Chapter 5

V. PROOF OF CONCEPTS IMPLEMENTATION DESCRIPTION

Section V.1 - Introduction

Hereafter we proceed to the description of the classes implemented to fulfil the management functionality conceived for the MANBoP architecture. The implementation description is structured in sections that represent the main steps taken by the MANBoP instances during its normal behaviour. More information about the implemented classes is available in the code itself in the form of Javadoc documentation [SunJAVAa]. This documentation can be found at [MANBoP].

Nevertheless, before proceeding to the code description we will briefly review the naming convention followed in the thesis and the Information Model used.

Section V.2 – Naming Convention

Describing the naming convention followed during the implementation is helpful to ease the understanding of the code and its structure.

The naming convention followed deals basically with four issues: the MANBoP packages naming convention, the database directory naming convention, the Naming Service registration convention and finally the dynamically installable files naming convention. These four issues are described in this section.

It is important noting, before proceeding to the naming convention description, that for classes and attributes we have followed the JAVA naming recommendations [SunJAVAc].

1st MANBoP packages naming convention

The functionality implemented for the different MANBoP components has been grouped under the corresponding JAVA package. We have followed a concrete naming convention to group these functionalities into JAVA packages and for naming these packages.

In the figure below the equivalent folder structure to the JAVA packages used in MANBoP is shown.

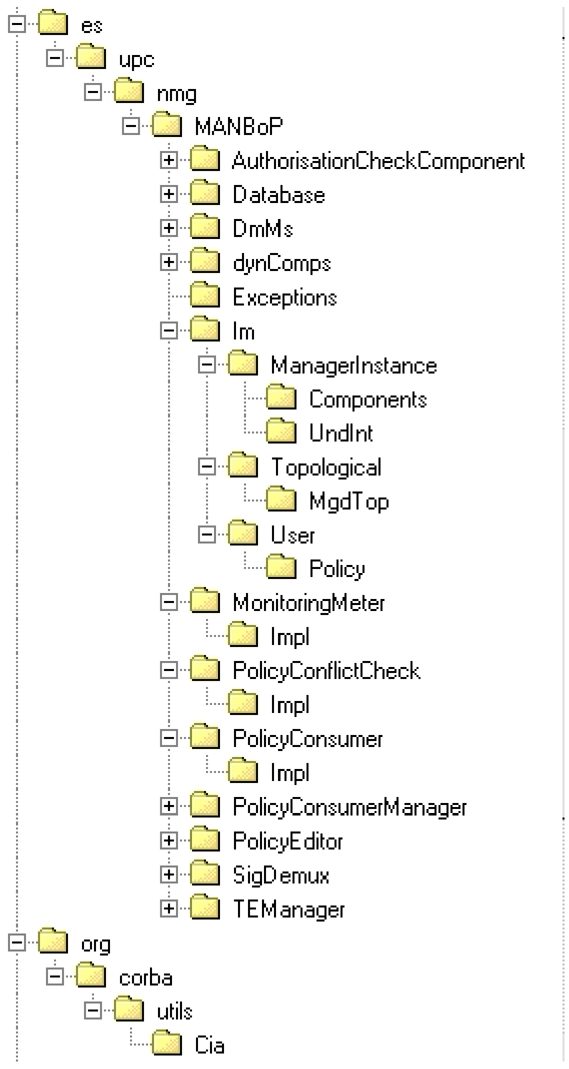

Figure 5 - 1. MANBoP Packages structure

All packages containing the definition of the Information Model Objects are contained within the *'es.upc.nmg.MANBoP.IM'* package.

The IMOs containing topological information (e.g. Node, Link) are grouped under the Topological package. In particular, within the Topological package we can find the managed topology IMOs grouped under the MgdTop package.

Those IMOs containing manager information, as the ManagerInstance object and the Device object, are grouped under the ManagerInstance package. This package also contains all IMOs representing components dynamically installed in the system under the Components package and the IMOs with underlying devices information are grouped under the UndInt package.

Finally, those objects containing user information are stored under the User package. Inside the User package, the Policy package contains all policy-related user information.

All packages containing MANBoP functionality are contained within the '*es.upc.nmg.MANBoP*' package.

Inside this package, all classes implementing functionality from a particular MANBoP component pertain to the same package, which is named with the component name. These are AuthorisationCheckComponent, Database, DmMs, PolicyEditor PolicyConsumerManager, PolicyConflictCheck, TEManager, SigDemux, MonitoringMeter and PolicyConsumer.

The implemented exceptions are used by different MANBoP components. Hence, the programmed exceptions have been grouped into a separate package named Exceptions also contained within the '*es.upc.nmg.MANBoP*' package.

Finally, in addition to the explicit MANBoP functionality some generic programming utilities, used to speed up the code creation process, have also been implemented. These utilities have been grouped under a package named '*org.corba.utils*'. Some of these utilities created are: the BasicObject class, which offers simple methods to realise the most common CORBA-related tasks, and the XPolicySender class, which simulates a higher-level application that introduces a policy into a MANBoP instance. More information related with this helping implemented tools can be found at Appendix D.

Also contained inside the '*org.corba.utils*' package is the CIA package. The CIA package contains all functionality related with the Code Installing Application utility developed as complement to the MANBoP system.

2nd Database directory naming convention

All Information Model Objects implemented in MANBoP are stored within the Database. The Database implementation chosen is the simplest one, based on serialising the objects into files saved at a particular path.

These directories and file names should follow a naming convention. In this section, we are going to describe this convention.

The root of the database directory is on the ($MANBoP)[21]/Database/Root path. Under this directory, the whole Information Model is stored following the hierarchy shown in the figure below.

[21] ($MANBoP) is a local environment variable pointing to the directory where the MANBoP package files are located. This variable is obtained at bootstrap from the manbop.props file.

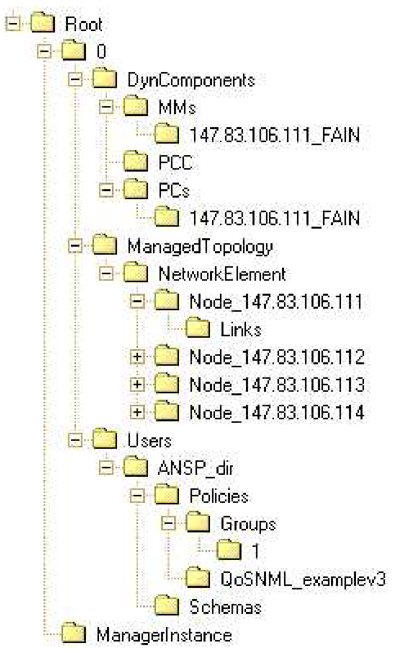

Figure 5 - 2. Database directory structure

As can be seen in the figure, under the Root directory there is the ManagerInstance directory and directories numbered starting from zero. The Manager instance directory contains information about the MANBoP instances running in this machine. In particular, the ManagerInstance objects for all MANBoP instances are stored in this directory.

The numbered directories contain information related with each MANBoP instance running in this machine. Indeed, the number designating each directory stands for the MANBoP instance identifier.

One of the directories storing MANBoP instance-dependent information is the ManagedTopology directory. This directory contains the NetworkElement directory from which a number of directories containing managed nodes information hang. In particular, outgoing link information (under the *'Links'* subdirectory), as well as total and used resources. These node directories are named as: *'Node_<node_id>'*. The *node_id* is a string with the IP address of the node. This simplifies the process of obtaining node information during policy processing.

Inside the NetworkElement directory, the Node and Device objects are stored. Node objects are stored in files named *'<node_id>'* and Device objects in files named *'<device_id>.iface'*.

Inside each node directory, we can find NResources and UNResources objects for that node. They are stored in files named, respectively, *'<node_id>_reso.out'* and *'<node_id>_ureso.out'*.

Finally, inside the Links directory the outgoing Link objects for that node are stored in files named: *'Link_<link_id>.out'*.

Another directory hanging from the numbered directories containing MANBoP instance-dependent information is the Users directory. This directory contains user-related information. More specifically, it stores User Information Model Objects and contains directories with other user-related information. The User IMOs are named with the username of the user they are representing, while the user directories have a *'_dir'* string added to the username (i.e. *'<username>_dir'*).

Each *'<username>_dir'* directory might contain a *Policies* directory containing information related with policies introduced by this user and a *Schemas* directory with the XML Schema files and Schema IMOs named as *<domainId>*. In particular, this *Policies* directory contains some *<domainId>* directories and might also contain a *Groups* directory.

The *<domainId>* directories contain policy-related IMOs pertaining to the policy functional domain after which the directory is named. In particular, the Information Model Objects that might be stored in these directories are the policy IMO itself, the XML policy serialised and the PRI (Policy Resource Information) Information Model Object. These IMOs are named respectively '*<policyconditions><policy rule name>_<sequence number>*[22]*.jav*', and the same with the extensions *'.xml'* and '*.pri*'. *<policyconditions>* are the names of all simple conditions in the policy ordered alphabetically. This information is included in the name to ease the policy conflict check class (by finding policies with similar conditions you find some potentially conflicting policies).

The Groups directory might contain a list of numbered directories. The number of each directory stands for the policy group number of the policy group whose information is contained within. Each one of these numbered directories contains: a Group Information Model Object, the policyId IMO, the credential IMO and policies (both the Policy Information Model and its XML counterpart). The policyId, credential and policies IMO are stored under names that stand for its position identifier within the policy group, they just differ in the extensions, which are respectively: *'.pid'*, *'.cred'*, *'.jav'* and *'.xml'*. The position identifier notion will be explored later on the Information Model section. The Group IMO is stored with the same name as the directory where it is contained (i.e. the policy group number).

Finally, the last directory hanging from the numbered directories containing MANBoP instance-dependent information is the DynComponents directory. This directory might contain a PCC directory, PCs directory and a MMs directory. Each of these directories will contain IMOs representing the dynamic components of each class installed within the system. In particular,

[22] The meaning of these fields is described in more detail in the Information Model chapter (i.e. in the Policy information model description).

the PCC directory might contain a PCC IMO named as *'pcc'*. The PCs directory might have many directories named *'<nodeSetId>'* each of them containing PC IMOs named *'<PCId>'* and representing the PC components implementing functionality for that nodeSet that have been installed within the system. In a similar way, the MMs directory might also contain many directories named *'<nodeSetId>'* each of them containing MM IMOs named *'<MMId>'* and representing MM components implementing functionality for that nodeSet that have been installed within the system.

3rd Naming Service registration naming convention

All MANBoP components are started as CORBA components and registered in the Naming Service. Hence, they should follow a particular naming convention so that they can be easily registered and retrieved from the Naming Service. Furthermore, since different MANBoP instances might be using the same Naming Service we should define a naming convention to avoid name clashes between the same components at different instances.

All classes that form a MANBoP component and that offer one of the component interfaces must be registered in the Naming Service. They are registered following a four-field naming convention as:

'<host_addr>_<MANBoPInstanceId>_<component_acronym>_<class_name>'

The first part of the naming convention is: *<host_addr>*. This part identifies the machine where the current MANBoP system is running by its IP address.

The second field is *<MANBoPInstanceId>*. It is used to distinguish between components from different MANBoP instances running in the same host. This field contains the identifier of the current MANBoP instance. This identifier is stored as a public static field of the PCMCoreImpl class. The value of this field is obtained at bootstrap by the DBCoreImpl class based on the number of ManagerInstance objects found in the database. Since there is one ManagerInstance object per MANBoP instance, the new MANBoP instance identifier is just the number of ManagerInstance objects stored plus one.

The third field is the component acronym that identifies the component to which the registered object pertains. The acronyms used are: PCM for the PolicyConsumerManager, ACC for the Authorisation Check Component, SD for the SigDemux component, TEM for the TEManager, DmMs for the Decision-making Monitoring system, PE for the PolicyEditor, DB for the Database, PC for the Policy Consumers, MM for the Monitoring Meters and PCC for the PolicyConflictCheck component.

The last field is the class name whose instance is being registered in the Naming Service.

An exception of the above-described rule applies to the dynamically installed components, in particular to the PC and MM components. These components are registered in the Naming Service as follows:

'<host_addr>_<MANBoPInstanceId>_<component_acronym>_<class_name>_<nodeSetId>_<codeId>'

The first four elements of the naming convention are exactly those described for the general rule. The *<nodeSetId>* field uniquely describes the nodeSet being managed by this PC or MM. The *<nodeSetId>* field is directly obtained from the UndInt file (see appendix B) at system bootstrap and stored in the corresponding Device IMOs.

The *<codeId>* field will be described in more detail in the following sub-section. It stands for the name of the code file that contains the component functionality.

Finally, not only MANBoP system components of the current instance are registered in the Naming Service. Also higher-level entities might be registered to receive enforcement or monitoring results as well as Code Installing Application Services.

The higher-level entities might be higher-level Policy Consumers waiting for the enforcement result of a policy they've sent, Monitoring Meters or generic receivers (which must implement the org.corba.utils.GenRecv interface). These three types of entities register in the Naming Service following different naming conventions.

The naming convention for higher-level PCs is:

'<host_addr>_<MANBoPInstanceId>_HLMANBoP_<hlpcId>'

Where the *<hlpcId>* field stands for the higher-level Policy Consumer identifier value. This value is also included in the received XML policies (see appendix C).

For MMs the naming convention is almost the same:

'<host_addr>_<MANBoPInstanceId>_HLMANBoP_<hlmmId>'

The *<hlmmId>* field stands for the higher-level Monitoring Meter identifier.

The naming convention for generic receivers is:

'<host_addr>_<MANBoPInstanceId>_GenRecv_<number>'

The generic receivers as they register at the Naming Service will be assigned a sequentially incremented number.

