the obtained references to it. It will also register the broker in the name service, so that the event suppliers and consumers may access it.

The NotifyConfigure Handlers

The event handlers should offer the following IDL external interface:

```
interface NotifyConfigure {
void configure(in FilterSeq addedFilters);
void reconfigure(in FilterSeq removedFilters);
};
```

This interface should be implemented by the entities interested in receiving filtering information for the events they could generate. This information may be useful to appropriately configure the different objects involved in event generation. The subscription broker will use call-back requests to broadcast the information requested by the registered handlers.

The reconfigure method has been added to provide a way to inform to the configurators about the removal of filters, so that appropriate actions can be realized on the monitoring system.

The subscription broker is able to detect the removal of event handler and update its information tables without explicitly informing it.

Metering Package

The Metering package includes classes that implement the measurement algorithms and the specific controllers required managing monitoring devices. The overall diagram in Figure 45 depicts the main classes and their relationships of this package.

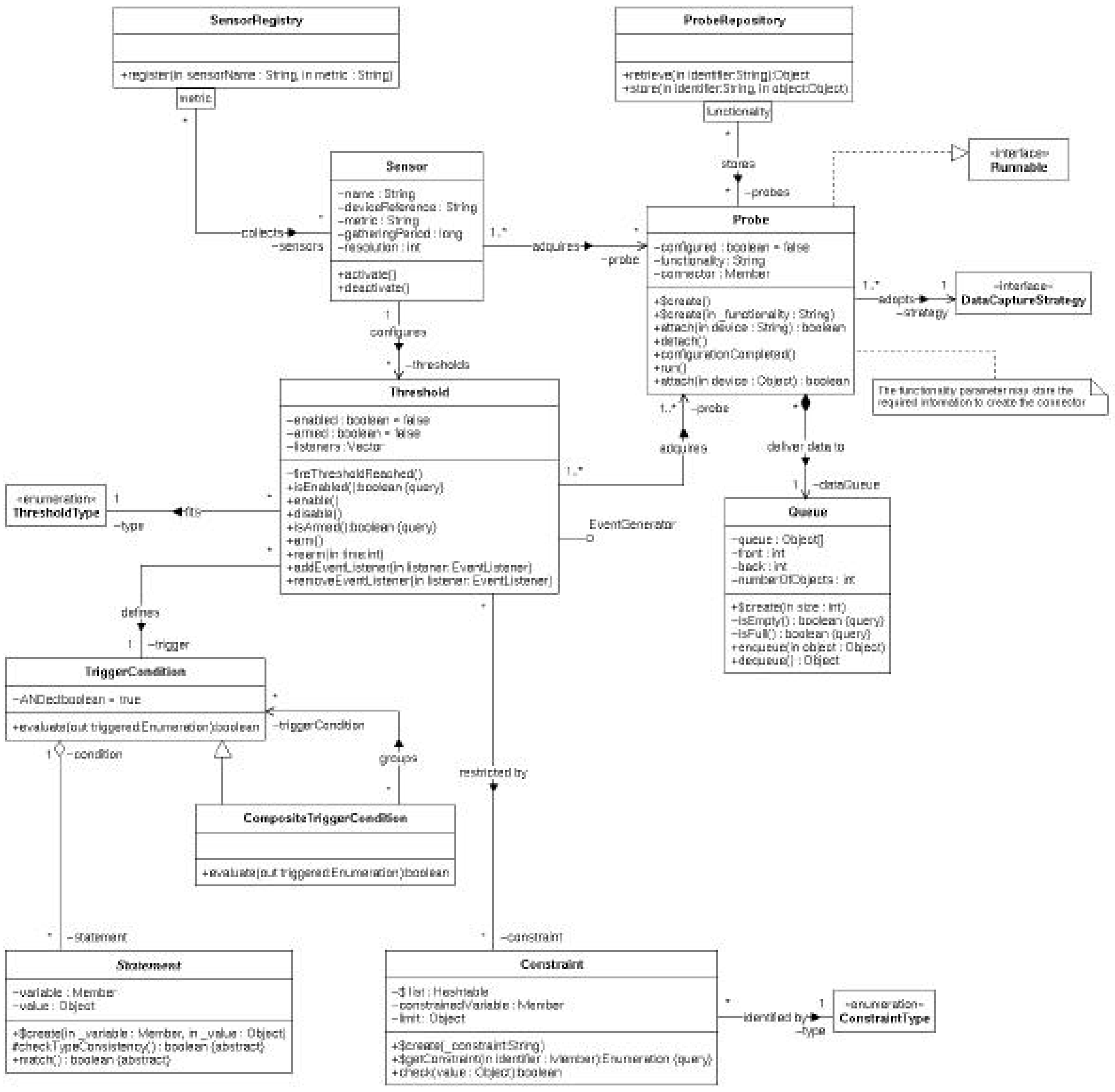

SensorRegistry
+register(in sensorName : String, in metric : String)
metric
ProbeRepository
+retrieve(in identifier:String):Object
+store(in identifier:String, in object:Object)
functionality
stores
-probes
«interface»
Runnable
Sensor
-name : String
-deviceReference : String
-metric : String
-gatheringPeriod : long
-resolution : int
+activate()
+deactivate()
collects
-sensors
1..*
acquires
-probe
Probe
-configured : boolean = false
-functionality : String
-connector : Member
+$create()
+$create(in _functionality : String)
+attach(in device : String) : boolean
+detach()
+configurationCompleted()
+run()
+attach(in device : Object) : boolean
1..*
adopts
-strategy
1
«interface»
DataCaptureStrategy
The functionality parameter may store the required information to create the connector
configures
-thresholds
Threshold
-enabled : boolean = false
-armed : boolean = false
-listeners : Vector
-fireThresholdReached()
+isEnabled():boolean {query}
+enable()
+disable()
+isArmed():boolean {query}
+arm()
+rearm(in time:int)
+addEventListener(in listener: EventListener)
+removeEventListener(in listener: EventListener)
1..*
-probe
acquires
deliver data to
-dataQueue
«enumeration»
ThresholdType
fits
-type
EventGenerator
Queue
-queue : Object[]
-front : int
-back : int
-numberOfObjects : int
+$create(in size : int)
-isEmpty() : boolean {query}
-isFull() : boolean {query}
+enqueue(in object : Object)
+dequeue() : Object
defines
-trigger
TriggerCondition
-ANDed:boolean = true
+evaluate(out triggered:Enumeration):boolean
-triggerCondition
groups
-condition
restricted by
CompositeTriggerCondition
+evaluate(out triggered:Enumeration):boolean
-statement
-constraint
Statement
-variable : Member
-value : Object
+$create(in _variable : Member, in _value : Object)
#checkTypeConsistency() : boolean {abstract}
+match() : boolean {abstract}
Constraint
-$ list : Hashtable
-constrainedVariable : Member
-limit : Object
+$create(_constraint:String)
+$getConstraint(in identifier : Member):Enumeration {query}
+check(value : Object):boolean
identified by
-type
«enumeration»
ConstraintType

Figure 45 Metering class diagram

Following the classes allowing the operation on the metering blocks are described. (*Implementation Note:* As an important non-functional requirement, the event handling mechanisms should not block the event generators).

Class Threshold

This class is in charge of the Threshold analysis and the delivery of threshold events to the event-dispatching queue. It is also responsible for monitoring whether a threshold has been reached and obtaining other statistical data. Two classes of threshold may be differentiated: those that check whether a certain value is surpassed and those that check whether a value goes down a certain limit.

The class `Threshold` should be able to analyse the amount of time a value is over a defined limit, since in some situations this is an important data. For this reason, each Threshold class, which *fits* a certain ThresholdType, *defines* a TriggerCondition whose evaluation will determine whether to raise a threshold event or not (note that it is possible to define a complex CompositeTriggerCondition). A set of constraints *restricts* the definition of trigger conditions so that they are properly formed. The constraints might be obtained from the device being monitored. These constraints are thus checked when establishing the trigger conditions for the threshold.

At the same time, an appropriate set of probes is required when the threshold configures the statements and trigger conditions. The probes will be the only means of accessing the devices.

Class Statement

A statement is formed by an expression, a value and a match operation. Subclasses of the Statement class should implement the match method according to the expression and value types and semantics.

Class Sensor

The Sensor is the central class of the metering package, being in charge of creating and coordinating the rest of classes. This class should obtain the available information on the device and specify the corresponding Thresholds and Probes as necessary to fulfil the measures.

The Sensor event handlers are able to locate the source of threshold events by inspecting the ThresholdEvent class. Since each sensor stores the relationship between a device and the Thresholds that have been defined for it, it is always possible to obtain the reference to the device on which an action is demanded.

The metering processes can be activated and deactivated by invoking the operations `activate` and `deactivate` respectively. Below some descriptions of the class main attribute:

- `deviceReference`: stringfied object reference to the device representative class. This reference will be used to connect the necessary probes to the device or resource being monitored.
- `metric`: contains a description of the metric being used by the Sensor.
- `gatheringPeriod`: specifies the time interval during which data is processed by the sensor. When the interval has finished, the resulting high level metric is delivered by the sensor. The special value 0 makes the sensor to deliver each collected event. However, be aware that this may cause an increase in the overhead.
- `resolution`: the resolution provides information about the amount of data that the meter should collect. The higher the resolution the more the resources that are consumed.

Class Probe

A `Probe` implements engines being in charge of checking the data of interest from the devices. Either metering blocks or threshold classes would use probes to obtain monitoring information. The logic associated to the data capture is confined within strategies. Therefore, the Probe class should only offer the strategy control and adjustment, whereas at the same time guarantees the data delivery. Several strategies could be used to obtain the data, such as polling

Once the probe has been retrieved from the repository, the `attach` operation should be invoked in order to connect the probe to a specific device. The device reference may be passed to the method as an interoperable object reference in string format. The `configurationCompleted` method should be called when the configuration of the Probe has been completed, i.e. it has been attached to a device and the strategy to access such device has been defined. Finally, the `detach` method closes the connection with the device and releases the associated resources. The Probe finalizer should check whether the Probe has been detached or not, taking the appropriate measures to assure that the system recovers the resources assigned to the Probe.

-Care must be taken to avoid a too low polling interval since it could become a bottleneck or injecting traffic in the network to measure the packets delay, errors, etc. Thread based engines might suspend until they receive appropriate exceptions or CORBA messages. Main attribute of probe are the following:

- `Configured`: informs whether the probe has been configured or not.
- `Functionality`: contains a description of the probe functionality. The description will be structured in several fields separated by blank spaces. The first field represents the strategy to be used by the Probe ("sample", "poll", etc). The second field explains the type of Member that should be accessed (a "field" or a "method"). The third field provides the name of the field or method to be accessed. This way, an example of functionality description would be:

  "poll method getNumberOfLostPackets"

  During the configuration process, the probe ensures a coherent state between the stated functionality, the connector and the strategy being used. Anytime a configuration change occurs, there is a convenient functionality update.
- `connector`: This parameter should contain the set of references that allows dynamically calling a method from another class. In that sense it becomes the "connector" of the Probe. It provides the necessary support for binding to anonymous classes, based on the introspection capabilities offered by the Java language. To avoid loosing generality, the connector has been defined Member, so that either Field or Method values can be accessed.

Class ProbeRepository

A `ProbeRepository` stores the probes available for a given device. The probe functionality becomes the key to access the repository. A probe repository might exist per device.

The following diagram depicts the structure of the strategy related classes.

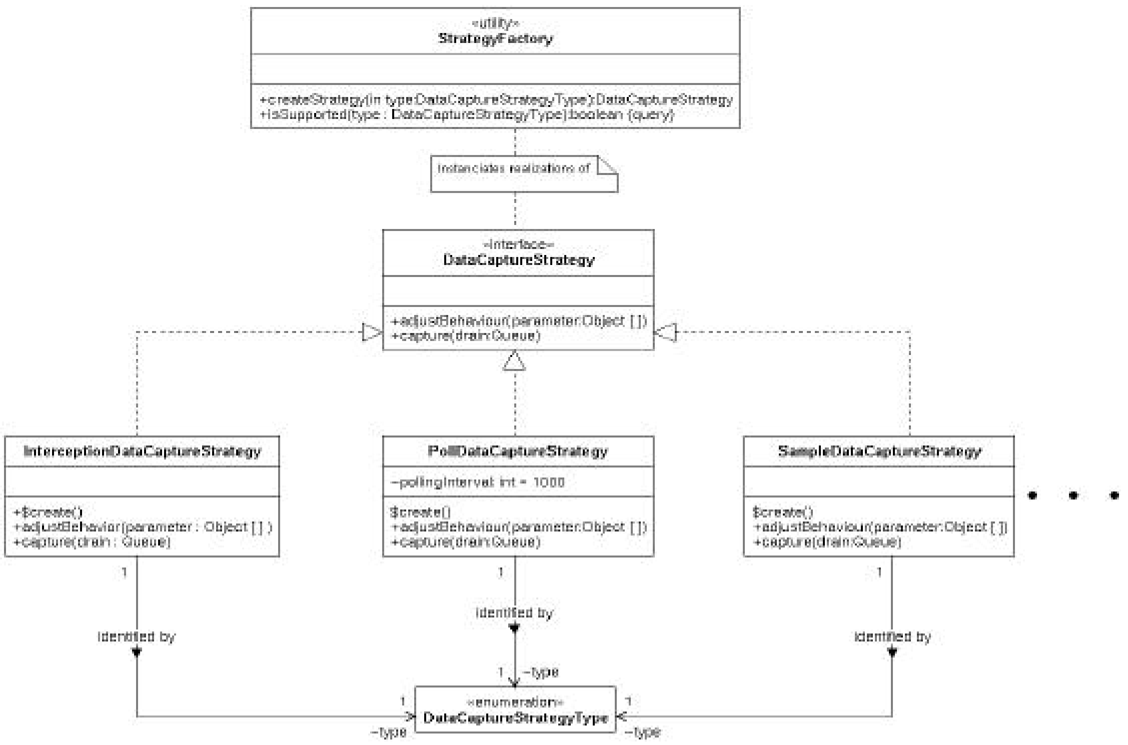

«utility»
StrategyFactory
+createStrategy(in type:DataCaptureStrategyType):DataCaptureStrategy
+isSupported(type : DataCaptureStrategyType):boolean {query}
Instanciates realizations of
«interface»
DataCaptureStrategy
+adjustBehaviour(parameter:Object [])
+capture(drain:Queue)
InterceptionDataCaptureStrategy
+$create()
+adjustBehavior(parameter : Object [])
+capture(drain : Queue)
PollDataCaptureStrategy
-pollingInterval: int = 1000
$create()
+adjustBehaviour(parameter:Object [])
+capture(drain:Queue)
SampleDataCaptureStrategy
$create()
+adjustBehaviour(parameter:Object [])
+capture(drain:Queue)
1
identified by
1
-type
«enumeration»
DataCaptureStrategyType

Figure 46: Strategy pattern class diagram

Class StrategyFactory

This class is responsible for creating the data capture strategies depending on their algorithm definition. It follows the strategy pattern.

The `createStrategy` method will be invoked in order to create a new DataCaptureStrategy that will use the algorithm corresponding to the type parameter. It may throw an UnsupportedStrategyException if the strategy is unknown or unsupported. The `isSupported` operation allows checking in advance whether the factory is able to create the requested DataCaptureStrategy.

Class DataCaptureStrategy

The `DataCaptureStrategy` class allows specifying several algorithms that can be used when accessing information on a given device. This way it is possible to alter the algorithm without having to change the rest of the system. The method `adjustBehavior` allows the specification of parameters that are required to adjust the algorithm. For example, in the case of a polling strategy, the polling interval could be defined by realising the following operation:

strategy.adjustBehaviour(new Integer(1000));

The concrete semantic of this method depends on the type of algorithm. This method only provides a convenient way to pass parameters to algorithm and adjust them on runtime.

Class DataCaptureStrategyType

Defines the type of the data capture strategy. The constructor is made private to avoid trying to create illegal types. Different data capture strategy types could include polling or waiting for an asynchronous event. So far, the following strategy types have been defined:

Class InterceptionDataCaptureStrategy

This Strategy will be used when an interceptor is required in order to capture data related to a request performed between two components. One of the components may be a middleware platform, such as a CORBA ORB and is specially useful in service activity monitoring.

Class PollDataCaptureStrategy

This Strategy is intended to be used when it is necessary to periodically poll a device or resource for data. The data will then be automatically sent to the output queue. Attribute `PollingInterval` defines the polling interval in milliseconds. The default polling interval will be 1 second.

Class SampleDataCaptureStrategy

This strategy is intended to be used when taking a single data sample is enough to perform a value capture. In this case it might be possible to define the exact time when the sample is required. This information could be passed as part of the algorithm parameter list.

ProtocolHandler Package

This package contains the protocol handlers for the protocols required for the system operation, such as COPS, SNMP or LDAP. Other packages, such as the notification or the metering ones, depend on classes contained in `ProtocolHandler`. However, a good design should minimize the possibilities of affecting external packages by defining interfaces that comply with the protocol specifications. Figure 8 displays the main classes that are required for COPS operation. Although the concrete package design would appear as part of other sections, the diagram is intended to highlight the existence of delegate classes that connect the client and server entities with external package modules implementing part of the behaviour.

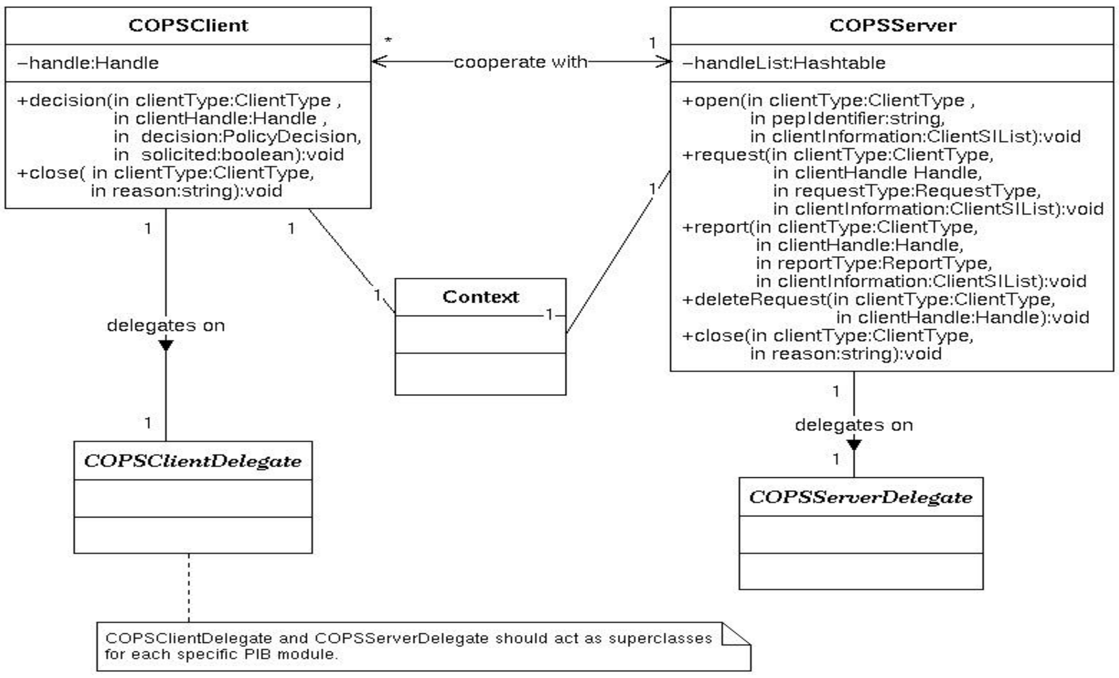

COPSClient
–handle:Handle
+decision(in clientType:ClientType , in clientHandle:Handle , in decision:PolicyDecision, in solicited:boolean):void
+close(in clientType:ClientType, in reason:string):void
COPSServer
–handleList:Hashtable
+open(in clientType:ClientType , in pepIdentifier:string, in clientInformation:ClientSIList):void
+request(in clientType:ClientType, in clientHandle Handle, in requestType:RequestType, in clientInformation:ClientSIList):void
+report(in clientType:ClientType, in clientHandle:Handle, in reportType:ReportType, in clientInformation:ClientSIList):void
+deleteRequest(in clientType:ClientType, in clientHandle:Handle):void
+close(in clientType:ClientType, in reason:string):void
cooperate with
Context
delegates on
delegates on
COPSClientDelegate
COPSServerDelegate
COPSClientDelegate and COPSServerDelegate should act as superclasses for each specific PIB module.

Figure 47: COPS protocol handler class diagram

Class COPSClient

The `COPSClient` contains the algorithms and the state machine of a COPS client. However, since a part of the state machine depends on the PIB, such part of the behaviour has been delegated to certain classes in the PIB package.

Class COPSServer

The `COPSServer` class contains the algorithms and state machine corresponding to a COPS server. Since part of its behaviour depends on the PIB modules it is associated with, the `COPSServer` delegates such part on the `COPSServerDelegate` defined in the PIB packages.

Class Context

The `Context` classes contain the information that associate a COPS client/server with its corresponding delegate in the PIB module.

P1520 Interface Package

This package will contain the P1520 wrappers required to communicate with the monitoring entities hosted in the active node. The P1520 interfaces will allow accessing monitoring information from external sources in a standard way.

5 R15 PBANM (NL-MS) DESIGN

5.1 Network Level Management System use cases

PBANM is using the generic use cases diagram of the NMS described in the previous chapter. However two additional components supporting decision-making with regards to resources control and the inter-domain communication have been introduced: Resource Manager and the inter-domain Manager components. The roles of both components are capital to deal with network wide concerns. The resource manager component will provide PDP with the best route domain wide according to resources status. The inter-domain component will be in charge of furnishing all mechanisms allowing communication with other domains. The inter-domain issue encompasses a lot of views that should be carefully tackle in order to avoid unnecessary complicated vision. In the PBNM system, the inter-domain component will be in charge of conveying and managing requests over FAIN domains. Figure 48 illustrates PBANM's components and how Inter-domain and Resource Manager are integrated in the whole sub-system.

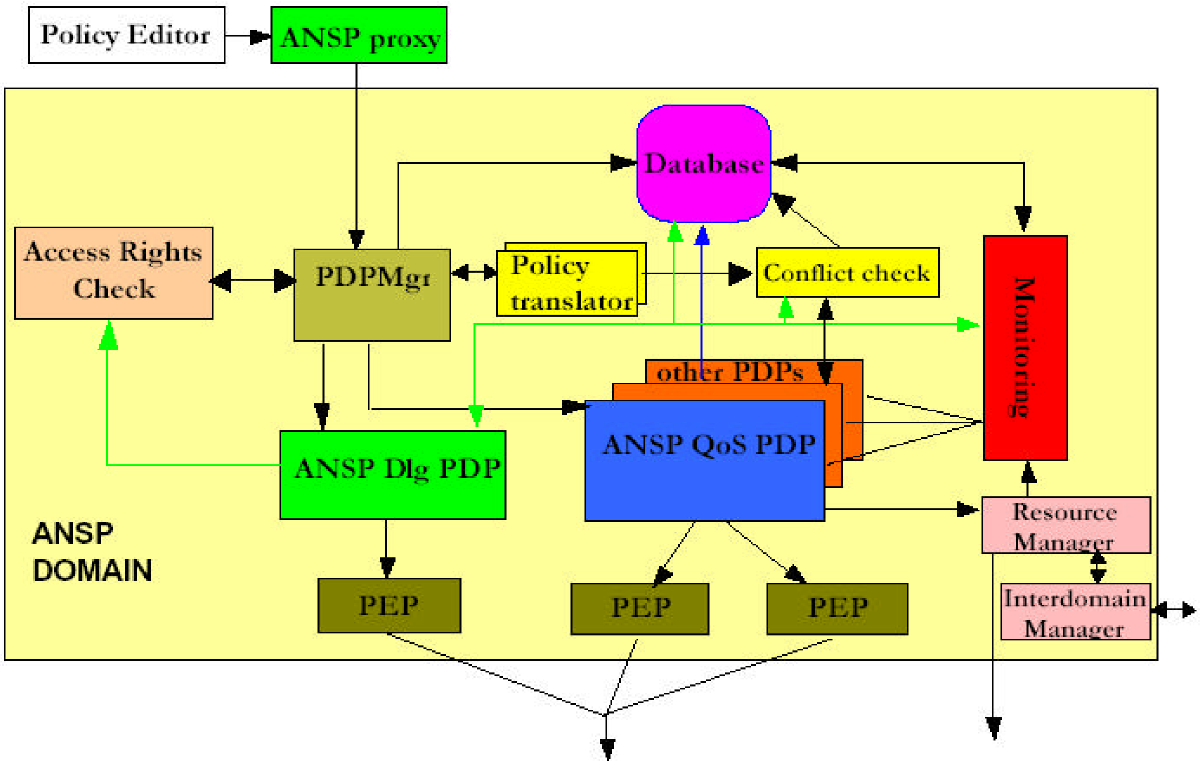


Figure 48: PBANM's Components

We are not going to deal with both components in this document since they have been postponed to future works (Y3) when prioritising activities.

5.2 components description

According to the 2-tiers Architecture, we will avoid redundancy. Only specific components (resource Manager, Inter-domain management) and whatever is specific to the network level in others components are described. However according to our priorities and time constraint the two most specific component (resource manager and inter-domain management components) of this level have been shifted to Y3 where they will be deeply investigated.

5.2.1 ANSP Proxy Component in Network Level

Policies coming from the ANSP or the SP policy editor are forwarded to the ANSP proxy. The ANSP proxy forwards the policies to the appropriate domain instance (either ANSP or SP domain). The ANSP proxy functionality is to authenticate the requests and then to find the instances to which the policies must be forwarded. The actual policies are processed by the proper instances and not by the ANSP-proxy.

In other words the ANSP proxy makes the management architecture more robust from the security point of view. After the ANSP proxy has performed the necessary security checking, and the SP is allowed to enter the management framework, the SP can perform SP-specific checking within the SP domain. This is similar to a proxy of a web site where a proxy checks if you are allowed to send traffic and then the web server software does additional security checks.

Although a proxy may be located to a different physical station, it could also be the case that is co-located with the ANSP or SP instances.

This component in network level works with interactions of Policy Editor and PDP managers, which can be multiple instantiation for an ANSP and several SPs. ANSP Proxy needs to dispatch policy data towards to an appropriate PDP manager.

5.2.1.1 Use cases

Firstly the ANSP Proxy checks the credentials of the incoming policy data, then dispatches it to one of the PDP managers. Also the ANSP proxy component will verify the reports sent by the PDP managers and will notify the results to the customers. This notification could be done via a GUI used by the Policy Editor.