In relation to the Code Installing Application (CIA) services, there is one CIA client registered in the Naming Service per machine where code corresponding to any of the MANBoP instances running in this station might be installed. For example, if we have in this station two MANBoP instances that might install code, in addition to in the station itself, two and three machines respectively, there will be a total of six CIA clients registered in the Naming Service: one for the station itself and five for the managed machines. The naming convention for the registration of these CIA clients is as follows:

'<host_addr>_CIA_CIAImpl'

Where *<host_addr>* stands for the IP address of the machine where the CIA client component (i.e. CIAImpl) is running.

4th Dynamically installable files naming convention

By dynamically installable files we refer here to all those files installed on run-time to upgrade the MANBoP system with a new component (i.e. PC, MM or PCC). We must differentiate between two types of such files: .class or .jar files and XML Schema files. The former are downloaded to add new functionality (i.e. new policy functional domains) to the system while the latter are dynamically downloaded to check the rights of a user who is trying to introduce a policy in the system.

In relation to code files, we need a naming convention both for the name of the file where the code is included and for the JAVA name of the loaded classes. The later one is basic for avoid collisions when these classes are loaded in the JVM.

The naming convention for code files (i.e. *.class* and *.jar* files) is as follows: *<code name>_<management topology id>_<interface id>*

The name of the code file following this naming convention is called code identifier (codeId).

The code name is the name of the component to be downloaded as comes specified in the policy (i.e. PCId or MMId). The name identifier might include as well the component version.

The management topology identifier is taken from the MANBoP instance itself. It defines the management level functionality implemented by the component as well as the expected underlying devices (e.g. network level functionality over element level managers).

The interface identifier defines the type of underlying devices that the component can manage and thus defines how the component interacts with the underlying device. Some possible values of the interface identifier can be, for example, MANBoP to indicate that the component is implemented to work over lower-level MANBoP instances, or FAIN to indicate that the component has been implemented to work over FAIN active nodes. The interface value can be any but, at least, there must be one component supporting each type of device included in the managed infrastructure.

In relation to the naming convention for the JAVA name of loaded classes, this name must have the following structure to avoid collisions:

es.upc.nmg.MANBoP.<*Component_name*>.Impl.<*codeId*>_<*class_name*>

The component name is, obviously, the name of the component of which an implementation is being dynamically installed (i.e. MonitoringMeter, PolicyConsumer, PolicyConflictCheck).

The codeId is the one already described in previous paragraphs.

The class name stands for the name of the component class that is being loaded in the JVM. For example, in the case of the Policy Consumer component one of its name classes is the MapperImpl class.

The naming convention for XML Schema files depends on the user that tries to introduce the policy. When the user has full access rights to the policy functional domain he is trying to access the XML Schema file will be name simply as: *<domainId>.xsd*

The domain identifier is simply extracted from the XML Schema name included in the XML policy. Hence, when the user has full access rights the XML Schema file assigned to that user will be simply the name of the XML Schema coming with the XML policy. However, in those cases when the user does not have full access rights, the user policy must be checked against a special constraint XML Schema. The name of this schema is stored within a Schema IMO stored in the DB and bind both with the domain identifier and the user name.

Section V.3 – Information Model

The Information Model is an essential part of every management system. MANBoP is not an exception. The Information Model designed for MANBoP can be divided in two big groups: policies and Information Model Objects.

The policy part dictates not only the appropriate syntax and semantics of policies but also the capabilities of those policies in terms of actions that can be enforced and conditions that should be monitored. Since, MANBoP is a dynamically extensible architecture the potential number of actions and conditions supported is infinite. Nevertheless, in this section we will just describe those that have been implemented for this proof-of-concepts. Although the Policy objects are as well Information Model Objects their relevance and importance within the architecture has moved us to explain them in a separate section.

Information Model Objects (IMOs) provide information to support each MANBoP system in developing its functionality smoothly. This information is divided in three main groups, user information, managed topology information and MANBoP architecture information. While the first two are used to take proper decisions and realise Call Admission Control (CAC), that is, to decide whether policy requests can be accepted, the third one is used to decide on the most adequate components to extend the architecture under a particular situation.

Hereafter, we are going to provide insights to the all Information Model Objects.

1st Policy Information Model

In MANBoP, policies are expressed and transmitted in XML. The concrete syntax and basic XML Schemas for this syntax are extensively described in appendix C. In this section we will just focus in the IMO representing a policy and the other objects that form it. The Policy IMO is obtained after parsing the corresponding XML policy into a Java object.

The policy object fields are all domain-independent, so that the system can parse all policies without the need to understand what they are aimed for. The processing of domain-dependant parts will be realised by the dynamically extensible components of the architecture (i.e. Policy Consumers, Monitoring Meters, etc.). To achieve this, some policy fields are designed in a generic way to allow any domain-dependent value.

In the following sub-sections, we first explore the Policy Information Model domain-independent fields and then, we explain some domain-dependant values we have implemented for this proof-of-concepts.

A Policy Core Information Model

The policy structure used in MANBoP is based on the IETF Policy Core Information Model [IETF] though simplified by defining as mandatory only those features essential for policy processing. Hence, the size of policies is considerably reduced (around five times smaller than the policy size following the PCIM model) and their processing is simpler.

Conceptually speaking, MANBoP policies can be initially divided into delegation (or authorisation) policies and obligation policies. Delegation policies are generally used to specify who is allowed to access to certain functionality and how it is allowed to do it. They can be further subdivided into delegation of management responsibility policies and delegation of access rights policies. The first ones cause, when enforced, the creation of restricted XML Schemas for a user, so that they can manage their resources using the operators' infrastructure in a controlled way. The last ones configure the security components of managed devices (i.e., active and programmable routers) so that users are allowed to realise certain actions directly over device interfaces. As an example, delegation of access rights policies would be used to specify the node OS functionality accessible to an active service running inside an execution environment of an active node. Hence, through the setting of the appropriate delegation of access rights policies a user can be potentially allowed to manage its resources with its own code. The user management code either could reside in a separate station or even be installed in execution environments of active or programmable routers.

Obligation policies specify actions that must be enforced over managed devices when specific events occur. Obligation policies can also be further subdivided into many types of policies, such as QoS policies, fault management policies, monitoring policies, etc.

All MANBoP policies follow the structure shown in Table 5 - 1. The policy rule consists of eleven fields.

Attribute Name	*IDL Type*	*Description*
schemaId	string	Name of the schema linked with this policy.
ruleId	t_policyruleId	Uniquely identifies the policy within the management infrastructure.
status	long	Integer containing the policy processing status. Possible values are: 2 being introduced, 1 not enforced, 0 enforced
roles	sequence<string>	Identifies the Roles to which the policy applies.
user	t_userInfo	Contains the identifier of the user that is introducing the policy in the framework.
validity	t_prvp	Includes the policy expiration date.
group	t_pgroup	Used for the correct processing of policy sets.
evaluation	t_eval	Structure containing information for the correct evaluation of policy conditions.
act	ActEnf	Structure containing further information for the correct processing of policy actions
conds	t_crefList	The *Conditions* element includes all policy conditions. Conditions can be either compound or simple and refer to an hour of the day, an IP flow, a concrete notification or a managed device status. The modules needed to monitor these conditions, if any, are also extracted from the *Conditions* element information. This element is optional; when not included, the framework interprets that the policy action should be enforced directly.
actions	t_arefList	The *Actions* element contains the action type and parameters as well as information about the module responsible of enforcing this action. At least one *Actions* element is mandatory in all policies but there can be more that one.

Table 5 - 1. MANBoP policy information fields

First, the *schemaId* string contains the name of the XML Schema that should be used to check the correct syntax of the policy as long as the user has full access to that functional domain. A functional domain is represented in MANBoP as an XML Schema that determines the allowed fields and field values. In case the user has restricted access to that functional domain the *schemaId* attribute is ignored since a special, restricted, XML Schema will be used to check user's policies From the *schemaId* attribute the policy functional domain identifier, *domainId*, is obtained by removing the schema extension *'.xsd'* from the *schemaId.*

The ruleId field uniquely identifies the policy within the management infrastructure. The rule information is enclosed within a t_policyruleId object formed by two strings and an integer. The strings contain respectively the policy rule name that uniquely identifies this policy type and the identifier of the higher-level component that sent the policy if any. The integer represents a sequence number used to distinguish between different policies of the same type and sent by the same user.

The *roles* field lists all the roles to which the policy applies. These roles might be used by Policy Consumers to select the managed devices where the policy must be enforced. The field is represented as an array of strings, each string containing one role name.

The *user* field contains information about the user who is introducing the policy in the framework. This information is used to authenticate the user and select the restricted XML Schema against which the user policy should be validated. The user information is enclosed within a *t_userInfo* object. This object consists of two strings, one containing the user name and the other containing the password.

The policy expiration date is contained within the *validity* field. Usually the validity period is given just with the day and hour the policy starts and stops being valid. Nevertheless, filters specifying concrete months, days and hours during which the policy is not valid can be also introduced. The policy rule validity period information is given as a *t_prvp* object. Six member fields form this object. The first one is a *t_period* object formed by four strings representing respectively: the starting day, starting hour, stopping day and stopping hour of the validity period of the policy. The second *t_prvp* field is an optional JAVA short primitive that represents a mask specifying which months of the year the validity period specified is applicable. The third is an optional JAVA long primitive representing a mask that specifies what days of the month the policy is valid. Fourth, an optional byte primitive that specifies the valid days of the week. Fifth, an optional string that establishes the valid day hours. Finally, the sixth field is an optional boolean that specifies if the hours given follow the local time or the UTC time.

The *group* field is used for the correct processing of policy sets or groups. A policy group is a set of policies that should be processed in a particular way, i.e. atomically, sequentially, etc. This adds more flexibility to the specification and deployment of policies and allows better determining the expected behaviour of managed entities. For example, a user (e.g. a service provider) might require several node resources in order to offer an active service to its customers. These resources are reserved using several policies that form a policy group. Such a policy set should be enforced atomically since a single unreserved resource disables the service, thus making unnecessary the reservation of the other resources.

The information contained in the *group* field is enclosed in a *t_pgroup* object. Such object contains five fields, four integers and a *t_order* object. The four integers represent respectively: the policy group number uniquely identifying the policy group for that user, the number of policies forming the group, the execution strategy that must be applied to this group (e.g. sequential, the first possible, atomically, etc.) and the management level at which the policy group must be evaluated. The possible values for this last integer are zero when the policy group must be evaluated at the network-level and one when it must be evaluated at the element-level. Finally, the *t_order* object indicates the order position of that policy within the policy group. More specifically, the *t_order* object contains four fields. The first one is a string that indicates the global position of the policy within the group. The second is an integer indicating the initial position of the policy within the group. By initial position we mean the position that the policy had within the group when it was created. This position might change as the policy is processed by MANBoP instances at different levels since the processing of the policy at these levels might require splitting the policy into several more specialised policies. The third field within the *t_order* object is indeed, an integer that indicates the number of times that the policy has been subdivided along higher-level MANBoP instances. The last field, is an array of *t_split* objects which are formed by two integers indicating respectively the partial position of a policy among the policies generated after a sub-division, and the total number of policies generated by this sub-division. There is one *t_split* object in the array per sub-division made to the policy before arriving to the current MANBoP instance.

The *evaluation* field contains information concerning policy conditions. More specifically, it contains two fields, a boolean and an integer. The boolean indicates whether the different policy conditions follow a Conjunctive Normal Form ("true") or a Disjunctive Normal Form ("false") [Weisstein]. The integer field specifies whether policy conditions must be evaluated just at the network-level (0), at the element-level (1) or at all management levels (2).

The *act* field contains information that affects the way policy actions should be processed. This field is contained within a *t_actEnf* structure. This structure is formed by two integers, one boolean and an array of strings. The array of strings contains the identifiers of the managed nodes where the policy actions must be enforced. The two integers represent respectively what is the aim of the policy actions (i.e. it specifies if the configuration must be: 0 created, 1 activated, 2 modified, 3 deactivated or 4 removed) and how policy actions must be enforced on the target nodes, either in a best effort way or in a guaranteed way (if the policy is not correctly enforced in ALL target nodes the enforcement is not considered successful and hence the policy is uninstalled from all nodes where it was enforced). Finally, the boolean establishes whether the policy action is oriented towards configuring the managed device ("false") or towards configuring the management station ("true"). This helps the PCCnt class from the PCM component to decide whether the Policy Consumer component that must enforce the policy should be installed (if not already done) at the managed topology, when possible, or at the management station. In this way, Policy Consumer components configuring the management station, as the Delegation Policy Consumer, are always installed at the management station and not at the managed device, which would be nonsense.

The *conds* field includes all policy conditions. Conditions can be either compound or simple and refer to an hour of the day, an IP flow, a concrete notification or a managed device status. The monitoring meters needed to monitor these conditions, if any, are also extracted from the *conds* field information. This field is optional; when not included, the framework interprets that the policy action should be enforced directly. The field type is an array of *t_condRef* objects. This object contains only one array of simple conditions represented as *t_simpleCond* objects, and another array of compound conditions represented as *t_comCond* objects. This is a way to exemplify that policy conditions can be either simple conditions or compound conditions.

Compound conditions are, at they turn, formed by other simple or compound conditions. Hence, the *t_comCond* object also contains arrays of *t_simpleCond* and *t_comCond* objects. In addition, the *t_comCond* object also contains five more fields. These fields are a string with the policy condition name, an integer identifying the group of conditions to which this condition pertains, a boolean that specifies if the condition is negated, an array of strings, each one containing the identifier of a monitoring meter needed to monitor these policy conditions and finally, a boolean that indicates whether packets that mirror the specified filter are to be treated as matching the filter.

To conclude the description of the *conds* field information, the *t_simpleCond* object consists of ten fields. These ten fields can be split in six domain-independent fields and four domain dependent fields. The domain-independent fields are: a string with the policy condition name; an integer specifying the condition group number; a boolean indicating whether the whole condition is negated; an identifier of the Monitoring Meter component that might be needed to monitor this simple condition; an array of strings each one containing the identifier of a node that must be monitored to evaluate this simple condition; and finally, a boolean that determines whether all monitoring nodes must evaluate to true so that the condition is true or instead, just one node evaluating to true is enough.