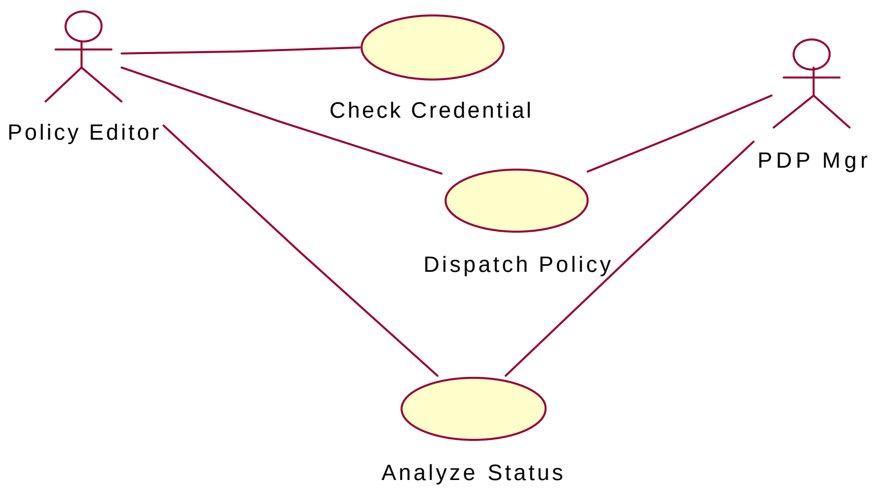


Figure 49: Main Use cases of the ANSP Proxy in Network Level

5.2.1.2 Class Diagram

The ANSProxyImpl class provides two methods. The sendPolicy() is used by the Policy Editor to install the policy data and the setReport() is used by the PDP Manager in order to report the status of the policy deployment.

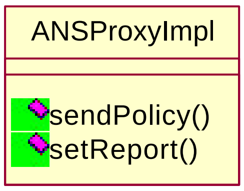


Figure 50: Class Diagram of the ANSP Proxy in Network Level

5.2.2 PDPMgr Component

The functionality of the PDP component at the network level is exactly the same as the one described previously for the element level in section **4.2.2**.

5.2.3 QoSPDP Component

At the network level the PDP component is also, as in the element level, one of the key components of our policy-based management architecture. There are not many differences between the PDP component designs at different levels, as one would rapidly notice comparing the element level and network level use case diagrams of this component. The network level use case diagram is given in Figure 51 below.

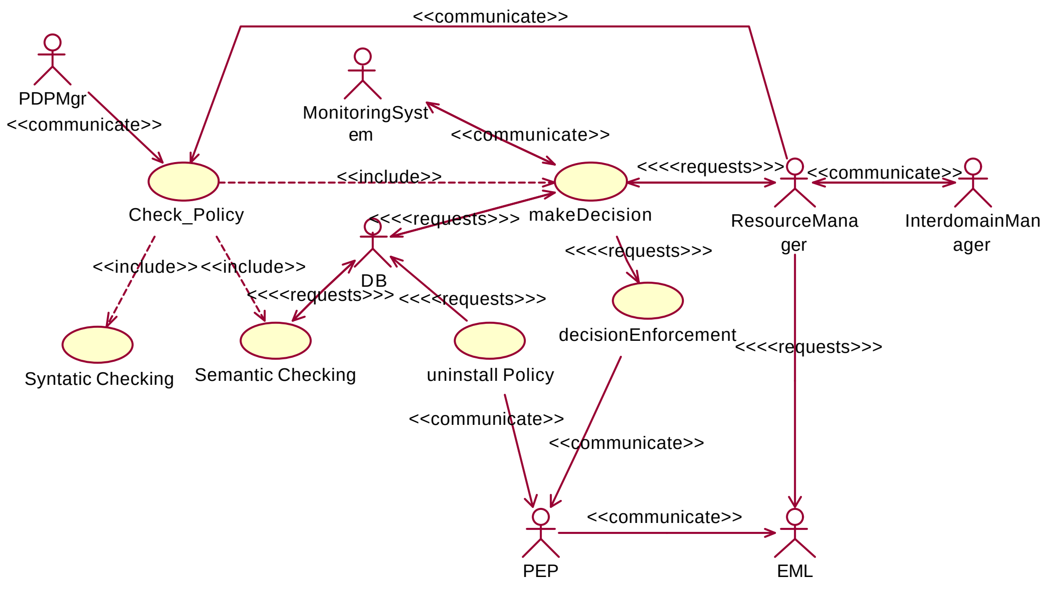


Figure 51 – NL-PDP Component use cases Diagram

In this section we are going to describe the network level QoS PDP functionalities and design. Since this description does not vary significantly from that given in a previous chapter for the element level QoS PDP, we are just going to point out here the main differences instead of repeating the whole description. The main differences in the functionalities with which the PDP at the network level deals with in relation with the element level are:

- The result of the enforcement of the decisions taken by the PDP component at the network level are element level policies and not commands to the active nodes interfaces as in the element level case.

· There is no support for signalling functionality at the network level. This is due to the fact that signalling requests for decisions are processed only at the element management stations. Nevertheless a similar type of functionality can be realised using the event notifications to the network level.

· The most important difference between both levels is the support for the "makeDecission" functionality. At the network level most policies request network-wide resources. Before deciding whether these policies should be enforced or not, the makeDecision functionality has to know the availability and location of requested resources in the network. To obtain this information the makeDecision functionality will contact the resource manager component. With the help of its peer at the element level, the monitoring system and the interdomain manager, the network level RM gathers information needed by makeDecision functionality.

Another difference in the makeDecision functionality at the network level enforcement is that, we do not have the same situation of having the enforcement points replicated at different active nodes virtual environments. So that we do not need to demultiplex the decisions to the appropriate enforcement point because there is only one suitable and it is well known.

The class diagram at the network level shown in Figure 52 reflects the network level PDP design. Again as with the use cases we are not going to provide the whole description of the QoS PDP class diagram, but just the main differences of this component design against the design of the element level QoS PDP described before:

· Since no signalling support functionality is realised, the class that developed this functionality disappears from the design (i.e. the SignallingComp class).

· Also, the command_demux class is removed from the PDP design at the network level because, as described above, there is no need of demultiplexing decisions to the correct enforcement point since there is only one suitable enforcement point.

· The Condition_interpreter class adds the necessary functionality to interact with the resource manager component for coping with policies requesting network-wide resources.

· Finally, the pdpQoSOpsImpl is also briefly modified since its request function is changed for supporting requests of resources from other domains coming from the interdomain manager and through the resource manager. At the element level this function was used to support signalling requests for decisions.

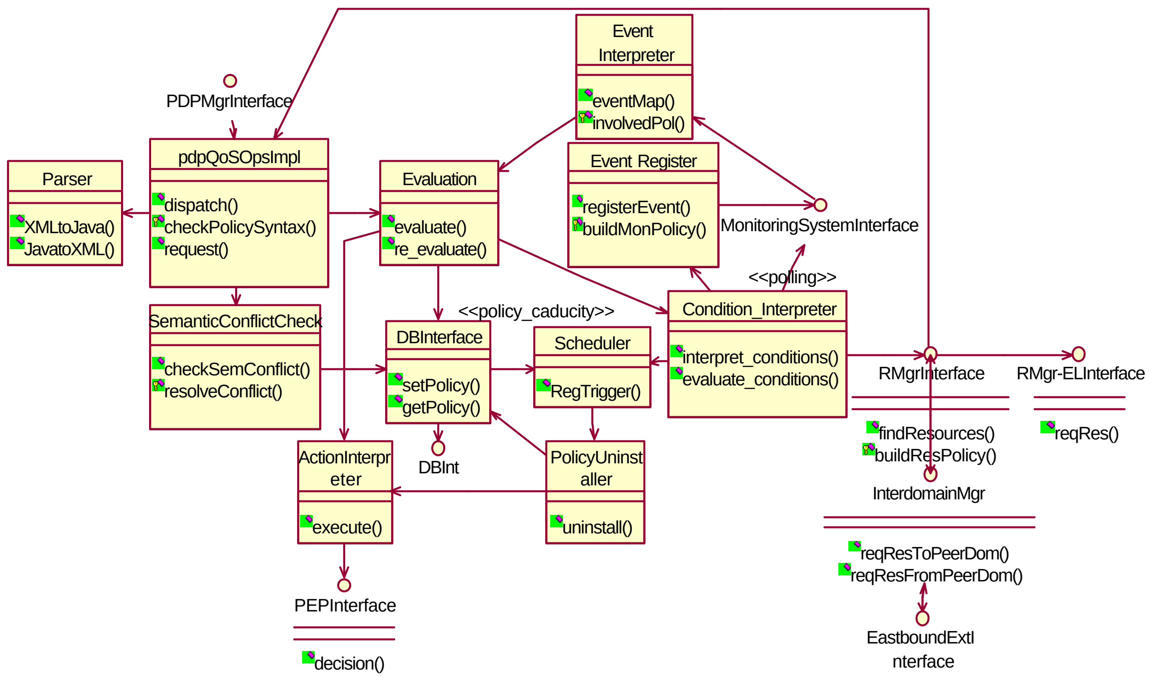


Figure 52 – PDP Component Class Diagram

5.2.4 QoSPEP Component

The PEP component at the network level has, from the conceptual point of view, the same functionality as the element level ones, except that at the network level they do not include signalling support functionality for the reasons previously exposed. However, the concrete processes needed for realising the functionality (i.e. translation of policy decisions into the target understandable commands) 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 53 in two ways:

- First the *map action to interface* functionality is now oriented to the translation of network to the element level policies,

- Second, since the policies between the network and the element level stations travel expressed in XML for several reasons that were already carefully described in FAIN Deliverable D3, we have included the translateToXML functionality, which covers this task.

Another important difference in the functionality of the PEP component (i.e. the QoS PEP component) regarding the element level is the de-multiplexing of the enforcement command to the appropriate PBANEM systems.

Finally, the dynamic conflict checking functionality appearing at the element level is not need at the network level and thus not designed.

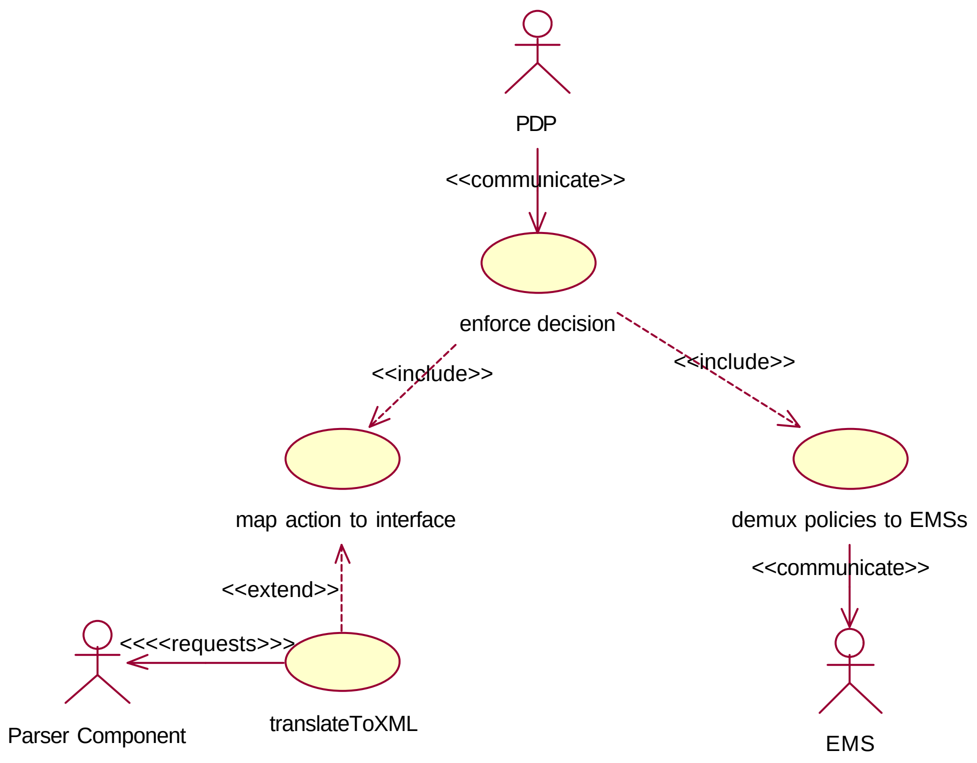


Figure 53 – QoSPEP Component Use Cases

In Figure 54 we can see which classes develop the PEP functionality at the network management level. As for the element level the PEPImpl class controls the decision enforcement process and, obviously, the classes that supported the signalling and dynamic conflict checking functionality at the element level have been removed. The mapping of the use cases to the classes that realise this functionality is the next one:

Enforce decision: The functionality of this use case is developed by the PEPImpl class, it basically initiates and keeps control of the enforcement process.

Map action to interface and translateToXML: The IntMapper is the responsible class for these two functionalities. The two functions of this class mapAction() and JavatoXML realise each one of the two functionalities respectively. In order to parse the JAVA element level policy into the XML policy the IntMapper class contacts the Parser component included within the PDP so as to avoid unnecessary replication of functionality.

Demux policies to EMSs: The functionality of this use case is realised by the EMS_Demux class which extracts from the XML element level policies the active nodes where they should be applied and maps this information to the corresponding PBANEM system associated to those active nodes. Then, it forwards each policy to the corresponding element management station.

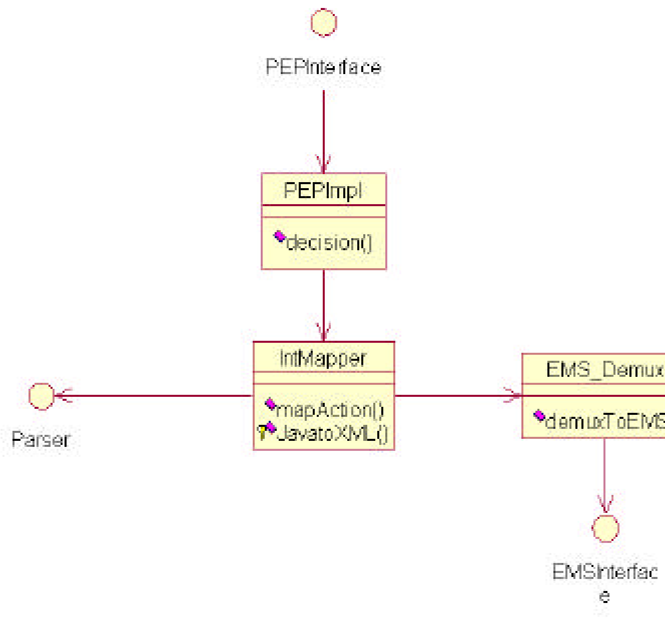


Figure 54 – QoSPEP Component Class Diagram

5.2.5 Network level Delegation PDP

The Basic functionality of the Delegation PDP in the network level is similar to that of the element level. However, from the network administration viewpoint, the functions dealing with notifications, which are sent, from active nodes or EMSs and the functions interacting with the end user (operators of ANSP and SPs) should be mentioned in the network level.

5.2.5.1 Delegation PDP use-cases

The Main use cases of the Delegation PDP in the network level are similar to the ones in element level, which are namely for Configure, Operate and Reconfigure stages. In Deployment, incoming policy data are checked and forward to Delegation PEP in Network Level. After deployment of policy data, Delegation PDP moves in Operate stage, which are checking the status of access right with information from Monitoring System of network level and PBANEMs. Here we can show the use case of Operate, which deals notification function. With this function, operators can know the status of access rights in their virtual networks.

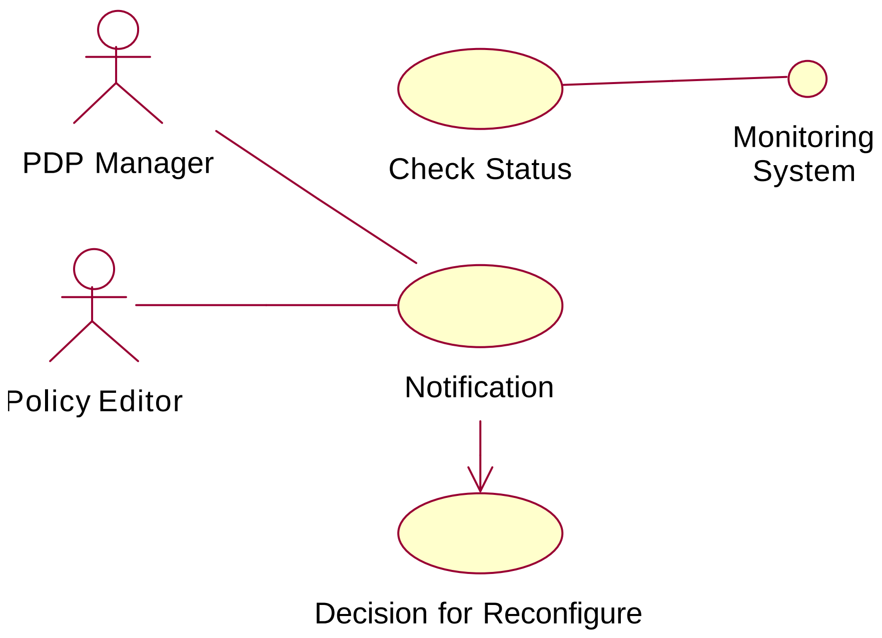


Figure 55: a Detailed Use cases of Operate stage in Delegation PDP

5.2.5.1.1 Messages

Notifications from Delegation PDP are made when necessary. These messages are sent both the synchronous and the asynchronous way, then finally arrived to Policy Editor or other components to display messages for users.

(1) Result of Configuration/Change

When the configuration of a new customer is completed successfully, this notification is sent. On the contrary, if the configuration of a new customer fails, an error message is sent with the alarm level and the error type.

A) Configuration error types

-policy check error

-policy syntax error

-policy semantics error

(2) Other Errors

-Invalid Operation: a customer's operation mismatches the attribute.

-Node Error: A node is down.

(3) Alarm Level

An alarm level is included in the error messages in order to indicate how critical is an error.

A) Critical

B) Warning

C) Notification

5.2.5.1.2 Delegation PDP Class Diagrams

Since basic functionality of delegation PDP in network level is similar with one in element level, most of classes of element level could be reused in this level.

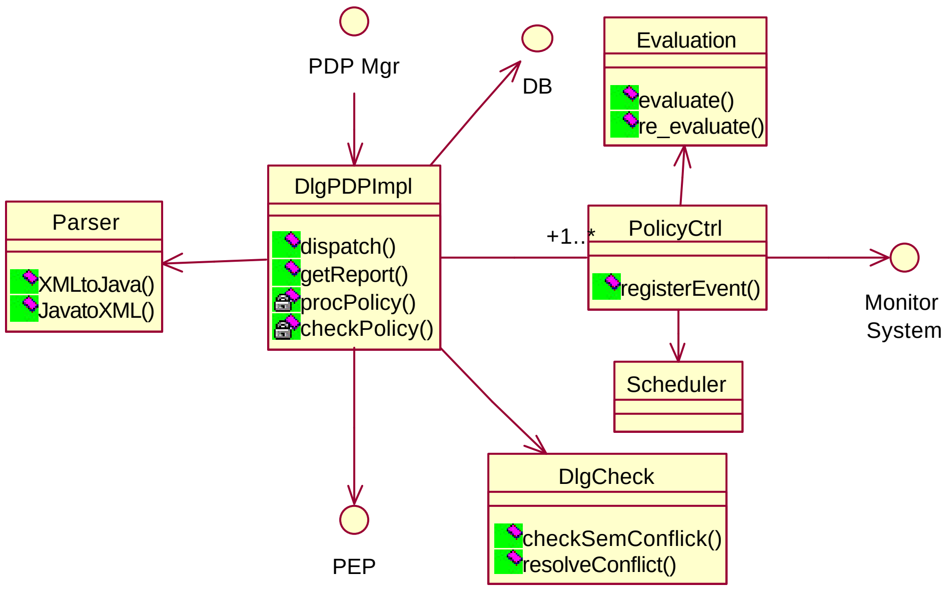


Figure 56: Class Diagram of Delegation PDP

5.2.6 Delegation PEP at the NL

The functions of the delegation PEP at the network level are: Firstly, it receives the delegation policies from the delegation PDP, it makes the necessary translations and then it delivers them to the EMS (namely the EL ANSP proxy).

We will use a policy example in order to make the functionality of the NL delegation PEP more vivid. An SP sends the following policy[7]:

"For the Edge nodes of my Virtual network that exist in the United Kingdom, I want medium security, and for the Core nodes I need high security".

[7] For the sake of simplicity, we assume that the SP uses the ANSP management framework, and since the ANSP has the rights to perform any kind of operations, there are no checks needed as to whether the ANSP has the rights to install the policy.

This Delegation policy will arrive through the Delegation PDP, to the DlgPEP. The DlgPEP will extract the proper parameters from the policy and will substitute certain fie lds in it. In our example it will substitute “high security” with ReadOnly (RO), and “medium security” with Read/Write (RW). It will also analyse that edge nodes are the nodes described in e.g. IP address list 1 and Core nodes are the nodes describ ed in IP address list 2. Thus the NL Delegation PEP will create brand new EL Delegation policies. It will then send these policies to the ANSP proxy of the element level.

In the case that we find ourselves inside the SP management domain, which means that the SP is responsible for deploying and enforcing the policies, the SP’s ability to enforce the particular policy will be questioned. The NL Delegation PEP will create a restricted schema and store it in the Schema repository, where it will be collected by the Access Control Check (ACC) component. The Access Control Check component will check if the SP is able to enforce a particular policy by comparing this policy against the restricted schema. If the outcome is positive, the PDP manager will send the policy to the delegation PDP for further processing.

There is a special type of policy coming from the Delegation PDP that dignifies that a new user wants to instantiate management components. In that case the Delegation PEP will use the instantiateDom() method offered by the EL PDP manager interface. The parameters passed to the PDP manager should be the name of the entity that wishes to instantiate (e.g SP) and the components that the entity wishes to be instantiated. The PEP will receive the result of the instantiation procedure.

5.2.6.1 Use cases of the NL Dlg PEP

The following use cases diagram captures the above iterations.

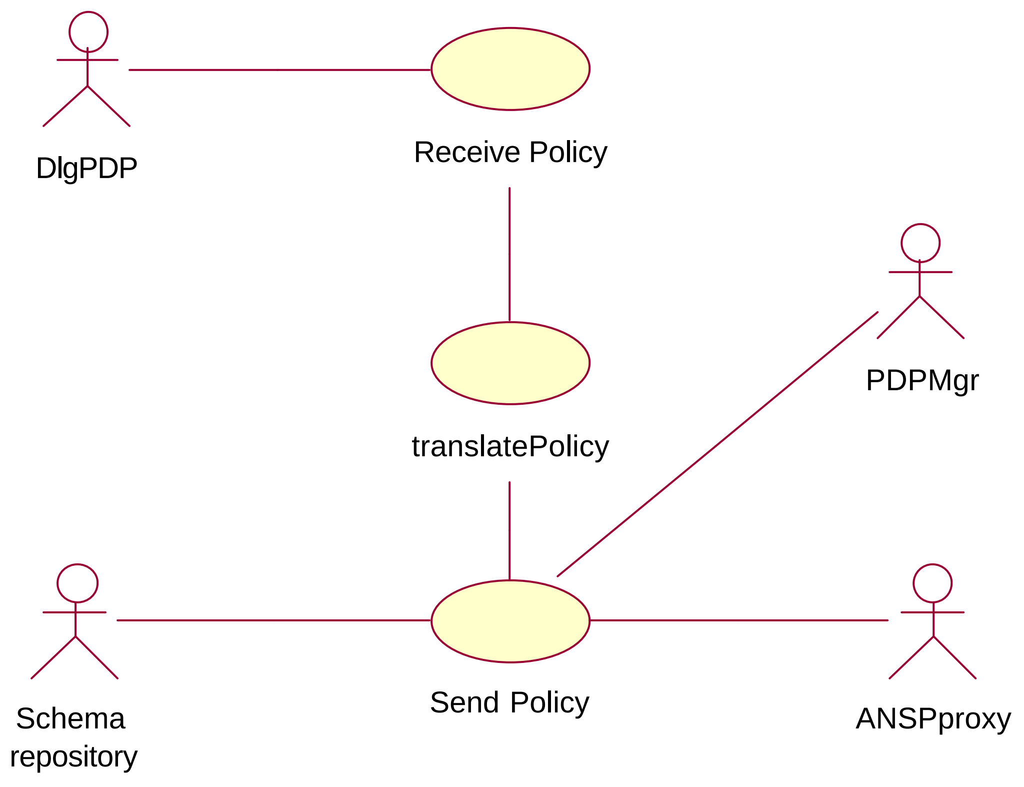


Figure 57: Use cases of NL Delegation PEP

5.2.6.2 Class diagram for the NL Delegation PEP

The corresponding class for the delegation PEP at the network level is the following:

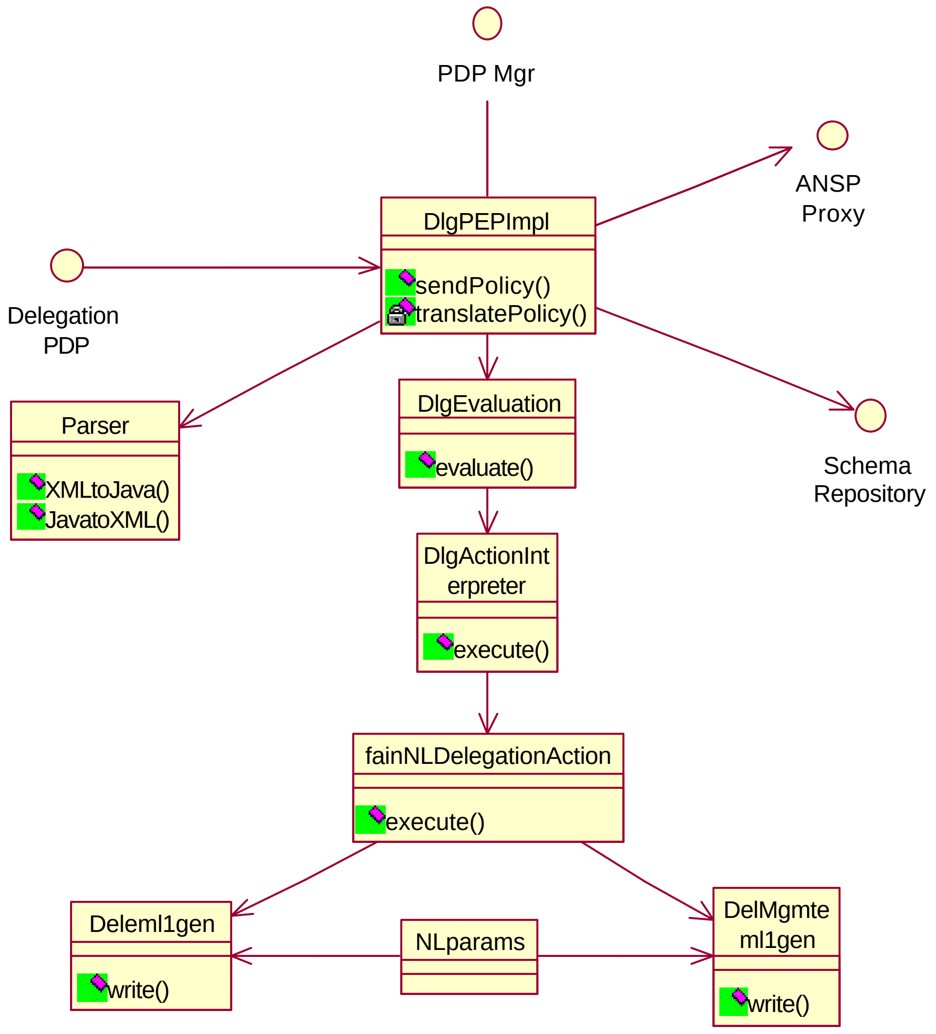


Figure 58: Class diagram of NL Delegation PEP

The sendPolicy() method is used by the delegation PDP in order to pass the policy to the delegation PEP for enforcement.