The four fields containing domain-dependant condition information are four strings. The first one stands for the name of the data to be monitored. The second one indicates the data type to be monitored. The fourth one is the data value that sets a threshold or a filter in the condition. Finally, the third one expresses how the data must be evaluated, that is, if the condition will be considered as match only when the data value (i.e. the fourth string) matches exactly with the data monitored, or when the data monitored is higher than the data value, or lower, etc…

The last field in the MANBoP's Policy IMO is the *actions* field that contains information about all policy actions. At least one policy action is mandatory in all policies but there can be more that one. The action information is represented as a *t_actRef* object. Such object consists of five fields. These fields are: a string with the XML type of action; a string with the concrete name of the action; another string that identifies the Policy Consumer component capable of processing this policy action; and finally, the domain-dependant policy action information. This information is expressed with two arrays of strings: one containing the name of the managed device attributes to be modified, and the second one containing the new values of these attributes.

All this information is mapped in the implementation into a JAVA class. The Policy class is a final class (not modifiable) with eleven fields and a constructor.

The constructor of the mapped JAVA Policy class simply initiates the instance fields with the values received as parameters. The constructor signature is: *public Policy(String, t_policyruleId, int, String[], t_userInfo, t_prvp, t_pgroup, t_eval, actEnf, t_condRef[], t_actRef[])*

The Policy JAVA class is included within the *es.upc.nmg.MANBoP.IM. User.Policy* JAVA package.

Other objects that are also part of the Policy Information Model although not included in the Policy IM are the *t_policyId* and *PRI* objects.

The *t_policyId* IMO is used to uniquely identify a Policy within the system so that it can be easily stored and retrieved from the Database. The *t_policyId* consists of four fields: three strings and a *t_policyruleId* field already described. The three strings express respectively, the identifier of the policy functional domain to which the policy pertains, the user name of the user that owns the policy and a concatenation of the simple conditions names in alphabetic order. This last field is used to ease the policy conflict check functionality. In particular the process of finding potentially conflicting policies.

The *PRI* IMO (Policy Resource Information) contains the resources that have been reserved or configured by the enforcement of a policy. This information is used when the policy must be de-installed to remove these reservations or configurations. In the current version of the proof-of-concepts implementation this class only contains one field. This field is an array of strings containing the identifiers of the managed nodes where the policy has been enforced. In future version, more fine-grained information will be included in this object.

B Implemented domains

Along this sub-section we are going to describe the information model parts related with functional domains implemented for the proof-of-concepts. These parts are mainly the condition and action field values understood by Monitoring Meter and Policy Consumer components that have been implemented.

Several conditions have been specified within the policies used in the proof-of-concepts scenarios described in the last section of this chapter.

Nevertheless, just one of these conditions need to be monitored. The remaining ones provide information to identify either a flow or a user.

In the following sub-sections we will describe all these condition and action field values.

a VANSitesInfo condition

The first condition we are going to describe is a compound condition composed by one or more simple conditions. This condition is used to identify the service provider's sites that must be interconnected when creating his VAN. However, in the current proof-of-concepts implementation this information is finally not taken into account.

In the table below we summarise the field values for this condition.

Compound condition fields values	*Simple condition fields values*
• Policy condition name: 'VANSitesInfo' • Group number: by default '1' • Condition negated: 'false' • The array of monitoring meter identifiers is empty • Mirrored field: 'false' • The array of compound conditions is empty • The array of simple conditions has one or more t_simpleCond objects	• Policy condition name: 'VANSiteInfo' • Group number: the same as the compound condition where it is contained • Condition negated: 'false' • Monitoring meter identifier: 'null' • The array of nodes to be monitored is empty • Need all nodes evaluate to true?: 'true' • Name of the data to be monitored: 'IPAddr' • Type of the data to be monitored: 'IPv4Addr' • Evaluation method: 'Match' • Condition Value: ip address of one of the service provider's sites

Table 5 - 2. VANSitesInfo condition values

When the management infrastructure contains element-level managers, the *VANSitesInfo* condition is translated into several *VANFlowCond* compound conditions. Indeed, there are as much *VANFlowCond* as couple of service provider's sites to be interconnected.

The table below summarises the values of the *VANFlowCond* fields.

Compound condition fields values	*Simple condition fields values*
• Policy condition name: 'VANFlowCond' • Group number: the same as the VANSitesInfo condition from which it derives • Condition negated: 'false' • The array of monitoring meter identifiers is empty • Mirrored field: 'true' • The array of compound conditions is empty • The array of simple conditions has two t_simpleCond objects	• Policy condition name: 'SourceFlowCond' • Group number: the same as the compound condition where it is contained • Condition negated: 'false' • Monitoring meter identifier: 'null' • The array of nodes to be monitored is empty • Need all nodes evaluate to true?: 'true' • Name of the data to be monitored: 'IPSource' • Type of the data to be monitored: 'IPv4Addr' • Evaluation method: 'Match' • Condition Value: ip address of the site acting as source for this flow
	• Policy condition name: 'DestFlowCond' • Group number: the same as the compound condition where it is contained • Condition negated: 'false' • Monitoring meter identifier: 'null' • The array of nodes to be monitored is empty • Need all nodes evaluate to true?: 'true' • Name of the data to be monitored: 'IPDest' • Type of the data to be monitored: 'IPv4Addr' • Evaluation method: 'Match' • Condition Value: ip address of the site acting as destination for this flow

Table 5 - 3. VANFlowCond condition values

b UserCredential condition

The UserCredential condition is a compound condition made of two simple conditions. The first one containing a user name and the second one containing the password for that user name. This condition is used in the proof-of-concepts to establish to which service provider the management functionality must be delegated.

The table below summarises the values of the *UserCredential* fields.

Compound condition fields values	*Simple condition fields values*
• Policy condition name: 'UserCredential' • Group number: by default '1' • Condition negated: 'false' • The array of monitoring meter identifiers is empty • Mirrored field: 'false' • The array of compound conditions is empty • The array of simple conditions has two t_simpleCond objects	• Policy condition name: 'Username' • Group number: the same as the compound condition where it is contained • Condition negated: 'false' • Monitoring meter identifier: 'null' • The array of nodes to be monitored is empty • Need all nodes evaluate to true?: 'true' • Name of the data to be monitored: 'Username' • Type of the data to be monitored: 'string' • Evaluation method: 'Match' • Condition Value: username of the service provider to which management functionality is being delegated
	• Policy condition name: 'Password' • Group number: the same as the compound condition where it is contained • Condition negated: 'false' • Monitoring meter identifier: 'null' • The array of nodes to be monitored is empty • Need all nodes evaluate to true?: 'true' • Name of the data to be monitored: 'Password' • Type of the data to be monitored: 'string' • Evaluation method: 'Match' • Condition Value: password of the service provider to which management functionality is being delegated

Table 5 - 4. UserCredential condition values

This condition is kept with the same values for the element-level policy.

c IFBWCond condition

This one is the only condition that needs to be monitored. The *IFBWCond* is a simple condition. It is used to monitor the bandwidth used in a router's interface. The condition establishes a threshold that, when reached, triggers the enforcement of the policy. The actual monitoring of this condition is done, in the proof-of-concepts, by the BWMM Monitoring Meter component that will be described later on the document.

The following table shows the condition values used in the proof-of-concepts implementation.

Simple condition fields values
• Policy condition name: 'IFBWCond' • Group number: by default '1' • Condition negated: 'false' • Monitoring meter identifier: 'BWMM' • The array of nodes to be monitored: the ip addresses of nodes that must be monitored • Need all nodes evaluate to true?: 'false' • Name of the data to be monitored: 'LinkUsedBW' • Type of the data to be monitored: 'integer' • Evaluation method: 'MoreThan' • Condition Value: ip address of the router being monitored, type of router, interface to be monitored and threshold value

Table 5 - 5. IFBWCond values

When the management infrastructure contains element-level managers. The BWMM working at the network-level, to monitor the condition, creates a monitoring policy that is sent to the appropriate element-level managers. This monitoring policy contains exactly the same condition values as the network-level one while the policy action indicates which network-level Monitoring Meter component must be warned when the condition is met.

d QoSAlloc action

The *QoSAlloc* action is used to allocate QoS resources to a service provider. In the proof-of-concepts implementation it is used for creating and activating the VAN to the service provider. The action is enforced by the QoSPC Policy Consumer component.

The table below includes the action values that have been used. The table contains three columns. The first one lists the values of those fields that are independent of the functional domain. The second column lists the fields included within the array of domain-dependant field names. Finally, the third column includes the values of these fields, which are included within the domain-dependant field values array.

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: QoSAlloc • Action name: NLAlloc • Policy consumer identifier: QoSPC	0. VNId 1. QoSClass 2. CompQoSClass 3. EE	0. wtv 1. gold 2. silver 3. JVM

Table 5 - 6. QoSAlloc action values

The *VNId* field stands for the unique identifier of the Virtual Active Network that encloses all QoS resources reserved to that service provider.

QoSClass stands for the class of forwarding quality of service assigned to the service provider while *CompQoSClass* stands for the computing class of quality of service. The possible values for both classes are bronze, silver and gold.

Finally, *EE* stands for the name of the Execution Environment to which the computing resources are assigned.

The values listed for these fields in the third column are those used at the proof-of-concepts demonstration.

When the management infrastructure includes element-level managers this policy action is translated into an element-level policy action of the same type (*QoSAlloc)* where quality of service information is more detailed. Nonetheless, for this proof-of-concepts implementation we have just included in the element-level action only those fields that are really taken into account by the managed devices.

In the table below we list the element-level action values.

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: QoSAlloc • Action name: ELAlloc • Policy consumer identifier: QoSPC	0. VNId 1. QoSParameters 2. TrafficProfile 3. EE	0. wtv 1. 1 2. 2000 3. JVM

Table 5 - 7. Element-level QoSAlloc action values

Among those domain-dependant field names that have not been yet explained, the *QoSParameters* field contains a numeric representation of the level of quality of service allocated, where 1 is the second better. The *TrafficProfile* field represents the average throughput permitted in kilobytes.

e newUser action

This action is used to register a new user within the management system. The action might even grant access to certain restricted domains to the user. The network operator must appropriately create these restricted functional domains. The *newUser* policy action must be enforced by the DelegationPC Policy Consumer component.

The table below includes the action values used in the proof-of-concepts

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: newUser • Action name: newUserDelegation • Policy consumer identifier: DelegationPC	0. User 1. Password 2. Services 3. Applies	0. wtv 1. wtvPass 2. ServicePC 3. All

Table 5 - 8. newUser action values

The *User* field specifies the name of the user that is going to be registered and the *Password* field his password.

The *Services* field contains the list of functional domains to which the service provider will have restricted access.

Finally, the *Applies* field determines where, within the management infrastructure, the new user must be registered. Possible values for this field are network-level, element-level or all.

This policy action is not translated, at the element-level it contains exactly the same fields as in the network-level.

f FDRestriction action

The Functional Domain Restriction (*FDRestriction)* action is used to create the restricted functional domains for the service provider. More specifically, this action specifies those values the service provider is allowed to introduce in certain fields. The *FDRestriction* policy action must be enforced by the DelegationPC Policy Consumer component.

The table below includes the action values used in the proof-of-concepts

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: FDRestriction • Action name: FunctionalDomainDelegationRestriction • Policy consumer identifier: DelegationPC	0. FDName 1. NodeRestrictions 2. ActRestrictions 3. PolicyRestriction FieldName="VNId" FieldRestrictionType="StringEnumeration" 4. PolicyRestriction FieldName="ServiceName" FieldRestrictionType="StringEnumeration" 5. Applies	0. ServicePC 1. 147.83.106.104 10.0.4.4 2. ServiceDeployment ServiceConfiguration 3. wtv 4. duplicator, transcoder 5. All

Table 5 - 9. FDRestriction action values

FDName stands for the name of the functional domain that is going to be restricted as result of this policy action enforcement.

The *NodeRestrictions* field lists the managed devices where the service provider is allowed to enforce a policy from this domain.

ActRestrictions lists all functional domain policy action names that are allowed to the service provider.

The *PolicyRestriction* is a generic way used for restricting any possible functional domain action value. Zero or more *PolicyRestriction* fields might be included within the policy action. In this case two fields are restricted, the *VNId* field with the only allowed value of wtv and the *ServiceName* field that permits the duplicator and transcoder service names.

Finally, the *Applies* field has exactly the same meaning as for the *newUser* action.

This policy action is not translated, at the element-level it contains exactly the same fields as in the network-level.

g ServiceDeployment

The *ServiceDeployment* action is used to request the deployment of an active service on one or more managed active nodes. The enforcement of this policy action is carried out by the ServicePC Policy Consumer component.

The fields, and their values, included in the policy action are shown in Table 5 - 10.

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: ServiceDeployment • Action name: ServiceDeployment • Policy consumer identifier: ServicePC	0. VNId 1. ServiceName 2. EE	0. wtv 1. duplicator 2. JVM

Table 5 - 10. ServiceDeployment action values

Both the *VNId* and *EE* fields have been described in previous policy actions. In relation to the *ServiceName* field, it contains the name of the active service that must be installed within one or more active or programmable routers.

As in the preceding policy actions, the *ServiceDeployment* action is also kept with the same format in element-level policies.

h ServiceConfiguration

The ServiceConfiguration policy action is used to configure active services installed within the managed network. Obviously, the information contained within these policies is service-specific, for this reason, the action fields are made generic to include such information. Furthermore, active services must include a particular configuration interface to be configured by the MANBoP management system. More specifically, it is the ServicePC component the one that develops the active service configuration. Nevertheless, there might be many types of common active services configuration interfaces and many types of ServicePC components for configuring these interfaces depending on the type of managed active node, the type of EE, etc. In any case, this is a network operator decision according to his business needs.