The translatePolicy() method is used internally by the Delegation PEP. In the policy example given above, this translation can be from vague router names such as Core or Edge to specific IP addresses of routers. The Deleml1gen and the DelMgmteml1gen classes generate the two EL Delegation policies that will be sent to the EL ANSP Proxy.

5.2.7 Conflict Check Component at NL

The text at section 4.2.10 for the conflict check component at the element level applies as well to the network level of this component, since the requirements, the functionality and the approach taken are the same.

5.2.8 Monitoring system-NL

Although a monitoring system at network level has been identified, it remains to define how it will operate. This section glances at some issues that have been identified and being on the list of a potential solution to be investigated. As the whole network level, the NL-Monitoring system would be in charge of gathering the state of resources in a given domain nodes in order to provide the other network level components (DelegationPDP, QosPDP, NL-Resource Manager) with domain wide resources information, refreshed at a time scale to be dimensioned.

To assume efficiently this role, the NL-Monitoring system subscribes to Monitoring Systems present on each Active Node of its domain. Then it registers the events which status it would like to receive either at some given interval of time or on demand. The choice for such refreshment of resources' status is related to their natures. Further investigations in this direction to integrate fully this view in the management architecture have been elicited and are being developed.

Figure 59 illustrates the network level monitoring system relationship to the EMS monitoring systems in a given administrative domain.

As well as the interactions with EMS monitoring system, NL Resource Manager and NL Delegation PDP in a given domain are obvious, the reflexive interaction (between) NL Monitoring system still needs to be developed. Indeed this interaction will be strongly connected to the FAIN network model solution vis à vis of sub-networking solution and inter-domain management that are still being tackled.

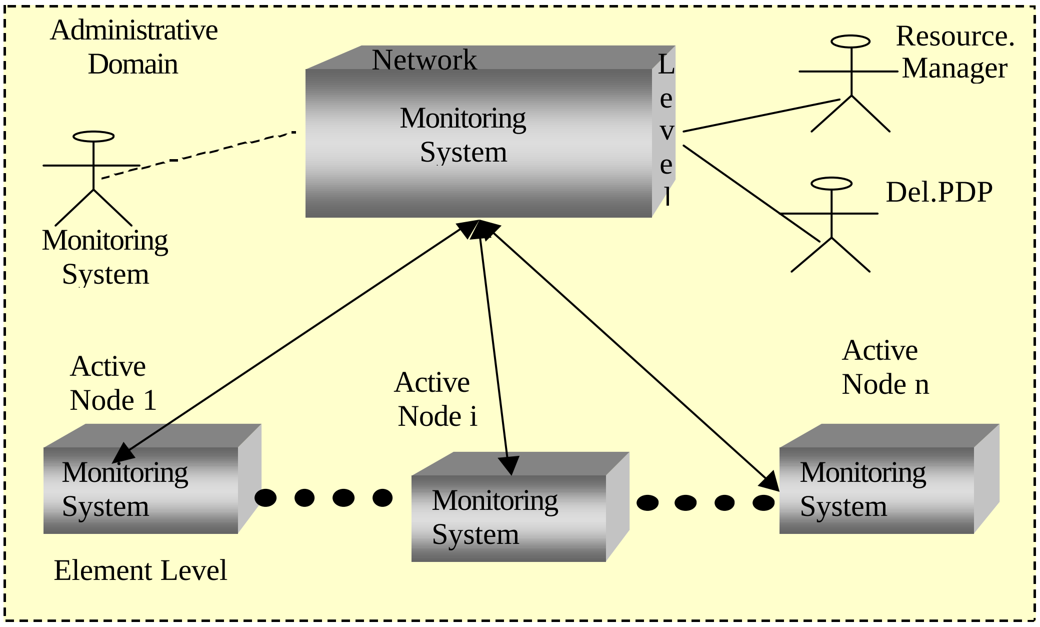


Figure 59: NL and EL monitoring Systems relationship

6 R16 ACTIVE SERVICE PROVISIONING – ASP

The Active Service Provisioning (ASP) is the sub-system of the FAIN Management system in charge of providing deployment of new or needed services in the relevant area of the FAIN network on demand either by the network operator or the network users (Customer, Service providers). The goal of this chapter is to focus on this functionality of the of the FAIN architecture pointing out its main features. To this instance, an incremental description from capturing main functionality, thus the main components to a deeper description of those components will be adopted through this chapter.

6.1 ASP use cases

Figure 61 depicts the main use case diagram of the ASP system. The main use cases of the ASP sub-system are:

- **Release service**. It describes the capability of the ASP system to make a service available for deployment in the active network.
- **Deploy service**. It describes the capability of the ASP to deploy a released service in a target environment within the active network.
- **Remove service**. It describes the capability of the ASP system to remove a service from a target environment.
- **Withdraw service**. It represents the capability of the ASP system to withdraw a service from a list of available services in the active network.

The main actors communicating with the ASP system are:

- Service Provider,
- Active Network Service Provider
- Network Infrastructure Provider,

The roles are described in the FAIN Enterprise Model in detail.

The ASP system capabilities represented by the main use cases are related to each other in that there is a valid sequence in which they occur for a given service. An activity diagram in depicts these relationships.

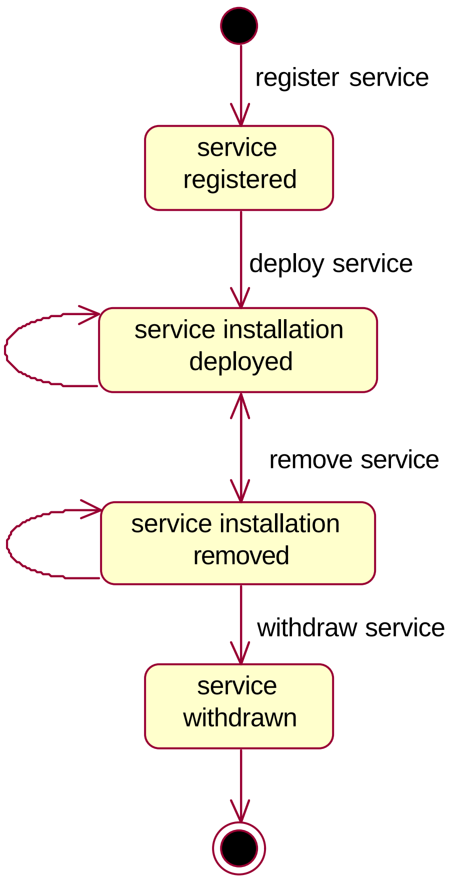


Figure 60 Activity diagram for an active service processed by the ASP.

- First, a service is **released** by the Service Provider in the active network. This means that the service provider makes the service available to the users by registering the service in the active network and storing the service metadata and service code modules with the ASP system.
- After having been released, a service may be **deployed** to a target environment in the active network. The Service Provider initiates this process by interacting with the ASP. In order to find out the target environment required by the service requirements, the ASP communicates with the Active Network Service Provider (who may also interact with the NIP) to request the information about the state of the network and to allocated needed resources to the service.
- A deployed service, i.e. a service installation, may be **removed** from the target environment it has been deployed to, if needed. Some interaction with Active Network Service Provider may occur when removing a service.
- Finally, the Service Provider may withdraw a service from a network.

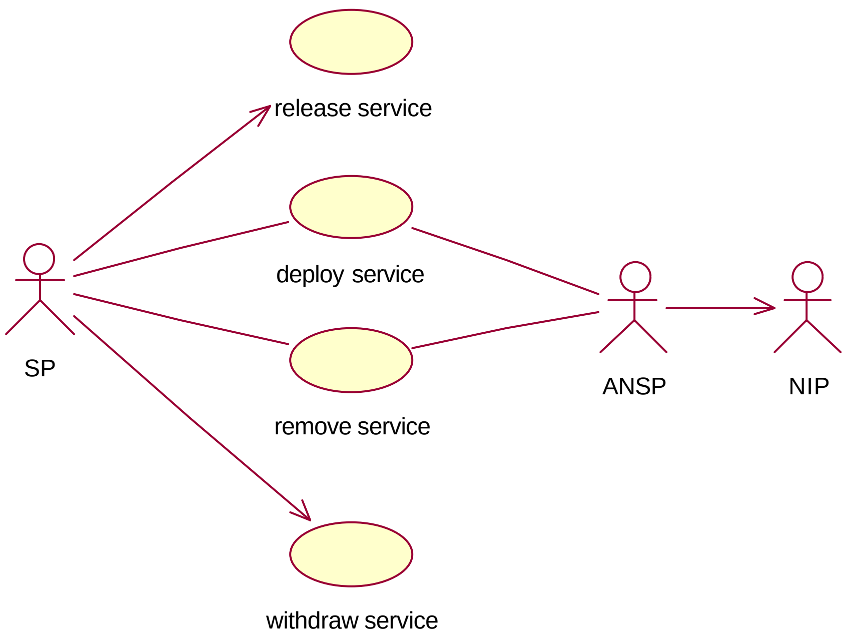


Figure 61 Main Use Case of the ASP.

The following subsections explain the use cases identified above.

6.1.1 release service

The Service Provider who decides to offer his service in the active network has to release it in the active network. It does so by contacting the NMS, which allows accessing the service release capability of the ASP. Figure 62 shows the `release` use case diagram. The service is released by registering its name and some deployment information (a list of required service component descriptors) with the network service registry, and uploading the service code including all the dependent code into the network-wide service repository.

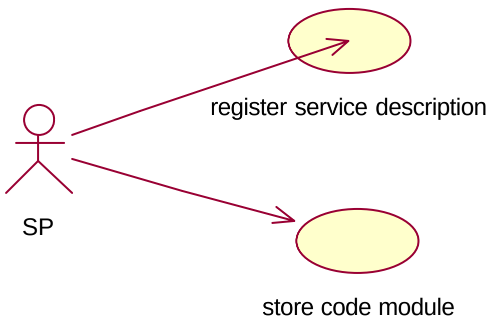


Figure 62 release service use case diagram.

6.1.2 deploy service

After the service is released in the network, the Service Provider may want to deploy his service to a specific target environment, which is most suitable for the given Consumer requesting access to the service. A target environment is formed by a set of active nodes on which the code modules of the active service are deployed.

Figure 63 depicts `deploy service` use case. The deployment process starts at the network level with mapping the service properties to target environment properties. The ASP identifies the target environment of the service in that matches the information about the current state of the active nodes available to the Service Provider (in terms of their static and dynamic properties and capabilities) against the requirements of the service described in the service component descriptor. This capability is represented as the `map service to target nodes` use case included by `deploy service` use case.

When deploying a service, downloading of the service is usually needed. It consists of downloading deployment descriptor (`download descriptor` use case) and its code modules (`download code module` use case) as well as all other services it depends on. To discover these dependent services, service dependencies are resolved (`resolve dependencies` use case). The fetched code modules may be cached locally on the target active nodes (`cache code` use case).

Another capability of the ASP is service installation. The service code modules fetched onto the node have to be installed in the appropriate execution environments (EEs) of the target environment. The ASP installs the code modules by performing some EE-independent pre-configuration of the service code modules and making them available to the target EEs. Some interaction with Network Infrastructure Provider (the active node) is needed to perform the latter deployment steps.

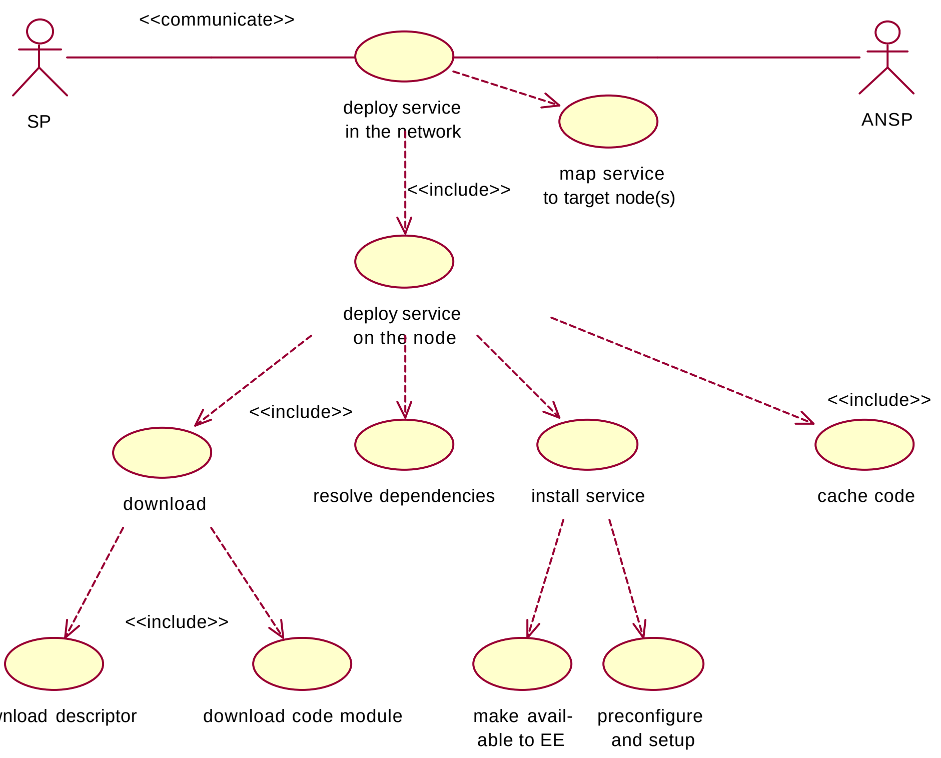


Figure 63 deploy service use case diagram.

6.1.3 remove service

The Service Provider (or Network Management System representing him) may request to remove a deployed service from the environment it was deployed in. This ASP capability is described in remove service use case depicted in Figure 64. The SP communicates with ASP system at the network level in `remove service from network` use case. The ASP identifies the target environment of the deployed service and removes the service from every active node forming the target environment (`remove service from node` use case). It includes:

- resolving dependencies of the service components to identify all dependent service components to remove.
- uninstalling the service components from the execution environments of the target environment. This includes removing the code modules from the execution environment and resetting the execution environment to the state before the service component installation. The uninstall service involves some interaction with Active Network Service Provider.
- and removing the code modules from cache if needed.

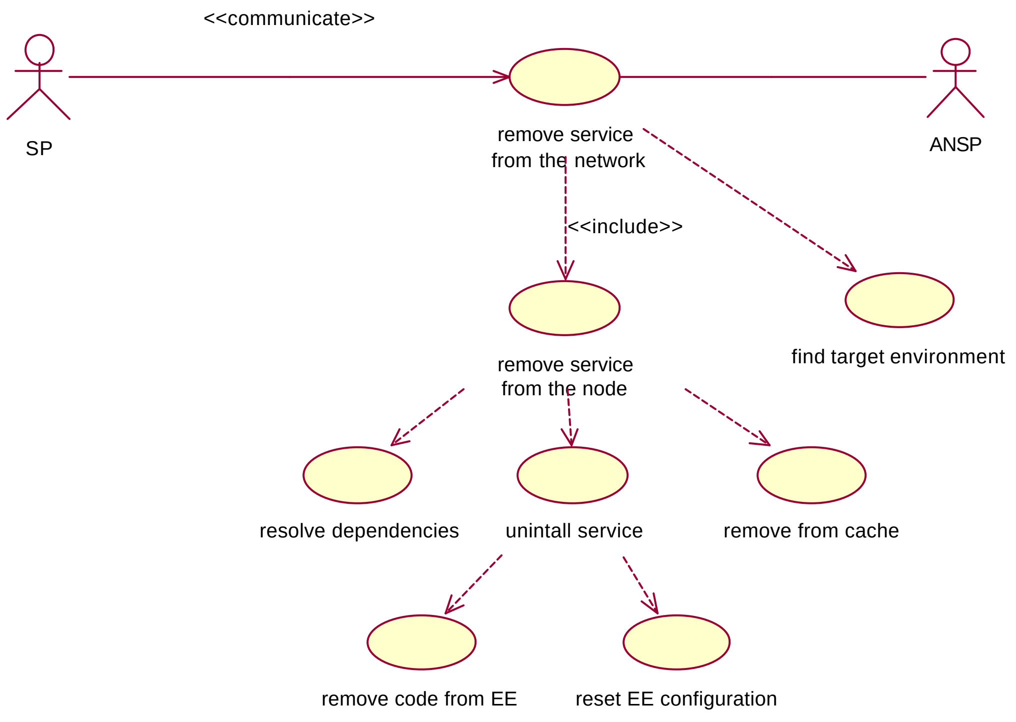


Figure 64 remove service use case diagram.

6.1.4 withdraw service

The SP who release his service in the network, may also want to withdraw the service from the active network. The `withdraw service` use case describes the capability of the ASP to unregister the service from the network service registry (`unregister service description` use case) and to discard the service code modules and their dependent code modules from the network service repository (`discard code module` use case).

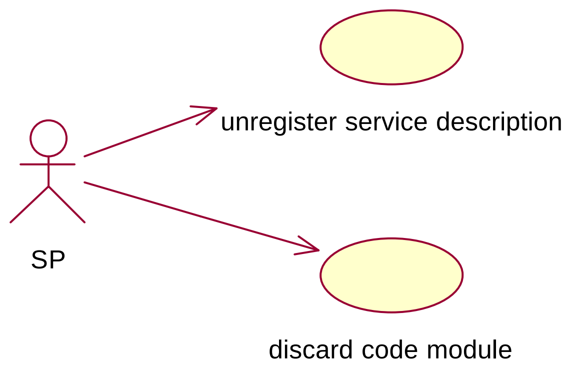


Figure 65 withdraw service use case diagram.

6.1.5 manage service installations

When deploying or removing a service from its target environment, the ASP performs management functions on the database with all service components installed on the active node. The `manage service` installations use case describes the capabilities of the ASP to manage the service components on the node that the service provider may use, like service component's code expiry.

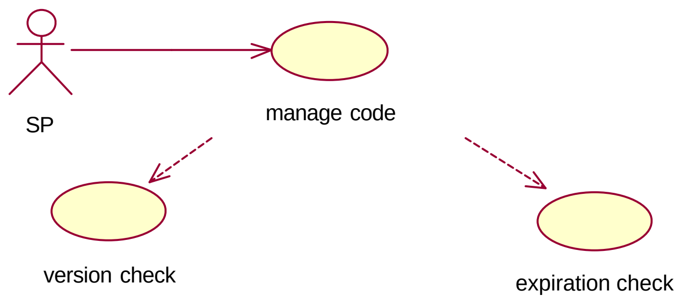


Figure 66 manage service installations use case diagram.

6.2 Components description

From those well-identified and isolated functionality of the ASP, relevant components have been identified and aligned to the 2-tiers Architecture assumption when needed. For more information on this aspect of the design, we invite the reader to have a look to the deliverable D3 [8]. The following describe those components.

6.2.1 Network ASP Manager

The Network ASP Manager is the initial point for the Active Service Provision on network level. An FAIN Service Provider or Customer which are authorized to take the role of a Service Provider may act as an ASP manager/user (person or component) in order to initiate a service deployment either via an IDL or GUI interface.

6.2.1.1 Network ASP Manager use case diagram

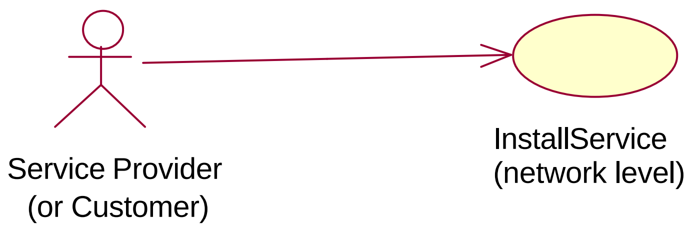

Service Provider (or Customer)
InstallService (network level)

Figure 67 – Network ASP Manager use cases

6.2.1.1.1 installService

For execution of the installService operation, the network ASP manager fetches the service deployment descriptor, which is an XML document, from the Service Registry. The descriptor is partially processed by the ASP network manager to determine an optimal code distribution on the network level, i.e. to find a set of active nodes that the service components have to be deployed onto. For a special case (as for M4) the user of this operation may specify the nodes on which to deploy the service explicitly.

The network ASP manager sends the deployment request in form of a mobile deployment agent to each of the active nodes' Node ASP Manager. The deployment agent is sent in an active packet that is dispatched to the node level ASP on each node. The deployment descriptor is then passed to the Node ASP Manager (agent system), which is the access component of the node level ASP for handling the deployment request. The Node ASP manager may perform additional functions and passes the deployment request with its description to the Service Creation Engine and Code Manager.

After execution of the install/deploy Service on one node, the deployment agent travels to next specified active node. After finishing execution on all active nodes, the deployment returns to the originator Network ASP Manager and passes back the interface references to the installed/deployed service components. These installed components can afterwards be instantiated and configured.

6.2.1.2 Network ASP Manager design

The Network ASP manager is realized as an stationary agent with a Graphical User Interface running in a priviledged Java i.e. agent system execution environment.

The Network ASP Manager interacts with the Service Registry and will generate a mobile deployment agent, which travels encapsulated in an ANEP packet to the Node ASP Manager (agent system).

The Network ASP Manager requires a connection of the Demux to the Agent Environment (Grasshopper), which is realized by using the Communication Service facilities of Grasshopper.

The main operation of the Network ASP Manager is the installService operation on a network level.

6.2.2 Node ASP Manager

The Node ASP Manager is the initial point for the Active Service Provision on node level. The Node ASP Manager is connected to the Demux (WP3) component, from which it receives the request for Active Service Deployment/Provision from the Network ASP Manager in form of a deployment agent.

6.2.2.1 Node ASP Manager use case diagram

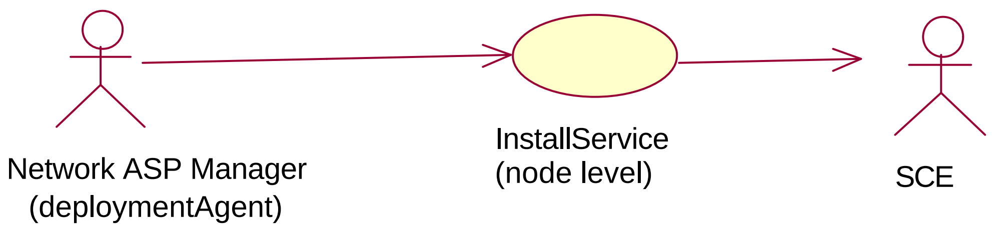


Figure 68 – Node ASP Manager use cases

6.2.2.1.1 installService

For deployment of a service on a node, a deployment agent arrives on a Node ASP Manager. The deployment agent is created from a Network ASP Manager (although the deployment might come from a neighbour Node ASP Manager, if a service has to be deployed on a set of active nodes).

The Node ASP manager may perform additional functions and passes the deployment request with its description to the Service Creation Engine and Code Manager. The result of the installService operation of the Service Creation Engine is collected from the deployment agent. Potentially, the deployment agent travels to next specified active node if more than one node is given as target nodes for deployment. After finishing execution on all active nodes, the deployment returns to the originator Network ASP Manager and passes back the interface references to the installed/deployed service components.

6.2.2.2 Node ASP Manager design

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).

The main operation of the Network ASP Manager is to provide an “installService” operation on a node level.

6.2.3 Code Manager

The Code Manager is a node-level ASP component, which maintains the information about the code modules installed on the node. It also supervises the process of fetching, installation and withdrawal of the service code modules. Code Manager mediates fetching service component descriptors, as well.

6.2.3.1 Use Cases

Figure 69 depicts the use cases of the Code Manager.

Code Manager provides a capability to fetch and install a service. The component is contacted after the Service Creation Engine has resolved the dependencies of the service component requested to be deployed (`fetch and install service` use case). Code Manager receives information what code modules to fetch from the Service Registry and install them in what execution environments of the node and supervises the fetching process (which is performed by the Local Service Repository) and installation process (performed by Node Manager). This use case includes managing the data base of the code modules installed on the node (`manage installation` use case).

Another capability of the Code Manager is to uninstall given code modules from their execution environments on the node (`uninstall service` use case). Code Manager receives the information on code modules and their execution environments from SCE after it has resolved dependencies of the servic e. The uninstallation process involves updating the data base with installed code modules maintained by Code Manager.

Code Manager also mediates in fetching service component descriptors from the Service Registry. It communicates with the Local Service Registry, which represents the Service Registry on the node.

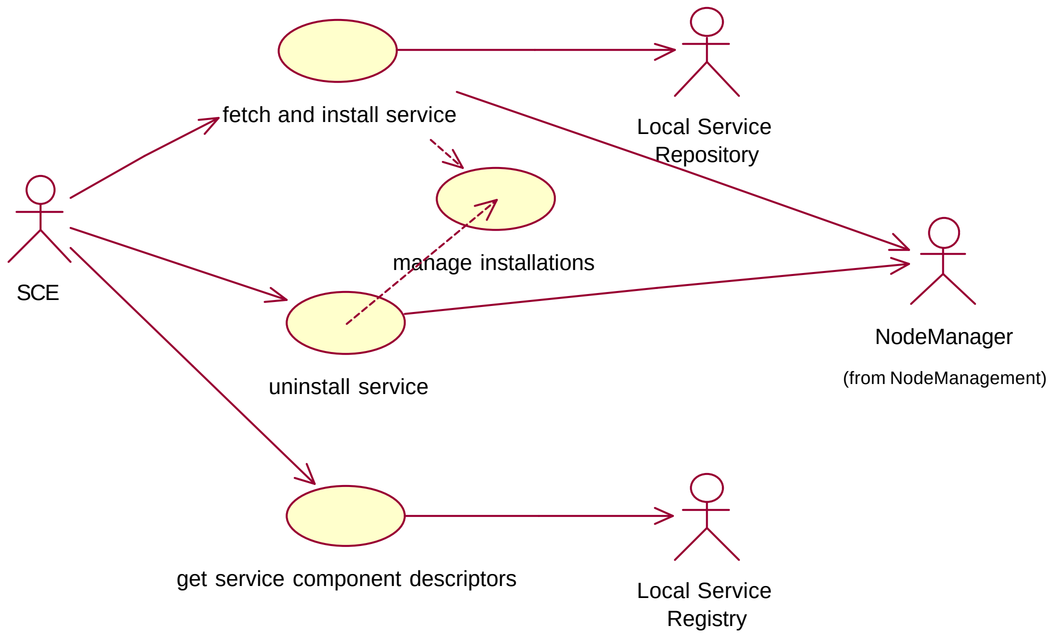


Figure 69 main use case diagram of the Code Manager.