The values of the *ServiceConfiguration* policy action fields, as they have been used in the proof-of-concepts, are included in the following table.

Common fields	Domain-dependant field names	Domain-dependant field values
• Action type: ServiceConfiguration • Action name: ServiceConfiguration • Policy consumer identifier: ServicePC	0. VNId 1. ServiceName 2. ConfigurationInfo 3. ConfigurationInfo 4. ConfigurationInfo 5. ConfigurationInfo 6. ConfigurationInfo	0. wtv 1. duplicator 2. 147.83.106.111 3. 20000 4. 10.0.4.4 5. 172.31.255.3 6. 16000

Table 5 - 11. ServiceConfiguration action values

The only action field that have not yet been explained is the *ConfigurationInfo* field. The policy action includes one or more *ConfigurationInfo* fields. These fields contain the configuration information that must be introduced to the managed active service.

In the particular case of the policy action values shown, used for configuring the duplicator active service in the proof-of-concepts scenario, the configuration values include the IP address and port of the video source first, and of the two video destinations afterwards.

At the element-level, the translated *ServiceConfiguration* policy action contains exactly the same fields as the network-level ones. However, in particular cases the configuration information might be more detailed at the element-level. This can only happen when the network operator's ServicePC component is aware of the active service and its configuration parameters.

It is worth noting that the network operator might decide to create service-aware Policy Consumer components for concrete active services. In this way, these active services would not need to offer the common configuration interface as the PC component will be aware of the concrete service interface.

i QoSRouteThrough action

This policy action is used to request the modification of a route for one or more flows. More specifically, it establishes the routers within the managed topology that flows must cross. The enforcement of this policy action is done by the QoSPC component.

The fields included within this policy action are shown below.

Common fields	Domain-dependant field names	Domain-dependant field values
• Action type: QoSRouteThrough • Action name: NLRouting • Policy consumer identifier: QoSPC	0. FlowSource 1. FlowDestination 2. Hops	0. – 1. 172.31.255.3 2. 172.31.255.1

Table 5 - 12. QoSRouteThrough action values

The *FlowSource* and *FlowDestination* fields are used to specify the flows that must be re-routed. Further fields containing source and destination ports as well as the protocol must be included in the future.

Finally, the *Hops* field contains the routers through which the flows must be routed.

The values shown in the table are those used in the proof-of-concepts. As can be seen, flows are re-routed based just on their destination.

The *QoSRouteThrough* action is translated as a *QoSNHRouting* at the element-level. The main difference of the element-level action is that it specifies just the next hop for the flow to configure the managed router accordingly.

The table below lists the fields included within the *QoSNHRouting* policy action.

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: QoSNHRouting • Action name: ELNH • Policy consumer identifier: QoSPC	0. NextHop	0. 172.31.255.1

Table 5 - 13. QoSNHRouting action values

The *NextHop* field specifies the next hop for the flows to be re-routed. In the element-level policy the flows are given in the policy conditions. Particularly, they are specified as *VANFlowCond* policy conditions, which have been already described.

j MonReporter action

This policy action is included within the monitoring policy created, in the proof-of-concepts, by the BWMM component to monitor the used bandwidth in a router's interface. The *MonReporter* action specifies the network-level MM component that must be informed when the monitored condition is met.

Table 5 - 14 lists the fields and values for the *MonReporter* policy action.

Common fields	*Domain-dependant field names*	*Domain-dependant field values*
• Action type: MonReporter • Action name: Reporter • Policy consumer identifier: MonPC	0. HLMM	0. BWMM

Table 5 - 14. MonReporter action values

The HLMM field contains the name of the higher-level component to be contacted when the monitored condition changes its value, either because the monitored value meets the condition or because the monitored value is no longer meeting the condition.

2nd Information Model Objects (IMOs)

As already mentioned in the Information Model introductory paragraphs Information Model Objects contain information needed to realise the specified management functions.

The figure below shows a logical representation[23] of the MANBoP Information Model Objects. The diagram includes containment information represented as numbers and inheritance information represented with the keyword <<*refine*>>. The arrow numbers specify how many objects at the tale of the arrow are contained at the object at the head of the arrow. When two points appear between the numbers at one of the arrow edges the one-to-many relationship is then limited within the edges specified with the numbers. The objects at the head of the arrow marked with the <<*refine*>> keyword inherit from the objects at the tale of that arrow.

The Root object acts merely as nexus between the branches, that is, as a way to conceptually unify the three main branches containing the Information Model Objects. No information is associated to the root object and therefore it has not been implemented.

The three branches contain, respectively, user related information, such as policies introduced and access rights in the form of schemas; managed topology information with the nodes, links and resources being managed by the MANBoP instance; and manager-related information describing the components currently running within the MANBoP instance and underlying devices information.

The user-related branch and the managed topology branch are logically linked through the Role UNResources objects. This is justified to represent the fact that users introduce policies requesting resources to nodes. The information contained within the UNResources objects will be described in detail in the following paragraphs.

[23] The diagram follows UML recommendations [OMG01].

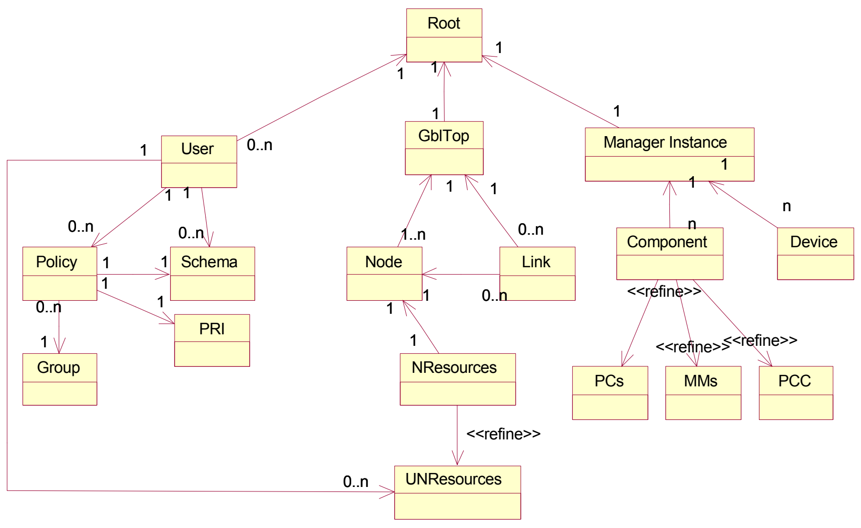

Figure 5 - 3. Representation of the MANBoP Information Model Objects

The Information Model Objects description is done in the following sub-sections. The next three sub-sections deal with the objects containing respectively, user, managed topology and manager information.

A User-related objects

The User Information Model Objects are those pending from the User object, the branch in the left. These are User, Policy, PRI, Schema and Group objects.

The User object contains user information such as its username, password, references to the policies he has introduced, references to the schemas against which his policies must be validated. These schemas will effectively delimit what fields and values the user can introduce in his policies, and therefore, what is he allowed to do in the managed network.

The Policy IMO has been already explained in the thesis document. It contains all policy fields necessary to correctly process the user policy. Since it has already been extensively described in the first section of the chapter, we will just refer to that section for more information.

The PRI IMO contains information about the resources affected by the enforcement of this policy. This IMO is used to find out exactly what resources must be freed when the policy is removed.

The Group IMO contains information oriented to facilitate the processing of policy groups, such as the policy group name, the execution strategy or references to received, send and enforced group policies.

The Schema IMO contains information to map a user with his access rights. More specifically, it maps a combination of user and functional domain with the access rights, in the form of XML Schema, that the user has to access that policy domain.

a User object

The User object contains user-related information. There is one user IMO for each user registered in a MANBoP system.

It contains the user credential. In the current implementation, the credential consists of a username and a password. This information is used by the authentication components and classes to validate the received user policies.

The table below summarises the information included within this object:

Attribute Name	*IDL Type*	*Description*
username	string	Unique identifier of a user of the MANBoP infrastructure.
password	string	Password of the user identified with the username field.

Table 5 - 15. The User object attributes

This information is mapped in the implementation into a JAVA class. The User class is a final class (not modifiable) with two fields and a constructor. The fields are strings containing respectively, the user's username and password. These are used to authenticate user's requests by means of policies.

The constructor of the mapped JAVA User class simply initiates the instance fields with the values received as parameters. The constructor signature is: *public User(String, String)*

The User JAVA class is included within the *es.upc.nmg.MANBoP.IM. User* JAVA package.

b Group object

The Group object contains information about a policy group that has been introduced within the MANBoP system. There is one Group object per policy group introduced. It contains policy group information such as the group identifier, the execution strategy, number of policies, actual group enforcement status and lists with the received, sent, enforced and removed group policies. This information is used by the PFwCnt class within the Policy Consumer Manager component for deciding when a policy from a group must be processed.

The table below summarises the information included within this object:

Attribute Name	IDL Type	Description
pgnum	long	Number that uniquely identifies the policy group between those sent by the same user.
nofp	long	Number of policies forming the group.
execst	long	Number specifying the execution strategy that must be followed to process this policy group.
recvPs	sequence<t_order>	List of group policies, identified by the group position, received.
sendPs	sequence<t_order>	List of group policies, identified by the group position, forwarded to be processed.
enfPs	sequence<t_order>	List of group policies, identified by the group position, that have been enforced.
remPs	sequence<t_order>	List of group policies, identified by the group position, that have been removed.
status	long	Number indicating the current status of the policy group. Its possible values are: waiting for policy (0), waiting for confirmation (1), completed (2).

Table 5 - 16. The Group object attributes

This information is mapped in the implementation into a JAVA class. The Group class is a final class (not modifiable) with eight fields and a constructor. The fields are four integers and four arrays of t_order class instances. The four integers contain respectively, the policy group number, number of policies, execution strategy and status of the policy group. The four arrays contain the positions (the t_order class structure has been described in the Policy Information Model section see pag. 220) of the received, sent, enforced and removed policies respectively.

The constructor of the mapped JAVA Group class simply initiates the instance fields with the values received as parameters. The constructor signature is: *public Group(int, int, int, t_order[], t_order[], t_order[], t_order[], int).*

The Group JAVA class is included within the *es.upc.nmg.MANBoP.IM.User.Policy* JAVA package.

c Schema object

The Schema IMO is used to assign XML Schemas to users. One Schema IMO is created for each combination of functional domain and user with certain access rights to access that domain. The Schema IMO contains only a string with the name of the XML Schema file against which user policies pertaining to that domain should be checked. This name is used by the Authorisation Check Component to obtain the access rights, in the form of an XML Schema assigned to the user that is trying to introduce a policy from a particular functional domain.

The table below summarises the information included within this object:

Attribute Name	IDL Type	Description
schemaId	string	Name of the assigned XML Schema file without the extension (i.e. .xsd).

Table 5 - 17. The Schema object attributes

This information is mapped in the implementation into a JAVA class. The Schema class is a final class (not modifiable) with one field and a constructor. The field is just a string with the XML Schema file name.

The constructor of the mapped JAVA Schema class simply initiates the instance field with the value received as parameter. The constructor signature is: *public Schema(String).*

The Schema JAVA class is included within the *es.upc.nmg.MANBoP.IM. User.Policy* JAVA package.

d PRI object

The PRI (Policy Resource Information) IMO stores the resources either reserved or configured after the enforcement of a policy. This information is kept within the Database to ease the complete removal of policy-related information from the managed devices when the policy expires or is uninstalled. At the current proof-of-concepts implementation the PRI IMO only contains the list of nodes affected. However, in future versions this object must be extended with more fine-grained information. There is one PRI IMO per policy enforced within the system.

The table below summarises the information included within this object:

Attribute Name	IDL Type	Description
Nodes	Sequence<string>	List of nodes whose configuration or state has been modified due to the policy enforcement.

Table 5 - 18. The PRI object attributes

This information is mapped in the implementation into a JAVA class. The PRI class is a final class (not modifiable) with one field and a constructor. The field is an array of strings with the identifiers of configured nodes.

The constructor of the mapped JAVA PRI class simply initiates the instance field with the value received as parameter. The constructor signature is public PRI (String *[]).*

The PRI JAVA class is included within the *es.upc.nmg.MANBoP.IM. User.Policy* JAVA package.

B Managed topology objects

The managed topology IMOs are those pending from the GblTop object, the branch in the middle of the diagram. These are the GblTop itself, Node, Link, NResources and UNResources objects.

The GblTop object contains all Nodes and Links forming the managed topology. Each Node object contains all Link objects representing its outgoing links and the NResources object representing its overall resources. Finally, the Node has a reference to the UNResources object that contains the total used resources in the Node. The UNResources object is a refinement (implemented as a simple inheritance) of the NResources object.

An exhaustive description of these Information Model Objects is given in the following sub-sections.

a GblTop object

The GblTop object contains information about all elements (i.e. nodes and links) conforming the managed topology. Furthermore, the GblTop object contains also a list with the references to those nodes acting as access points to the managed network. This list is needed, in particular, by the TEManager to establish all possible paths between access or end points within the network and their associated costs.

The table below summarises the information included within this object:

Attribute Name	*IDL Type*	*Description*
nodes	sequence<string>	List of identifiers of all nodes within the managed topology.
aps	sequence<string>	List of identifiers of those nodes acting as access points within the managed topology.
links	sequence<string>	List of identifiers of all links within the managed topology.

Table 5 - 19. The GblTop object attributes

This information is mapped in the implementation into a JAVA class. The GblTop class is a final class (not modifiable) with three fields and a constructor. The fields are string arrays containing respectively, the nodeIds of all managed network nodes, the nodeIds for all access points of the managed network and the linkIds of all managed network links. nodeIds and linkIds are fields of the Node and Link JAVA classes uniquely identifying each instance of these classes respectively. These identifiers are used to retrieve from the DB the appropriate Node and Link object when necessary.