6.2.3.2 Code Manager Design

Code Manager provides an Element ASP-internal interface. This interface is used by the SCE and includes definitions of the following operations:

- `FetchAndInstallService` triggers fetching and installation of a given list of code modules belonging to the service to deploy. The operation contacts the Local Service Repository to fetch the code modules and the Node Manager to install the fetched code modules. However, the installation process is triggered by Code Manager, it is performed by the EE-specific components of the Node Management Framework.

- `UninstallService` manages the uninstallation process. As input parameters, it receives a list of code modules to remove. It triggers the EE-specific uninstallation process communicating with Node Manager and updates the Code Manager data based maintaining information on code modules installed on the node.

`GetServiceComponentDescriptor contacts the Local Service Registry to fetch a descriptor list of released service components realizations for a request`

6.2.4 Service registry Component

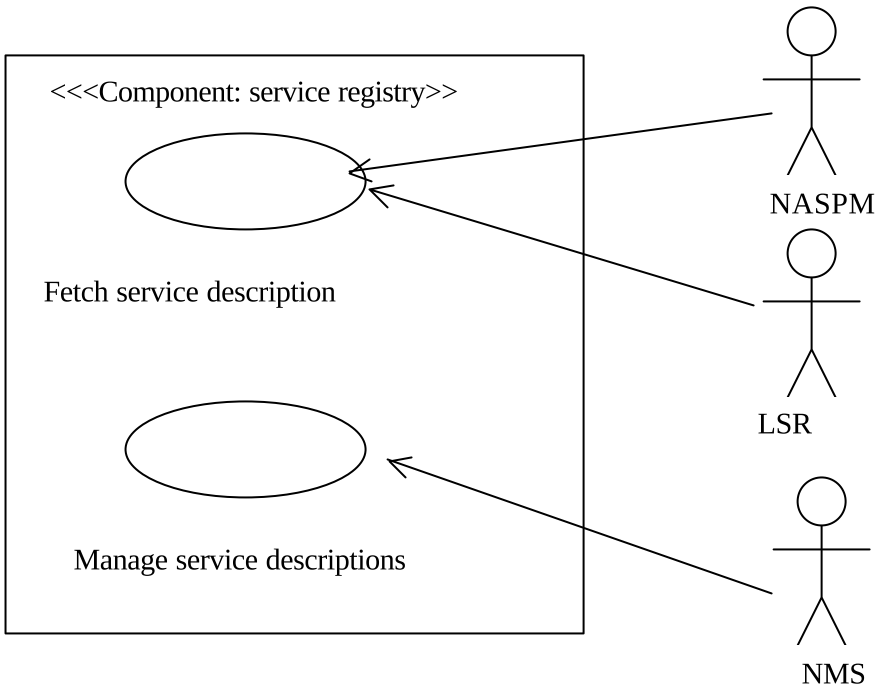
In the ASP part of the FAIN architecture, the service registry is in charge of managing the description of services that can be loaded into active nodes (register, unregister, find services).

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

The interfaces are detailed and based on the interfaces specified in the deliverable D3.

6.2.4.1 Service Registry Use cases

The actors are the NASPM, LSR and the NMS.


<<<Component: service registry>>
Fetch service description
Manage service descriptions
NASPM
LSR
NMS

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

Fetch service description Use case.

The first use case starts either when the NASPM needs to fetch the description of a service or when the LSR is asked by the Node Code Manager for the description of a service it doesn't have locally.

In the first case, the NASPM 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 asks for its descriptions (a sequence of XML file) by calling the method *fetchService*. There might be several descriptions for a service, then several XML schemes might be returned.

In the second case, the LSR then asks the descriptions for this service to the service Registry by calling the method *fetchService*.

The LSR can also get the list of the available service descriptions by calling the method *getServicesList* of the CORBA interface (if needed).

Manage service descriptions use case.

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*.

6.2.4.2 Service registry Design

The service registry must register new services, unregister old services when requested by the Network Management System.

It must get the list of all available services and the descriptions of a given service when requested by the Network ASP Manager.

It must get the descriptions of a service to the Local Service Registry when the latter doesn't have it already in cache.

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 are 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 in charge of 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.

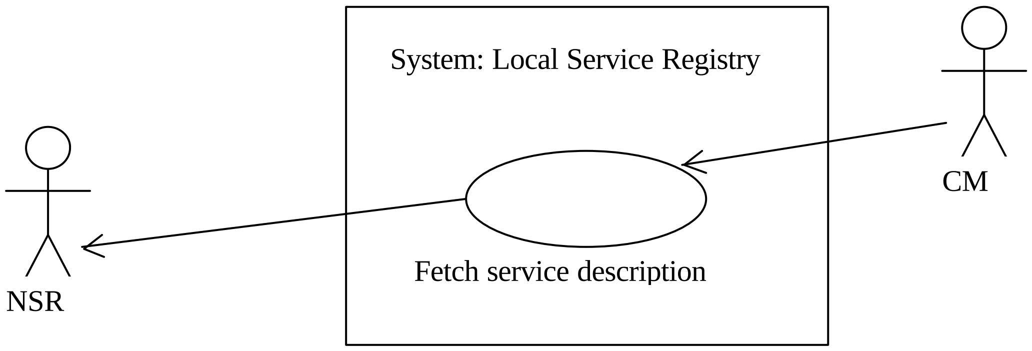
Some exceptions are also defined: checking the correctness of the name, the syntax of the XML descriptor….

6.2.5 The Local Service Registry

Inside an active node, the local service registry is in charge of managing the description of services that are requested by the Code Manager and can be loaded into active nodes. It the local service registry hasn't the description of the given service, it asks the network service registry for it.

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

6.2.5.1 Local Service Registry Use cases Diagram



The actors are: the CM and the NSR.

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

Use case Fetch service description.

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.

6.2.5.2 Local Service Registry Design

The Local Service Registry is in charge of 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 want 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's 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 (the same than the service Registry

- fetchService
- getServicesList

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

6.2.6 Local Service Repository

The purpose of the local service repository (local cache) is to decrease the latency for the installation of a new service. The local cache stores recently used service components in the node, so that if they are requested by a new service, they will not have to be downloaded from a remote code repository.

The amount of components that are cached depends on the available storage space on the node. In the case that available space is exhausted, a replacement algorithm must be 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.

The code manager is the main "client" of the local repository and it is the only other ASP component, which uses the local repository interface. The most important functionality required is the fetching of code modules, which are necessary for the instantiation of a new service on the node. The code manager requests a code component from the local repository. The repository then has to check if the module is already cached, or else it must be downloaded from the network. The local cache also carries out this operation. For this reasons the cache has knowledge of the location of the network-wide service repository, which stores the implementation components of the services, which are available in the network.

6.2.6.1 Local Service Repository use case diagram

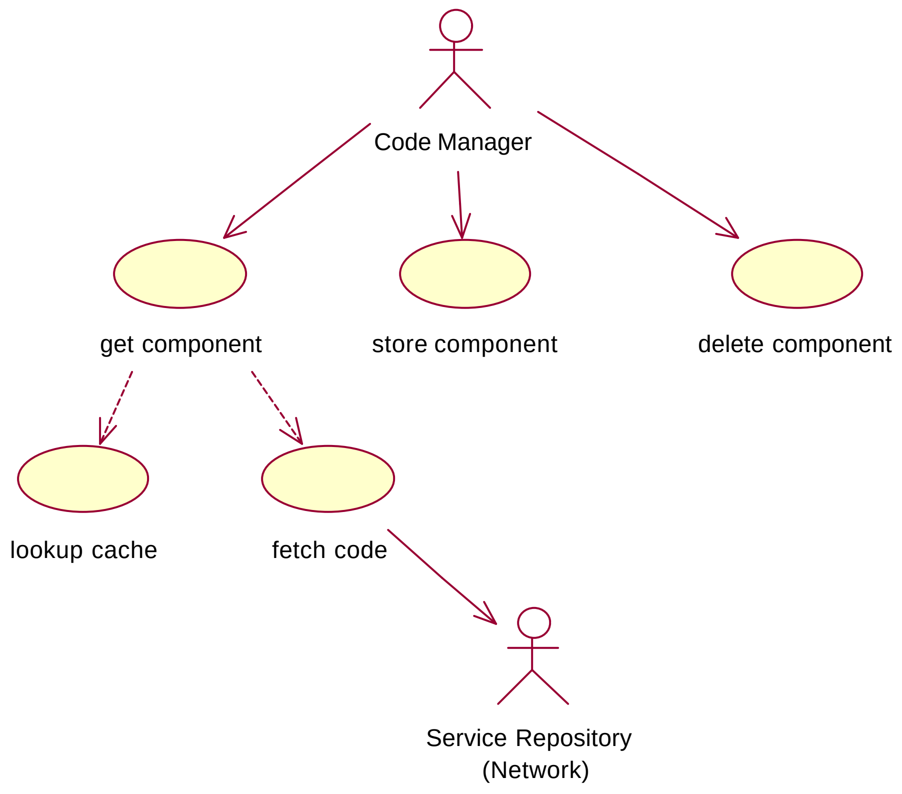


Figure 70 – Local Service Repository use cases

As we can see from the use case diagram, the actors that interact with the Local Service Repository are the Code Manager and the network-wide Service Repository. The use cases are described below.

6.2.6.1.1 get component

After a service descriptor has been parsed, the Code Manager is in charge of retrieving the necessary code modules, which implement the desired service. The Code Manager contacts the Local Service repository and requests the necessary components. From this moment the responsibility of locating and fetching the code modules passes to the Local Service Repository. The following two use cases are included:

- lookup cache

  The Local Service Repository first looks up in its cache, to see if the requested component is already available at the node.

- fetch code

  If the component is not cached locally, it has to be downloaded from a code server. The code module is retrieved from the Service Repository and is returned to the Code manager.

6.2.6.1.2 store component

As we mentioned above, the Code Manager manages the components stored in the cache, so it may request the storage of a new component.

6.2.6.1.3 delete component

As with storing, the Code Manager can also decide that it is necessary to delete a component from the local cache.

6.2.6.2 Local Service Repository design

It is foreseen that because of disk space restrictions the amount of components that will be cached in the node will be relatively small, so it is not necessary to use a database or a more complex storage system for the caching of code modules.

The code files are stored in the cache directory and a hashtable is used as an index of the local repository. The code manager can get a reference to the implementation files, when a component is requested.

6.2.6.2.1 Local Service Repository Interface

The interface of the local service repository is described in IDL. The operations provided are the following:

- `getComponent`

  This operation is responsible for retrieving a code module. The repository first checks if the requested component is cached locally. If the code does not exist locally, it is downloaded from the network service repository. 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.

The IDL description of the Local Service Repository Interface is the following:

```
typedef string CodeModuleID;
typedef string CodeModuleRef;

interface ServiceRepository {

     boolean storeComponent(in CodeModuleRef codeComponent,
                in CodeModuleID componentName);
     boolean deleteComponent(in CodeModuleID componentName);
     CodeModuleRef getComponent(in CodeModuleID componentName);
};
```

6.2.7 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 main idea is that 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.

A service can be created by assembling together a set of required components (creating a package), which are specified using a descriptor. The components that make a service package can either be grouped together in a single archive, or they could be stored individually. In the service repository we choose to have each component in a separate file for the following reasons. First, because one component may be used by different services, so it could belong to different packages and second because a service could be updated by replacing one of each component with e.g. a new implementation version.

The exact format of the stored components varies according to the type of the Execution Environment, for which they are designated. For example, an implementation for a (Java Virtual Machine) Jvm-based EE may be a Java.jar file, while an implementation for a high-performance EE will be in a native object file.