The constructor of the mapped JAVA GblTop class simply initiates the instance fields with the values received as parameters. The constructor signature is: *public GblTop(String[], String[], String[])*

The GblTop JAVA class is included within the *es.upc.nmg.MANBoP.IM. Topological.MgdTop* JAVA package.

b Node object

The Node object contains information about node properties and its resources. Node resources have been divided in active and passive resources. Passive resources are represented by references to the nodes outgoing links, while active resources are represented by one reference to a NResources object and another one to UNResources object. The active resources references are different from *null* only when the node is an active or programmable one. This is indicated by one of the object attributes.

The table below summarises the information included within the Node Information Model Object:

Attribute Name	*IDL Type*	*Description*
nodeId	string	Unique identifier of the Node object instance. It is the IP address of the managed node.
type	long	Specifies whether the node is passive (0), active (1) or programmable (2)
edge	boolean	True when the node acts as access point to the managed network
outL	sequence<string>	List of identifiers of all links leaving the node.
inL	sequence<string>	List of identifiers of all links entering the node.
nResoId	string	Identifier of the NResources object instance containing the overall node resources.
uNResoId	string	Identifier of the UNRresources object containing the used node resources.

Table 5 - 20. The Node object attributes

This object is mapped in the implementation to a JAVA class. The Node class is a final class (not modifiable) with seven fields and a constructor. The seven fields are the direct mapping of the above attributes into class fields. Hence, the nodeId, type, edge, outL, inL, nResoId and uNResoId attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules [OMG02c]. The last four fields contain references to Link, NResources and UNResources objects. These references are in the form of the linkId and nResoId and uNResoId identifiers for the respective object instances. These identifiers will be described in the corresponding object description section.

The constructor of the mapped JAVA Node class simply initiates the instance fields with the values received as parameters. The constructor signature is: *public Node(String, int, boolean, String[], String[], String, String)*

The Node JAVA class is included within the *es.upc.nmg.MANBoP.IM. Topological.MgdTop* JAVA package.

c Link object

The Link Information Model Object contains information about the properties and resources of a managed link. The resource information given is not only the total available resources but also the used resources. For this proof-of-concepts implementation the only link resource considered is the total and used link capacity in KB. However, more resource parameters can be added in future implementations.

The table below summarises the information included within the Link Information Model Object:

Attribute Name	IDL Type	Description
linkId	string	Unique identifiers of the Link object instance
sourceNode	string	nodeId of the source node of this link
sinkNode	string	nodeId of the sink node of this link
hops	long	Number of hops between the source and the sink nodes.
capacity	long	The total link capacity in KB.
ucapacity	long	The used link capacity in KB.

Table 5 - 21. The Link object attributes

This object is mapped in the implementation to a JAVA class. The Link class is a final class (not modifiable) with six fields and a constructor. The six fields are the direct mapping of the above attributes into class fields. Hence, the linkId, sourceNode, sinkNode, hops, capacity and ucapacity attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules. The sourceNode and sinkNode fields contain references to the corresponding Node objects. These references are in the form of the corresponding nodeId identifiers.

The constructor of the mapped JAVA Link class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public Link(String, String, String, int, int, int)

The Link JAVA class is included within the *es.upc.nmg.MANBoP.IM.Topological.MgdTop* JAVA package.

d NResources object

The NResources Information Model Object contains information about the total active resources associated to an active or programmable node. Furthermore, the total number of EEs available in that node and their identifiers are included. For the implemented proof-of-concepts the active resources considered are the CPU in cycles per second, disk capacity for active code in KB and memory for active code in KB. More active resources can be easily added in future implementations if necessary.

The table below summarises the information included within the NResources Information Model Object:

Attribute Name	IDL Type	Description
nResoId	string	Unique identifier of the node resources object instance
cpu	long	Total number of CPU cycles per second available in the node for active code.
disk	long	Total quantity of disk available for active code in KB.
memory	long	Total size of memory available for active code in KB.
mumberOfEEs	long	Number of EEs available in the node.
EEIds	Sequence<string>	Unique identifiers of all EEs available in the node.

Table 5 - 22. The NResources object attributes

This object is mapped in the implementation to a JAVA class. The NResources class is a final class (not modifiable) with six fields and a constructor. The six fields are the direct mapping of the above attributes into class fields. Hence, the nResoId, CPU, disk, memory, numberOfEEs and EEIds attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA NResources class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public Link(String, int, int, int, int, String[])

The NResources JAVA class is included within the *es.upc.nmg.MANBoP.IM.Topological.MgdTop* JAVA package.

e UNResources object

The UNResources is a simple inheritance from the NResources class. Therefore, its attributes, and thus its mapping to a JAVA class is the same as for the NResources class.

The difference between the two Information Model Objects, and thus JAVA classes, is the semantics of its attributes. While the NResources object reflects the total number of active resources available in a node, the UNResources objects reflect the quantity of used resources available in a node. Depending on what class contains the UNResources instance the used resources can be global for all users (it is the Node instance the one that contains a reference to this UNResources instance) or those resources used by a particular user (it would be an User instance in this case).

The UNResources, as the NResources JAVA class, is included within the *es.upc.nmg.MANBoP.IM.Topological.MgdTop* JAVA package.

C Manager-related objects

The manager-related Information Model Objects are those pending from the ManagerInstance object, the right-hand branch in the diagram. These are the ManagerInstance itself, the Component objects (i.e. PCs, MMs and PCC objects) and Device objects. The Component object is an abstract class only serving as parent class for all component objects.

The ManagerInstance contains references to all components installed within the current MANBoP system. Furthermore, it also contains the references of all Device Information Model Objects. These objects provide information about how to configure the managed node by accessing the underlying devices (independently of whether they are network elements or lower-level MANBoP instances). The information provided is that needed to access these devices and to select and install the appropriate component that must interact with them. The different component Information Model Objects contain information about its functionality and location.

An exhaustive description of these Information Model Objects is given in the sub-sections hereafter.

a ManagerInstance object

The ManagerInstance Information Model Object provides information about a MANBoP instance running at this machine. The information contained deals with the dynamic components installed in the instance as well as with the devices controlled by the current instance. One ManagerInstance object is created per MANBoP instance running within the current machine. Each MANBoP instance is numbered sequentially starting from zero. This number acts as MANBoP instance identifier within the scope of the current machine.

The table below summarises the information included within the ManagerInstance Information Model Object:

Attribute Name	*IDL Type*	*Description*
managerId	string	Unique identifier of the MANBoP instance within the current machine.
pcs	Sequence<String>	List of references to the IMOs representing the Policy Consumers currently installed within the present MANBoP instance.
mms	Sequence<string>	List of references to the IMOs representing the Monitoring Meters currently installed within the present MANBoP instance.
pcc	String	Reference to the Information Model Object representing the Policy Conflict Check component installed within the present MANBoP instance.
devices	Sequence<string>	List of references to the IMOs representing Device controlled by the present MANBoP instance.

Table 5 - 23. The ManagerInstance object attributes

This object is mapped in the implementation to a JAVA class. The ManagerInstance class is a final class (not modifiable) with five fields and a constructor. The five fields are the direct mapping of the above attributes into class fields. Hence, the managerId, pcs, mms, pcc and devices attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA ManagerInstance class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public ManagerInstance(String, String[], String[], String, String[])

The ManagerInstance JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance* JAVA package.

b Device object

The Device IMO provides information about either a managed network element or a lower-level MANBoP instance with which the current MANBoP instance interacts. One Device object is created per managed node. Although, several managed nodes might be configured through a single lower-level MANBoP instance, creating one Device object per node eases the process of finding the correct Device Information Model Object based on the node that must be managed. The information provided is used to locate and correctly interact with the device as well as to select and install the appropriate type of Policy Consumers and Monitoring Meters components that will interact with that device.

The table below summarises the information included within the Device Information Model Object:

Attribute Name	*IDL Type*	*Description*
id	string	Unique identifier of the Device object instance which is equal to the assigned managed nodeId.
iface	string	Unique identifier of the underlying device's interface
nodeSetId	string	Unique identifier of the nodeSet assigned to this device. It has the following structure: <IPAddr>_<interface>
nodeSetLoc	string	String containing the exact location where dynamic components for this nodeSet must be installed.
addr	string	Internet address of the underlying device
info	sequence<string>	Array of strings containing any possible extra information that might be needed to contact the underlying device

Table 5 - 24. The Device object attributes

This object is mapped in the implementation to a JAVA class. The Device class is a final class (not modifiable) with five fields and a constructor. The five fields are the direct mapping of the above attributes into class fields. Hence, the id, iface, nodeSet, addr and info attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA Device class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public Device(String, String, String, String, String, String[])

The Device JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance.UndInt* JAVA package.

c Component object

The Component Information Model Object contains the abstract representation of every possible component dynamically installed in the system. All fields contained within this object must also appear in the component IMO. This component might be a PCC, PC or MM. The information contained within this object refers to those fields common to all components. No Component IMO is ever created or stored within the system, it is just used as basic class to all other component-specific IMOs.

The table below summarises the information included within the Component Information Model Object:

Attribute Name	IDL Type	Description
componentId	string	Unique identifier of the component within the system.
version	string	Component version.

Table 5 - 25. The Component object attributes

This object is mapped in the implementation to a JAVA class. The Component class is a class with two fields and a constructor. The two fields are the direct mapping of the above attributes into class fields. Hence, the componentId and version attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The Component JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance.Components* JAVA package.

d PCC object

The PCC (Policy Conflict Check) object provides information about the PCC component that is currently running within this MANBoP instance. The information contained in this IMO is that obtained from the Component IMO plus a new field containing the list of functional domains that this version of PCC component supports. There can be only one instance of the PCC IMO per MANBoP instance, since there can be only one PCC component at the same time within a MANBoP instance. The object is named following the code naming convention rules already described in a previous section.

The table below summarises the information included within the PCC IMO:

Attribute Name	IDL Type	Description
componentId	string	Unique identifier of the component within the system.
version	string	Component version.
suppDomains	Sequence<string>	Array of strings, each one containing the identifier of a policy functional domain supported by the PCC component represented by this object.

Table 5 - 26. The PCC object attributes

This object is mapped in the implementation to a JAVA class. The PCC class is a final class (not modifiable) with three fields and a constructor. The three fields are the direct mapping of the above attributes into class fields. Hence, the componentId, version and suppDomains attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA PCC class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public PCC(String, String, String[])

The PCC JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance.Components* JAVA package.

e PC object

The PC (Policy Consumer) Information Model Object provides information about PC components currently running within this MANBoP instance. The information contained in this IMO is basically that obtained from the Component IMO plus two new fields that indicate respectively the position within the management infrastructure at which this component is expected to run, and the underlying interface over which this component works. Within the DB, there is one instance of PC IMO per Policy Consumer dynamically installed within the system. The object is named following the code naming convention rules already described in a previous section.

The table below summarises the information included within the PC IMO:

Attribute Name	*IDL Type*	*Description*
componentId	string	Unique identifier of the component within the system.
version	string	Component version.
mgmtTopId	long	Identifier of the position within the management infrastructure at which this component is expected to run.
iface	string	Unique identifier of the device interface with which this component is expected to work.

Table 5 - 27. The PC object attributes

This object is mapped in the implementation to a JAVA class. The PC class is a final class (not modifiable) with four fields and a constructor. The four fields are the direct mapping of the above attributes into class fields. Hence, the componentId, version, mgmtTopId and iface attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA PC class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public PC(String, String, int, String)

The PC JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance.Components* JAVA package.

f MM object

The MM (Monitoring Meter) Information Model Object provides information about MM components currently running within this MANBoP instance. The information contained in this IMO is basically that obtained from the Component IMO plus two new fields that indicate respectively the position within the management infrastructure at which this component is expected to run, and the underlying interface over which this component works. Within the DB, there is one instance of MM IMO per Monitoring Meter dynamically installed within the system. The object is named following the code naming convention rules already described in a previous section.

The table below summarises the information included within the MM IMO:

Attribute Name	*IDL Type*	*Description*
componentId	string	Unique identifier of the component within the system.
version	string	Component version.
mgmtTopId	long	Identifier of the position within the management infrastructure at which this component is expected to run.
iface	string	Unique identifier of the device interface with which this component is expected to work.

Table 5 - 28. The MM object attributes

This object is mapped in the implementation to a JAVA class. The MM class is a final class (not modifiable) with four fields and a constructor. The four fields are the direct mapping of the above attributes into class fields. Hence, the componentId, version, mgmtTopId and iface attributes are mapped to fields of the same name with the corresponding JAVA types following the IDL-to-JAVA mapping rules.

The constructor of the mapped JAVA MM class simply initiates the instance fields with the values received as parameters. The constructor signature is:

Public MM(String, String, int, String)

The MM JAVA class is included within the *es.upc.nmg.MANBoP.IM. ManagerInstance.Components* JAVA package.

Section V.4 – Implemented Code

1st System Bootstrap

The first system functionality implemented has been that of the system bootstrap. To cover this functionality we have defined the appropriate interfaces in IDL files and implemented some of the defined methods from several MANBoP packages as well as the Code Installing Application utility and some exceptions. All system IDL files are included in the appendix A.

In concrete, the public and protected methods implemented are detailed in the table below:

Component Name	Class Name	Method Signature
PCM	PCMCoreImpl	public static void main(String[])
		public void main(int, String, String)
	GraphBuilder	protected boolean instObjects(String, String, int)
DmMs	DLgcImpl	public DLgcImpl(int, int, String, ORB)
ACC	ACntImpl	public ACntImpl(String, int, ORB)
PE	PECoreImpl	public PECoreImpl(int, int, ORB)
TEManager	TECoreImpl	public TECoreImpl(ORB)
SigDemux	SDCoreImpl	public SDCoreImpl(int, int, ORB)
DB	DBCoreImpl	public DBCoreImpl(String, ORB)
		public int getInstId()
DB	TopologyImpl	public boolean createGblTop(String[], String[], String[])
		public boolean createLinkObj(String, String, String, int, int, int)
		public boolean createNResoObj(String, int, int, int, int, String[])
		public boolean createTopObj(String, int, boolean, String[], String[], String, String)
		public GblTop getGblTop() throws DBObjectNotFound
	ManagerImpl	public boolean createUndIntObj(String, String, String, String)
		public boolean setMI(ManagerInstance)
		public ManagerInstance getMI() throws DBObjectNotFound
CIA	CIAImpl	public static void main(String[] args)
	CodeServerImpl	public static void main(String[] args)

Table 5 - 29. Methods implemented for the system bootstrap

In the following sub-sections we provide a more exhaustive description of the tasks implemented within these methods and the implemented exceptions.

A PCM: PCMCoreImpl

a public static void main(String[])

The main method used to start the system from the command line. The arguments that must be introduced to start a MANBoP instance are:

· A number indicates the location of the instance within the management infrastructure. Possible values are: 0 when at the network level, 1 when at the element level, 2 when at the network level working over element level MANBoP instances and 3 when at the network level working over subnetwork level MANBoP instances.

· The path where the file with the managed topology information is located.

· The path where the file with the underlying devices interface information is located.

This method just makes an initial test over the arguments to assess their correctness and starts the ORB and the POA storing its references in protected class attributes.

Furthermore, within this main method the system properties that are needed to run the MANBoP instance correctly are retrieved from the '*manbop.props*' file. This file is located at the root path where the MANBoP system is installed. It contains properties such as the directory where the MANBoP package is located.

Then, the method creates an instance of the PCMCoreImpl class passing the received arguments to the constructor. These arguments are stored as attributes of the PCMCoreImpl: the location within the management infrastructure as a public attribute and the file paths as protected attributes. Finally, the '*public void main(int, String, String)*' method form the PCMCoreImpl object is called.

b public void main(int, String, String)

This overloaded main method coordinates the whole MANBoP instance bootstrap taking into account the specified location.

The bootstrap of the different components and classes is realised following a specific order, since bootstrap processes within some components need that other components are already running. The bootstrap sequence is as follows:

· 1. Database: all DB interfaces are started and registered in the Naming Service.

· 2. PCM internal classes: all PCM internal classes are instantiated in this order: GraphBuilder, PFwCntImpl, PCCnt, PCCCnt and LfCnt. Only the PFwCntImpl object offers an external interface to other MANBoP components, hence it is the only one started as CORBA object and registered in the Naming Service. References to all these objects are stored as private attributes of the PCMCoreImpl object.

· 3. Location-independent MANBoP components: all components that should be started irrespectively of the location at which the MANBoP instance is running are started now. These components are started in the following order: Policy Editor (PECoreImpl), Decision-making Monitoring system (DLgcImpl), Authorisation Check Component (ACntImpl) and the SigDemux (SDCoreImpl). All these components are registered in the Naming Service.

· 4. Location-dependant MANBoP components: the last component instantiated is the TEManager. It is instantiated only if the MANBoP instance is not running at the element level. When instantiated, it is also registered in the Naming Service.

B PCM: GraphBuilder

a protected boolean instObjects(String, String, int)

This method is called during bootstrap by the GraphBuilder constructor and every time the managed topology is updated. The method arguments are the file paths of the managed topology and underlying devices files and an integer that specifies whether the method is being called due to an addition of a node to the managed topology (0), a removal of a node from the managed topology (1) or because the system is being bootstrapped (2). Based on the value of this integer private methods realising the requested functionality are called.

When called during the bootstrap, the functionality implemented in the GraphBuilder object is that of creating the Information Model Objects that reflect the information within the managed topology and underlying devices files. Such objects are the GblTop, Node, Link, NResources and Device objects, which have already been described in detail in the Information Model section. The way the information is structure within the managed topology and underlying devices files is described in appendix B. It must be pointed out that the ManagerInstance object stored in the database is updated with references to all Device objects created during this process.

C DmMs: DLgcImpl

a public DLgcImpl(int, int, String, ORB)

This constructor method initialises the component variables with the received parameters (i.e. management topology identifier, MANBoP instance identifier, path where MANBoP directories are placed and ORB where the component must run) and starts the component on the received ORB.

D ACC: ACntImpl

a public ACntImpl(String, int, ORB)

This constructor method initialises the component variables with the received parameters (i.e. path where MANBoP directories are placed, MANBoP instance identifier and ORB where the component must run) and starts the component on the received ORB.

E PE: PECoreImpl

a public PECoreImpl(int, int, ORB)

This constructor method initialises the component variables with the received parameters (i.e. management topology identifier, MANBoP instance identifier and ORB where the component must run) and starts the component on the received ORB.

F TEManager: TECoreImpl

a public TECoreImpl(ORB)

This constructor method just starts the component on the received ORB. For this proof-of-concepts the Traffic Engineering Manager component has no been implemented.

G SigDemux: SDCoreImpl

a public SDCoreImpl(int, int, ORB)

This constructor method of the SDCoreImpl class is used mainly for starting and registering in the database all SigDemux interfaces. In particular, it starts one SigDemux component (i.e. its interfaces) per managed nodeSet. The received parameters are two integers and an ORB reference. The integers contain respectively the managed topology identifier and the MANBoP instance identifier. These two values are needed to the correct behaviour of the component. Finally, the ORB is used to start the SigDemux component within the same ORB as the PCM component to avoid unnecessary resource consumption.

H DB: DBCoreImpl

All Information Model Objects are stored in the Database. The DB offers methods to create, retrieve, modify and remove all these objects. These methods are grouped under different DB interfaces according to the object types being stored. The storage is made by serialising the object into a file at the appropriate path, as described in the Naming Convention section.

a public DBCoreImpl(String, ORB)

This constructor method starts and registers within the Naming Service the different Database interfaces. Furthermore, this method is responsible of obtaining the number of ManagerInstance objects stored in the DB (thus, the number of MANBoP instances running in this machine) and assigns the corresponding value to the MANBoPInstanceId field based on it. The PCMCoreImpl class will afterwards obtain this value. The constructor is also responsible of creating and storing the ManagerInstance object for the current MANBoP system.

The received parameters are String and an ORB reference. The first one contains the root directory from where all MANBoP files hang. This information is obtained at bootstrap by the PCM component. The latter is used to start the DB component within the same ORB as the PCM component to avoid unnecessary resource consumption.

b *public int getInstId()*

This method is offered by the DB through the DBCoreImpl class. It is used only once at bootstrap. The PCM component accesses this method to retrieve the value of the MANBoP instance that has been calculated by the DBCoreImpl class at bootstrap.

There are no received arguments, and the logic of the method is as simple as returning the value of the MANBoPId protected field of the DBCoreImpl class. This field is an integer, which is directly returned.

I *DB: TopologyImpl.*

a *public boolean createGblTop(String[], String[], String[])*

Method offered by the DB, through the Topology interface, to create the GblTop object based on the received arguments and store it.

The received arguments are three arrays of strings with the identifiers of the nodes, access points and links that conform the managed topology respectively.

The logic created for this method simply instantiates the GblTop object with the received arguments and stores (i.e. serialises) it at the appropriate location (i.e. file path).

The method returns a boolean indicating whether the object creation has succeeded.

b *public boolean createLinkObj(String, String, String, int, int, int)*

Method used for creating and storing a Link object.

The received arguments are the link identifier, source node identifier, sink node identifier, number of hops between the source and the sink, total link capacity and used link capacity.

The functionality implemented in the method instantiates a Link object with the received arguments and stores it at the appropriate location.

The method returns a boolean indicating whether the object creation has succeeded.

c *public boolean createNResoObj(String, int, int, int, int, String[])*

This method creates and stores two node resource objects: one NResources and one UNResources object. As this method is used only at bootstrap or when a node is being added to the managed topology that is, while the node resources are still used. We simplify the node resources objects creation process by instantiating both the NResources object (specifying the total amount of resources in the node) and the UNResources object (specifying the amount of resources unused) in a single step.

The received arguments are the node resources identifier, number of CPU cycles, disk, memory, number of EEs within the node and a list with the node EE identifiers.

The programmed functionality instantiates a NResources object and an UNResources object with the received arguments and stores them at the appropriate location.

The method returns a boolean indicating whether the objects creation has been successful.

d *public boolean createTopObj(String, int, boolean, String[], String[], String, String)*

Method used for creating and storing a Node object within the DB.

The arguments received are the Node identifier, the type of node (i.e. passive, active or programmable), a boolean indicating whether the node is an access point or not, a list of outgoing link identifiers, a list of incoming list identifiers, the associated NResources object identifier and the associated UNResources object identifier.

The code implemented instantiates a Node object based on the received arguments and stores it at the appropriate location.

The method returns a boolean indicating whether the objects creation has been successful.

e *public GblTop getGblTop() throws DBObjectNotFound*

This method is used for retrieving from the Database the GblTop object of the current MANBoP instance.

The method receives no arguments; the MANBoP instance identifier needed for retrieving the object is directly obtained from a DBCoreImpl field.

The method returns the GblTop object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

J *DB: ManagerImpl*

a *public boolean createUndIntObj(String, String, String, String, String, String[])*

This method is used for creating and storing in the Database a Device object.

The arguments received are four strings and an array of strings. The four strings specify the device or node identifier, the device interface identifier, the device nodeSet identifier, the device nodeSet location and the device address. Finally, the array of strings contains any possible extra information that might be needed to contact the underlying device.

The method programmed instantiates a Device object based on the received parameters and stores it at the appropriate location.

The method returns a boolean indicating whether the objects creation has been successful.

b public boolean setMI(ManagerInstance)

Method used for storing in the database the ManagerInstance object received as parameter.

The only argument received is the ManagerInstance object, which is stored at the appropriate Database location.

The method returns a boolean indicating whether the object storage has been successful.

c public ManagerInstance getMI() throws DBObjectNotFound

Used for retrieving from the Database the ManagerInstance object of the current MANBoP instance.

The method receives no arguments; the MANBoP instance identifier needed for retrieving the object is directly obtained from a DBCoreImpl field.

The method returns the ManagerInstance object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

K CIA: CIAImpl

The Code Installing Application is not considered part of the MANBoP framework. Nonetheless, it offers a service of great importance to MANBoP and it has been designed indeed to cover MANBoP needs. The CIA service is used to download and dynamically install components. These components are installed in the directory specified in the method call and run in the same ORB as the CIA service.

The CIAImpl component acts as client component of the CIA service. There must be one CIAImpl component per machine where MANBoP components might be dynamically installed. The CIA clients obtain the CodeServerImpl component, which acts as code server, from the Naming Service that runs at port 12002 of the machine introduced at the CIAImpl component bootstrap. Each CIAImpl component registers itself at the Naming Service of the machine where the MANBoP instance to which they serve runs. The naming convention for this registration has already been described at the Naming Convention section (see pag. 216).

a public static void main(String[] args)

The main method used to start the CIA service client component from the command line. The arguments that must be introduced to start it are:

· A string containing the domain name of the CIA server (the code server).

· The IP address of the station where the MANBoP instance linked with this CIA client runs.

· The IP address with which we want the CIA component to register at the Naming Service. By default it would be the machine's localhost IP address.

This method just makes an initial test over the arguments to assess their correctness and starts the ORB and the POA storing its references in private class attributes.

Then, it obtains from the Naming Service running at port 12002 of the code server domain, the CodeServerImpl component that will act as CIA server.

Finally, it registers itself at the Naming Service running at port 12001 of the station where the MANBoP instance linked with this CIA client runs and starts the service.

L CIA: CodeServerImpl

The CodeServerImpl component acts as a server of the CIA service, that is, a code server. This server will receive code requests, look for the requested code in the repository and send it to the CIA client that requested it if it could be found.

The code repository is structure around a root directory introduced at bootstrap. From this directory hang a number of directories containing code files and a *'Schemas'* directory that contains XML Schema files. The directories containing code files are named *'<codeId>'*, where the code identifier is received in the request. This directory contains two files which are both sent: one *'<codeId>.sto'* file and one *'<codeId>.jar'* file.

The *'.sto'* file contains a JAVA class that will be dynamically loaded in the JVM of the CIA client. This JAVA class is started to install correctly the MANBoP component requested.

The *'.jar'* file is a JAVA jar package containing all component files.

The CodeServerImpl service uses sockets at port 12003 to send the requested files.

a public static void main(String[] args)

The main method used to start the CIA service server component from the command line. The argument that must be introduced to start it is:

· A string containing the directory from where all code files hang.

This method just makes an initial test over the arguments to assess their correctness and starts the ORB and the POA storing its references in private class attributes.

Finally, it starts the CIA code server service.

M Exceptions

a DBObjectNotFound

This exception is sent by the DB component interfaces when a requested object could not be found in the DB.

The exception is defined with just one field, which is a string used to provide extra information about the cause of the raised exception.

This exception will be captured by the client component so that it can react in consequence.

2nd Policy Processing: Policy Reception and Policy Group Processing

After the bootstrap, we have implemented policy-processing functionality. Extensive description of this functionality can be found in the Proposed Model chapter. In this section we will just enumerate and briefly describe the code that has been implemented to develop part of the designed functionality and particularly, within this section, code involved when the policy is introduced in the system.

The implementation covered the definition of a number of IDL files, the development of methods from the Policy Editor, Policy Consumer Manager and Database components and a number of exceptions. All system IDL files are included in appendix A.