6.2.7.1 Service Repository use case diagram

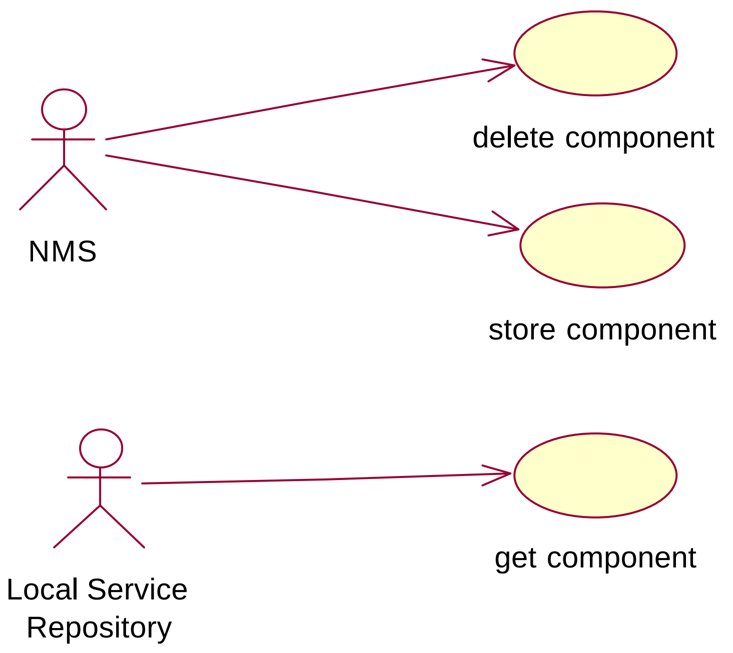


Figure 71 – Service Repository use cases

6.2.7.1.1 store / delete component

The addition/deletion of code modules in the repository is performed by the Network Management System, as described in the corresponding use cases store/delete component. A new service first has to be registered with the Service Registry and then its components are stored in the repository. Likewise, when a service is removed from the network, first its description must be removed from the Service Registry and then its code modules will be deleted from the Service Repository.

6.2.7.1.2 get component

The Local Service Repository residing on an active node may request the download of a specific code module to that node.

6.2.7.2 Service Repository design

The main functionality required from the Service Repository is similar to that of the local cache. However, while we do not expect the local cache to store a large number of components, this is not the case with the Service Repository. It acts as a code server, which contains all the service implementations, which may be installed in the active nodes of the network. For this reason, scalability should be taken into consideration for the design and implementation of the service repository. The component files are stored in the repository using a directory-based structure. Each component stored in the repository must have a unique name. The code modules are stored using a pathname, which is determined using the information such as file name, developer name, target Execution Environment type, and implementation version.

The main operations of the Service Repository are the following:

- `storeComponent`
- `deleteComponent`

These two operations are used to add/remove software components to/from the repository. These operations are used by the Network Management System and they are not available for a user of the system, as the storage and deletion of a service component are coupled with the registration or deregistration of the corresponding service. Accordingly, the Network Management System offers an interface for the service providers to make their services available and then it uses the ASP interface to register the service and store its components.

- `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.

6.2.8 The Local Service Creation Engine (SCE)

This document describes the Local Service Creation Engine (SCE), which is a sub-component of the active service provisioning (ASP) component offered by a FAIN active node [1][2]. The SCE is responsible to map a service component name to an implementation suitable to the local node environment.

6.2.8.1 SCE use case diagram

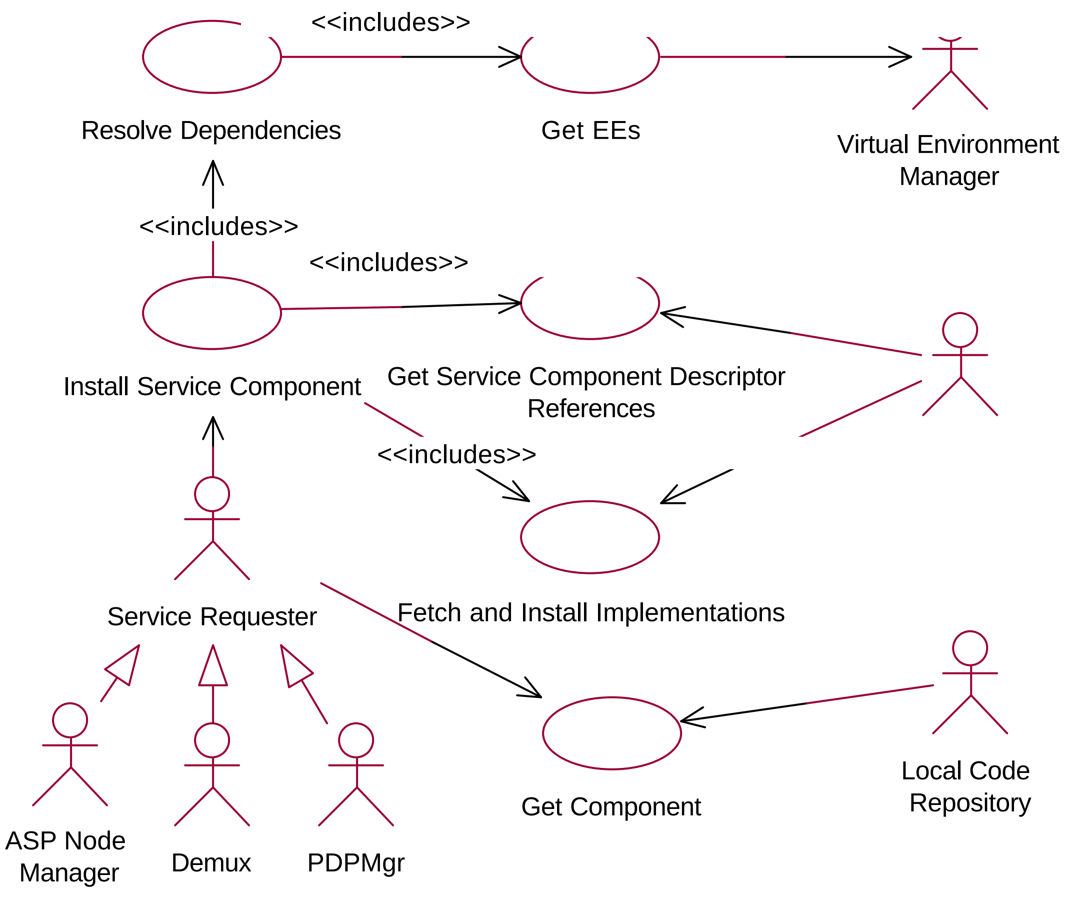


Figure 1 Service Creation Engine Use Cases

6.2.8.1.1 Install Service Component

This use case starts when a service requester requests the installation of a service component into a particular VE. 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 (included use case “Get Service Component Descriptor”) from the code manager. The SCE further consults the Virtual Environment (VE) manager to get information about the available Execution Environments (EE) in a specific Virtual Environment (VE) (included use case “Get EEs”).

From the list of service component descriptors, the SCE selects – based on the mapping policies and 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 (included use case “Resolve Dependencies”).

A service component descriptor might also contain a reference to code. If such a service component is selected by the SCE, the necessary information is stored in the installation map. The resolution process terminates when all dependencies are resolved. 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 (included use case “Fetch and Install Implementations”).

6.2.8.1.2 Get Component

This use case starts when the PDPMgr wants to download a management component. The SCE gets an implementation identifier and delegates the downloading of the appropriate implementation to the local service repository. Dependencies are not resolved. The PDPMgr executes installation and configuration of the module autonomously.

7 CONCLUSION

This document has provided a revised outline of the case studies that are to be used to evaluate the overall FAIN approach and associated architecture. The work documented here has focused in particular on the design and development of case studies associated with policy based network management and the dynamic provisioning of services (ASP) in an active networking domain. A trade of between a too detail presentation of the design and a focus on components main purposes has been adopted to ease reading of this document.

The system implementation is currently on going with the development of components that realise these case studies. An advanced version has already been developed, and is being demonstrated as M4 and part of M5 major events. This work is of course on going and as such it is likely that the prototypes will undergo further refinements and enhancements as the work on FAIN progresses. As such, it is expected that the prototypes and the scenarios that they support will evolve into a more complete final FAIN demonstration. This will highlight the success of the general overall approach, i.e. that the FAIN architecture is sufficiently well defined (through generic components and interfaces between the different actors) to support a variety of different services and management capabilities in a dynamic (active) manner. This will also show the overall benefits of an active networking approach for service provisioning and network management.

8 APENDICES

9 ACRONYMS

AC: Active Code

AN: Active Networks

ANE: Active Network Element

ANN: Active Network Node

ANSP: Active Network Service Provider

API: Application Programming Interface

ASN: Abstract Syntax Notation

ASP: Active Service Provisioning

BML: Business Management Layer

CDB: Conflict Detection Block

CIM: Common Information Model

CLI: Command Line Interface

CMIP: Common Management Information Protocol

COPS: Common Open Policy Service

CORBA: Common Object Request Broker Architecture

DAP: Directory Access Protocol

DCE: Distributed Computing Environment

DCN: Data Communication Network

DEN: Directory Enabled Networks

DIT: Directory Information Tree

DME: Decision Making Entity

DMTF: Distributed Management Task Force

DPE: Distributed Processing Environment

DSCP: Diffserv Code Point

EE: Execution Environment

EM: Element Management

EMS: Element Management System

FAIN: Future Active IP Networks

FAIN TA: FAIN Technical Annex

FCAPS: Fault Configuration Accounting Performance Security

FIFO: First In First Out

GDMO: Guidelines for Definition of Managed Objects

GUI: Graphic User Interface

IDL: Interface Definition Language

IETF: Internet Engineering Task Force

ISE: Information Storage Entity

ITU: International Telecommunication Union

JDBC: Java Database Connectivity

JNDI: Java Naming and Directory Interface

LAN: Local Area Network

LDAP: Light Directory Access Protocol

LPDP: Local Policy Decision Point

LRU: Least Recently Used

LSP: Label Switched Path

MA: Mobile Agents

MD: Mediation Device

MF: Mediation Function

MI: Management Instance

MIB: Management Information Base

MIF: Management Information Format

MPLS: Multiprotocol Label Switching

NACK: Not Acknowledged

NE: Network Element

NEF: Network Element Function

NIP: Network Infrastructure Provider

NM: Network Management

NMF: Network Management Forum

NMS: Network Management System

ODBC: Open Database Connectivity

OMG: Object Management Group

ORB: Object Request Broker

OSD: Open Software Description

OSF: Operating System Function

PBANEM: Policy-based Active Network Element Management

PBANM: Policy-based Active Network Management

PBM: Policy-based Management

PBN: Policy-based Networking

PBNM: Policy-based Network Management

PBVPN: Policy-based Virtual Private Network

PCIM: Policy Core Information Model

PCIMe: Policy Core Information Model extensions

PDP: Policy Decision Point

PEP: Policy Enforcement Point

PHB: Per-hop Behaviour

PIB: Policy Information Base

QAF: Q Adaptor Function

QoS: Quality of Service

QPIM: QoS Policy Information Model

RAP: Resource Allocation Protocol

RCF: Resource Control Framework

RDBMS: Relational Database Management System

RSVP: Resource Reservation Protocol

SC: Security Context

SID: Security ID

SLA: Service Level Agreement

SML: Service Management Layer

SNMP: Simple Network Management Protocol

SP: Service Provider

SPPI: Structure of Policy Provisioning Information

SQL: Structured Query Language

SSL: Secure Sockets Layer

TCA: Traffic Control Agreement

TINA: Telecommunications Information Networking Architecture

TMF: Telecommunications Management Forum

TMN: Telecommunications Management Network

TOM: Telecom Operations Map

ToS: Type of Service

TTCN: Tree and Tabular Combined Notation

UML: Unified Modelling Language

VE: Virtual Environment

VPN: Virtual Private Network

WAN: Wide Area Network

WSF: Workstation Function

XML: Extensible Markup Language

10 REFERENCES

[1] Yechiam Yemini, Germán Goldszmidt, and Shaula Yemini. Network Management by Delegation. In The Second International Symposium on Integrated. Network Management, Washington, DC, April 1991.

[2] N.Damianou, N. Dulay, E. Lupu, M. Sloman, “The Ponder Policy Specification Language”.

[3] “Principles for a Telecommunications Management Network”, ITU-T Recommendation M.3010.

[4] FAIN Internal document, “PBNM architecture proposal” WP4-HEL-032-PBNM-ARCH-Int-v0.2

[5] P.Martinez, M. Brunner, J.Quittek, F. Strauss, J, Schönwalder, S.Mertens, T. Klie, “Using the Script MIB for Policy-based Configuration Management” IEEE/IFIP Network Operations and Management Symposium 2002

[6] Steve Jackowski, “Bury Policy Management!”, Deterministic Networks (September 1999) - http://www.deterministicnetworks.com/burypolicy.html

[7] K.L.E. Law, A. Saxena, "UPM: Unified Policy-Based Network Management," in Proc. SPIE ITCom 2001, Vol.4523, pp.326-337, Denver, August, 2001 - http://www.comm.toronto.edu/~eddie/Papers/upm_spie_itcom2001.pdf

[8] FAIN Deliverable 3 “Initial Specification of Case Study Systems”, May 2001 – http://www.ist-fain.org

[9] FAIN Deliverable 1 “Requirements Analysis and Overall AN Architecture”, May 2001 – http://www.ist-fain.org

[10] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, A. Sastry, " The COPS (Common Open Policy Service) Protocol", IETF RCF2748, January 2000

[11] Emil Lupu, Morris Sloman, “ Conflicts in Policy-based Distributed Systems Management”, IEEE Transactions, November 1999

[12] IETF “Policy Core Information Model -- Version 1 Specification”, RFC3060, Feb 2001

[13] Internal FAIN document, WP4-UPC-003-R14-Int, “Management Components Design”

[14] 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

[15] Matthias Bossardt, Lukas Ruf, Rolf Stadler, Bernhard Plattner: Service Deployment on High Performance Active Network Nodes. IEEE Network Operations and Management Symposium (NOMS 2002), Florence, Italy, April 2002.

[16] FAIN Document: WP4-ETH-001-I16-Int.doc

[17] M. Bossardt, R. Stadler. Service Deployment on High Performance Active Network Nodes. TIK-Report 122, Swiss Federal Institut of Technology (ETH), Zurich, Switzerla nd, September 2001