In concrete, the public and protected methods implemented are detailed in the table below:

Component Name	Class Name	Method Signature
PE	AuthenticationModule	protected static boolean authenticate(credential)
	PECoreImpl	public void recvXPolicy(credential, string) throws UnknownUser
PCM	PFwCntImpl	public void dispatch(credential, string) throws UnProcessablePolicy
	Parser	public Policy parse() throws WrongSyntaxException
		protected static String getCondNames(Policy)
PCM: PGES	seqACK_PGES	protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy
	seqNACK_PGES	protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy
	first_PGES	protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy
	atomic_PGES	protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy
	be_PGES	-[24]
DB	UserImpl	public User getUser(string) throws DBObjectNotFound
		public boolean setUser(User)
	PolicyImpl	public Policy getPolicy(t_policyId) throws DBObjectNotFound
		public boolean setPolicy(Policy, String, t_policyId)
	PGroupImpl	public boolean setGroup(Group,string)
		public boolean setGroupP(Policy, String, credential, t_policyId)
		public Group getGroup(int, string) throws DBObjectNotFound
		public Policy getGroupP(int, string, string) throws DBObjectNotFound
		public string getGroupXP(int, string, string) throws DBObjectNotFound
		public boolean rmGroupP(int, string, string)
		public t_policyId getGroupPId(int, String, String)
		public credential getGroupPUserCred(int, String, String)

Table 5 - 30. Methods implemented for policy group processing

In the following sub-sections we provide a more exhaustive description of the tasks implemented within these methods and the implemented exceptions.-

A PE: AuthenticationModule

a protected static boolean authenticate(credential)

This method is called by the PECoreImpl class to authenticate a user who is trying to introduce a policy in MANBoP. The only argument defined is the credential of the user. In the current implementation, the credential structure

[24] Although not method has been implemented yet for the best effort Policy Group Execution Strategy, we've added it here to show the five execution strategies that have been implemented in this proof-of-concepts.

consists of two strings containing respectively the user name and the password. The method returns a boolean.

Based on this information, the logic implemented within the AuthenticationModule will try to retrieve from the DB the User IMO corresponding to that user and compare the credential information contained within that object with the one received. Either if the User IMO searched cannot be found or if the credential information contained within the object does not match with the one received the method returns false, otherwise, it returns true.

A User IMO can be created either after a user registration through the GUI or after the enforcement of a delegation policy. We have also implemented a helping tool named UserCreator to directly create User objects in the DB. More information about this helping tool can be found at appendix D.

To retrieve the User IMO information the AuthenticationModule uses the User interface from the DB implemented in the UserImpl class.

B PE: PECoreImpl

a public void recvXPolicy(credential, string) throws UnknownUser

Method offered to higher-level applications or other MANBoP instances to introduce policies. This is one of the possibilities offered by the PE to introduce policies and probably the main alternative to the one represented by the PE component GUI. The method has been defined with two arguments: the user's credential and a string. The user's credential is used to request the user authentication to the AuthenticationModule class just described. The string contains the policy to be processed expressed in XML language.

The logic implemented in this method is as simple as requesting the authentication of the user to the AuthenticationModule class and, unless the authentication fails, forward the policy to the PCM component through its dispatch interface implemented by the PFwCntImpl class and described in the next sub-section.

When the user authentication fails the method throws an UnknownUser exception.

C PCM: PFwCntImpl

a public void dispatch(credential, string) throws UnProcessablePolicy

The dispatch method implements one of the "northern" access methods defined in the PCM component interface. It is used to introduce policies to be processed within this MANBoP instance. More specifically, the PE component uses this method to forward policies to the PCM component. The method defines two input arguments, which are the user's credential and a string with the policy in XML.

The implemented logic for this method deals with the processing of policy groups with the help of many auxiliary classes implemented that are described hereafter. Apart from the policy-group processing logic, this method also requests the parsing of the XML policy received to a JAVA Policy IMO so that the processing of the policy is computationally easier and faster.

When the received policy is not part of a policy group the policy is directly stored in the DB and sent to the PCMCoreImpl object for its processing. Otherwise, the policy is processed according to its policy group position and the policy group execution strategy specified.

When the policy could not be processed correctly within this component by any reason an UnProcessablePolicy exception is thrown. More information about the cause of the exception might be included in the exception itself.

D PCM: Parser

a public Policy parse() throws WrongSyntaxException

This method is offered by the internal class Parser to create a Java Policy IMO based on the information extracted from the XML policy received. The method is defined without input parameters since the string containing the XML policy to be parsed is introduced in the constructor, when the Parser instance is created. The method returns the parsed Policy object.

The implemented logic for this method simply looks for chains of characters that establish the location of expected information within the XML information, extracts the information found and converts it to the type specified in the corresponding field of the Policy object.

If at any time along the process an expected information (i.e. chain of characters) is not found, a WrongSyntaxException is thrown and the parsing is skipped.

b protected static String getCondNames(Policy)

This method is used by other component classes, in particular the PCMCore class, to build a string containing a concatenation in alphabetic order of all simple condition names included within the policy. This method is used when the t_policyId structure corresponding to the policy is being created. The policy identifier contains the concatenation of simple conditions to ease the policy conflict checking functionality by simplifying the process of finding potentially conflicting policies (e.g. those that have common conditions).

The method receives the Policy IMO from which the condition names must be extracted and returns the concatenated simple condition names in form of a string.

E PCM: PGES: seqACK_PGES

a protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy

The seqACK_PGES class encloses all the logic related to the sequence with acknowledgement policy group execution strategy. This execution strategy forwards the group policies only if the precedent policy, based on group positions, has been enforced correctly (i.e. the enforcement acknowledgement has been received). More specifically, the *precv* method contains the logic for processing group policies following this execution strategy at the time they are introduced in the system. The method is defined with two arguments: the Policy IMO that corresponds to the received policy and the Group IMO of the corresponding policy group.

Based on this information, the implemented logic decides if the received policy should be processed or instead should be stored until a precedent policy is correctly enforced. It updates the Group object accordingly and returns the list of policies, identified by their policy positions that must be processed.

In case an unexpected error occurs during this process, an UnProcessablePolicy exception is thrown.

F PCM: PGES: seqNACK_PGES

a protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy

The seqNACK_PGES class, as well as its precv method, is the equivalent to the above except that the execution strategy implemented is the sequence not acknowledged policy group execution strategy. So, policies are forwarded as long as precedent policies in policy group order have been previously sent to process. Therefore, the precv method returns the list of policies (the one received plus maybe others previously received although not yet processed) that must be processed. These policies are included in the list in the same order as they must be forwarded to the PCMCoreImpl object.

In case an unexpected error occurs during this process, an UnProcessablePolicy exception is thrown.

G PCM: PGES: first_PGES

a protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy

The first_PGES class encloses the logic related to the Policy Group Execution Strategy (PGES) that enforces group policies in sequential order until one of them is enforced successfully. At this point, the group processing is stopped, that is, no more group policies are enforced. More specifically, the precv method contains the logic for processing group policies following this execution strategy at the time they are introduced in the system. The method is defined with two arguments: the Policy IMO that corresponds to the received policy and the Group IMO of the corresponding policy group.

Based on this information, the implemented logic decides if the received policy should be processed or instead should be stored until a precedent policy is enforced. It updates the Group object accordingly and returns, when any, the list of policies (identified by their policy group positions) that must be processed.

In case an unexpected error occurs during this process, an UnProcessablePolicy exception is thrown.

H PCM: PGES: atomic_PGES

a protected static t_order[] precv(Policy, Group) throws UnProcessablePolicy

The atomic_PGES class encloses the logic related with an atomic policy group execution strategy. In this execution strategy, group policies are enforced in sequential order and only when the previous group policy has been enforced correctly. Furthermore, if one of the group policies cannot be enforced, all previous group policies that had been enforced correctly are uninstalled from the system. More specifically, the precv method contains the logic for processing group policies following this execution strategy at the time they are introduced in the system. The method is defined with two arguments: the Policy IMO that corresponds to the received policy and the Group IMO of the corresponding policy group.

Based on this information, the implemented logic decides if the received policy should be processed or instead should be stored until a precedent policy is correctly enforced. It updates the Group object accordingly and returns the list of policies, identified by their policy group positions that must be processed.

In case an unexpected error occurs during this process an UnProcessablePolicy exception is thrown.

I PCM: PGES: be_PGES

The be_PGES is the simplest possible Policy Group Execution Strategy. It simply forwards the group policies to be processed as they arrive to the system. The group is completed when all group policies have been processed, no matter if they could be enforced successfully or not.

J DB: UserImpl

a public User getUser(string) throws DBObjectNotFound

This method is used for retrieving from the Database the User object that represents the user name introduced as parameter.

The method receives just one argument: a string containing the user name.

The method returns the User object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be thrown.

b *public boolean setUser(User)*

Method used for storing in the database the User object received as parameter.

The only argument received is the User object, which is stored at the appropriate Database location, as described in the naming convention section.

The method returns a boolean indicating whether the object storage has been successful.

K *DB: PolicyImpl*

a *public Policy getPolicy(t_policyId) throws DBObjectNotFound*

This method is used for retrieving from the Database a Policy object identified with a *t_policyId* structure.

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy and, hence, locate it within the DB.

The method returns the Policy object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

b *public boolean setPolicy(Policy, string, t_policyId, credential)*

Method used for storing in the database the Policy object together with the corresponding XML policy as well as the user credential linked with this policy. All these objects are received as parameters.

The arguments received are the Policy object, the XML policy and the credential, which are stored at the appropriate Database location as well as the t_policyId structure use to establish the location at which the policy objects must be stored.

The method returns a boolean indicating if the storage of the objects has been successful.

L *DB: PGroupImpl*

a *public boolean setGroup(Group,string)*

Method used for storing in the database the Group object received as parameter.

The arguments received are the Group object to be stored and a string containing the user name of the user owning the group to choose the appropriate DB location.

The method returns a boolean that indicates if the object storage has been successful.

b *public boolean setGroupP(Policy, string, credential, t_policyId)*

Method used for storing in the database the Policy object and the corresponding XML policy as well as the credential of the user that introduced the policy and the t_policyId structure of this policy. All these objects are received as parameters. These policies, and related objects, are part of a policy group. They are stored until they turn to be processed arrives. The credential and t_policyId objects are stored together with the policy to easily obtain this information, necessary for the policy processing, when the group policy is finally processed.

The arguments received are the Policy, the XML policy, the credential and t_policyId objects, which are stored at the appropriate Database location for the policy group.

The method returns a boolean indicating if the storage of all objects has been successful.

c *public Group getGroup(int, string) throws DBObjectNotFound*

This method is used for retrieving from the Database a Group object identified by the policy group number and the user name of the user owning the policy group. These two are the arguments introduced in the method as an integer containing the policy group number and a string containing the user name.

The method returns the Group object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be thrown.

d *public Policy getGroupP(int, string, string) throws DBObjectNotFound*

This method is used for retrieving from the Database a Policy IMO that corresponds to a group policy. The requested group policy is identified by the received arguments, which are an integer representing the policy group number to which the policy pertains, a string indicating the position identifier of the requested policy and another string indicating the user name of the user owning the group policy.

The method returns the Policy object, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

e *public string getGroupXP(int, string, string) throws DBObjectNotFound*

This method is used for retrieving from the Database an XML policy, stored as a serialised string, and that corresponds to a group policy. The requested XML group policy is identified by the received arguments, which are an integer representing the policy group number to which the policy pertains, a string indicating the position identifier of the requested policy and another string indicating the user name of the user owning the group policy.

The method returns the XML policy, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

f *public boolean rmGroupP(int, string, string)*

Method used for removing from the Database, in particular from the corresponding group directory, the Policy object and the corresponding XML policy pointed by the received parameters.

The received parameters are an integer with the policy group number, a string containing the position identifier of the policy to be removed and the user name of the user owning the policy.

The method returns a boolean indicating if the removal of both objects has been successful.

g *public t_policyId getGroupPId(int, String, String) throws DBObjectNotFound*

This method is used to obtain from the Database the policy identifier stored, together with a group policy, as a serialised string. The requested t_policyId structure is identified by the identification of the linked policy with the received arguments. These arguments are an integer representing the policy group number to which the policy pertains, a string indicating the position identifier of the requested policy and another string indicating the user name of the user owning the group policy.

The method returns the t_policyId structure, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

h *public credential getGroupPUserCred(int, String, String) throws*

DBObjectNotFound

This method is used to obtain from the Database the user credential stored, together with a group policy, as a serialised string. The requested credential object is located by finding the linked policy with the received arguments. These arguments are an integer representing the policy group number to which the policy pertains, a string indicating the position identifier of the requested policy and another string indicating the user name of the user owning the group policy.

The method returns the credential object in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

M Exceptions

a UnknownUser

This exception is sent by the PE component when the authentication of the user trying to introduce a policy within the MANBoP system fails.

The exception is defined with just one field, which is a string used to provide extra information.

The client might capture this exception to react in consequence.

b UnProcessablePolicy

This exception is sent when a received policy could not be processed because of unexpected errors along the process.

The exception is defined with string field providing additional information about the exception cause.

c WrongSyntaxException

The Parser class within the PCM component raises this exception when an unexpected policy syntax structure is detected.

The exception is defined with a string that can be used to provide additional information about the reason of the exception being raised.

3rd Policy processing: Task coordination and Policy checking

The next functionality we have implemented, following policy-processing steps, is the coordination of all tasks and policy checking (i.e. authentication and conflict checking). Extensive description of this functionality can be found in the Proposed Model chapter. In this section we will just enumerate and briefly describe the code that has been implemented to develop part of the designed functionality.

The implementation covered the definition of a number of IDL files, the development of methods from the Policy Consumer Manager, Authorisation Check Component, Policy Conflict Check and Database components and a number of exceptions as well as the Code Installing Application (CIA) service. All system IDL files are included in appendix A.

In concrete, the public and protected methods implemented are detailed in the table below:

Component Name	*Class Name*	*Method Signature*
PCM	PCMCoreImpl	public void procP(credential, t_policyId) throws UnProcessablePolicy
	PCCCnt	protected boolean checkConfl(t_policyId, int, String[])
ACC	ACntImpl	public boolean authorise(t_policyId, credential, String)
CIA	CIAImpl	public void dwSchema(String, String) throws CodeNotFound
		public void dwCode(String, String, String[]) throws CodeNotFound
	CIAClassLoader	protected Class findClass(String) throws ClassNotFoundException
		protected String findLibrary(String)
	CodeServerImpl	public String obtainLocation(String, String, int)
		public void obtainCode(String)
DB	PolicyImpl	public String getXPolicy(t_policyId) throws DBObjectNotFound
	SchemaImpl	public Schema getSchema(String, String) throws DBObjectNotFound
		public boolean setSchema(Schema, String, String)
	ManagerImpl	public PCC getPCC() throws DBObjectNotFound
		public boolean setPCC(PCC)

Table 5 - 31. Methods implemented for task coordination and policy checking

In the following sub-sections we provide a more exhaustive description of the tasks implemented within these methods and the implemented exceptions.

A PCM: PCMCoreImpl

a public void procP(credential, t_policyId) throws UnProcessablePolicy

The procP (stands for process policy) method implements the main functionality for coordinating the policy-processing tasks and the taking of decisions. More specifically, the PFwCntImpl class uses this method to start the processing of a policy after the policy group logic, when applicable, has been developed. The method defines two input arguments, which are the user's credential and the policy identifier of the policy to be processed.

The implemented logic for this method deals mainly with the coordination of all tasks involved in policy-processing by contacting other components and acting based on the result. The tasks are extensively described in the design chapter but, in brief, are the authorisation checking, conflict checking, monitoring, policy enforcement and result notification.

The policy to be processed is retrieved from the Database with the policy identifier received as parameter.

When the policy could not be processed correctly within this component by any reason an UnProcessablePolicy exception is thrown. More information about the cause of the exception might be included in the exception itself.

B PCM: PCCCnt

a protected boolean checkConfl(t_policyId, int, String[])

The checkConfl method implements the logic for finding out whether the PCC component installed is capable of realising the requested check, and if not, install the newer version of the PCC component before requesting to it the realisation of the appropriate policy conflict checking.

This method is always called by the PCMCoreImpl class as part of the normal policy processing functionality and is used for both static and dynamic policy conflict checks. The method defines three input arguments, which are the identifier of the policy to be checked, an integer establishing the type of check to be done and the list of node identifiers where the policy must be applied (when specified in the policy).

The method returns a boolean containing the result of the policy conflict checking.

C ACC: ACntImpl

a public boolean authorise(t_policyId, credential, String)

The authorise method implements the logic for assessing if the user introducing the policy is allowed to do so. This functionality has been implemented taking advantage of XML language and tools. In particular, we realise the assessment by validating the received XML policy against an XML Schema assigned to that user and functional domain. If the user has no XML Schema assigned for that functional domain it means that he is not authorised to access that domain. If he does have one, then his XML policy is validated against his XML Schema, which might be restricted to allow only a certain type of action or condition values. The validation itself, functionality designed for the ACkr class of the ACC component, is done by a freely available XML validation tool. More specifically, we are using the Sun Multi-Schema XML Validator tool [SunMSV].

This method is always called by the PCMCoreImpl class as part of the normal policy processing functionality. The method defines three input arguments, which are the identifier of the policy to be authorised, the credential of the user introducing the policy and a string identifying the functional domain of the policy being authorised.

The method returns a boolean containing the result of the authorisation check.

D CIA: CIAImpl

a public void dwSchema(String, String) throws CodeNotFound

This method is used to request the download of an XML Schema file to the correct directory of the MANBoP DB. Thereby, the download of XML Schemas for new policy functional domains is done dynamically avoiding the need of human intervention. The schema will only be requested when the user whose policy is being checked has rights to access the functional domain represented by that Schema, which might be a restricted schema created for the user or not.

The method logic simply requests the XML Schema file to the code server and stores it in the appropriate MANBoP DB directory. The method contains two input arguments, which are the name of the XML Schema file to be downloaded and the destination directory where this file must be stored.

The method throws a CodeNotFound exception if the requested XML Schema could not be found in the code server.

b public void dwCode(String, String, String[]) throws CodeNotFound

This method is used to request the download of a dynamically installable component to a MANBoP instance and its installation.

The method will first check that the requested code is located within the code server. If so, it requests the download of the component code, packed in a JAVA jar file, and the download of the installer class. This installer class must compulsorily implement the Installer interface.

The Installer interface defines just one method that expects two arguments: an array of strings that contain the arguments needed to appropriately start the downloaded component and the ORB where the component must run.

The installer class will be then dynamically loaded in the Java Virtual Machine (a CIAClassLoader has also been implemented to aid in this process), started and a request to its install method will be raised. When receiving this request the install class will realise all needed tasks to correctly start the dynamically installable component within the MANBoP instance. These tasks are, for example, the registration of the component at the Naming Service, or the creation of the corresponding component IMOs (i.e. PCC, PC or MM IMO) using the appropriate DB interfaces.

The dwCode method is defined with three input arguments, which are the identifier of the requested code, the version (if any is specified) of the requested code and an array of strings containing the arguments that must be used to appropriately start the component.

The method throws a CodeNotFound exception if the requested code could not be found in the code server.

E CIA: CIAClassLoader

The CIAClassLoader class extends the generic JAVA ClassLoader to provide the needed functionality to find and load the installer class. This functionality, as described in the JAVA documentation, must be introduced through the findClass and findLibrary methods. More information about the exact functionality can be found in the JAVA language documentation [SunJAVAb].

a *protected Class findClass(String) throws ClassNotFoundException*

Method used to locate and return as a Class object the installer class downloaded together with a dynamically installable component.

The method defines just one input argument: the JAVA name (including the package) of the class to be loaded.

It returns a Class object representing the installer class to be loaded. For more information refer to JAVA documentation.

b *protected String findLibrary(String)*

Method used to locate any possible library that might be needed to start a dynamically installable code.

The method defines just one input argument, which is the name of the library to be loaded.

It returns a String containing the full path where the searched library can be found.

F CIA: CodeServerImpl

a *public String obtainLocation(String, String, int)*

The obtainLocation method is used by the CIA client to check if the requested code is stored in the code repository and if so, where exactly it is placed.

The method receives three input parameters that contain respectively, the identifier of the code requested, the requested version of the code and an integer that expresses if the request is for code (0) or an XML Schema file (1).

The method returns a string containing the full path within the code repository where the requested code or XML Schema can be found.

b *public void obtainCode(String)*

This method implements the server-side code downloading functionality. When a request is received a new thread is started that will accept client bindings to the server socket and proceed to send the requested code through this socket.

The method is defined with one input parameter: a string containing the location of the requested code. This location will have been previously obtained by the CIA client through the obtainLocation method.

G DB: PolicyImpl

a public String getXPolicy(t_policyId) throws DBObjectNotFound

This method is used for retrieving from the Database the XML Policy linked with a *t_policyId* structure.

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy and, hence, locate it within the DB.

The method returns a String containing the XML Policy, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

H DB: SchemaImpl

a public Schema getSchema(String, String) throws DBObjectNotFound

This method is used for retrieving from the Database a Schema IMO for a concrete user and functional domain.

The method receives two strings containing respectively, the user name and the functional domain identifier. These strings are used to locate the Schema IMO within the DB.

The method returns the Schema, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

b public boolean setSchema(Schema, String, String)

This method is used for storing in the Database a Schema IMO for a concrete user and functional domain.

The method receives the Schema IMO to be stored and two strings containing respectively, the user name and the functional domain identifier. These strings are used to decide where to store the Schema IMO within the DB.

The method returns a boolean indicating whether the storage of the Schema IMO has been realised successfully.

I DB: ManagerImpl

a public PCC getPCC() throws DBObjectNotFound

This method is used to obtain from the Database the PCC IMO representing the Policy Conflict Check component currently running with the MANBoP instance.

Since there can be only one PCC component running at the same time within a MANBoP instance, there is no need for an input argument to identify the PCC IMO to be obtained.

The method returns the PCC IMO, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

b public boolean setPCC(PCC)

This method is used for storing in the Database a PCC IMO representing the PCC component being installed in the MANBoP instance.

The method receives the PCC IMO to be stored as parameter. It returns a boolean indicating whether the storage of the PCC IMO could be realised successfully.

J Exceptions

a CodeNotFound

Exception sent by the Code Installing Application client when a code or schema that had been requested could not be found in the code repository. This exception is not a MANBoP exception but a CIA exception, hence it has been implemented within the package where the whole CIA service is contained.

The exception is defined with just one field, which is a string used to provide extra information.

The MANBoP instance component requesting the code might capture this exception to react in consequence.

4th Policy processing: Monitoring

One of the key parts within the framework is the decision taking. This part is realised jointly by the PCM and the DmMs components. In this section we are going to describe the monitoring functionality implemented. The monitoring functionality within MANBoP is focused to evaluate the value of policy conditions; hence, it provides the basis for decision-making. Extensive description of this functionality can be found in the Proposed Model chapter. In this section we will just enumerate and briefly describe the code that has been implemented to develop part of the designed functionality.

The implementation covered the definition of a number of IDL files, the development of methods from the Policy Consumer Manager, Decision-making Monitoring system, Monitoring Meters[25] and Database components and a number of exceptions. All system IDL files are included in appendix A.

In concrete, the public and protected methods implemented are detailed in the table below:

Component Name	*Class Name*	*Method Signature*
PCM	PCMCoreImpl	public void triggerEnf(t_policyId, boolean)
DmMs	DLgcImpl	public boolean regCond(t_policyId, Policy, credential)
		public void ISValue(String, boolean)
	MMCnt	protected boolean regIS(String, t_simpleCond, credential) throws MonitoringError
DB	ManagerImpl	public Device getUndIntObj(String) throws DBObjectNotFound
		public MM getMM(String, String) throws DBObjectNotFound
		public boolean setMM(MM, String)

Table 5 - 32. Methods implemented for monitoring functionality

In the following sub-sections we provide a more exhaustive description of the tasks implemented within these methods and the implemented exceptions.

A PCM: PCMCoreImpl

a public void triggerEnf(t_policyId, boolean)

The triggerEnf (stands for trigger enforcement) method is used to start the tasks that lead to either the enforcement or de-enforcement of the policy. This method is only used by the DmMs component when the global assessment of the policy conditions changes. That is, if the policy conditions do not longer match (or assess to false) the DmMs requests the de-enforcement of the policy; on the contrary, when policy conditions do finally match (or assess to true), the DmMs requests the enforcement of the policy. The method is defined with two parameters the policy identifier of the policy to be enforced and a boolean containing the new value of the policy conditions.

The implemented logic for this method is that designed for the policy enforcement procedure. These tasks are extensively described in the design chapter but, in brief, are the Policy Consumers lifecycle control, the dynamic conflict checking, policy enforcement and result notification.

The policy to be processed is retrieved from the Database with the policy identifier received as parameter.

[25] The implementation of Monitoring Meter components will be described in the domain-dependent implementation section.

B DmMs: DLgcImpl

a public boolean regCond(t_policyId, Policy, credential)

This method implements the main functionality for coordinating monitoring tasks. More specifically, the PCMCoreImpl uses this method to request the monitoring of policy conditions and be warned when the global assessment of these conditions changes. The method defines three parameters, which are the identifier of the policy whose conditions must be monitored, the Policy IMO representing this policy and the user's credential.

The implemented logic for this method deals basically with creating a logic expression that reflects the policy conditions, map it to the policy identifier and request the monitoring of each element (i.e. Individual Statement) of the logic expression. Further information about these tasks can be found in the design chapter.

The method returns a boolean that determines if the registration of the policy conditions to be monitored could be carried out successfully.

b public void ISValue(String, boolean)

The ISValue method implements the logic to change the value of an Individual Statement within a logic expression and re-assess the logic expression. If the global value changes the DLgcImpl class will request the enforcement or de-enforcement of the corresponding policy to the PCM component. More specifically, Monitoring Meter components use this method to inform when the value of an IS they are monitoring changes its value. The method is defined with two parameters, which are, respectively, the identifier of the monitored IS and the new value of this IS. Further information about these tasks can be found in the design chapter.

C DmMs: MMCnt

a protected boolean regIS(String, t_simpleCond, credential) throws MonitoringError

This is a component internal method offered by the MMCnt class of the DmMs component to request the monitoring of an Individual Statement (IS). More specifically, the DLgcImpl class uses this method to request the monitoring of an IS by the appropriate Monitoring Meter component. The method receives three parameters: the identifier of the Individual Statement to be monitored, a structure containing the simple condition linked with this IS and the credential of the user that introduced the policy causing this monitoring.

The implemented logic for this method deals basically with finding out which Monitoring Meter component must monitor this IS, request the installation of this MM if not already installed and finally, forward to it the monitoring request. The method also updates accordingly the MM lifecycle control information kept within this class. Further information about these tasks can be found in the design chapter.

The method returns a boolean with the initial value of the Individual Statement. This initial value is calculated from the responses of all MM components contacted to monitor the IS.

When for any reason an error occurs in the registration of the IS or its monitoring to obtain the initial value the method throws a MonitoringError exception.

D DB: ManagerImpl

a public Device getUndIntObj(String) throws DBObjectNotFound

This method is used for obtaining from the Database a Device object.

The argument received is a string containing the unique identifier of the Device IMO within the system.

The method returns the requested Device IMO if it could be found within the Database. Otherwise, it throws a DBObjectNotFound exception.

b public MM getMM(String, String) throws DBObjectNotFound

This method is used to obtain from the Database a MM IMO representing one of the Monitoring Meter components currently running at the MANBoP instance.

The method receives two strings used to uniquely identify the requested MM IMO. These strings are respectively the identifier of the nodeSet where the component represented by the requested IMO must run and the identifier of the component itself.

The method returns the requested MM IMO, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

c public boolean setMM(MM, String)

This method is used to store in the Database a MM IMO.

The method receives as parameters the MM IMO to be stored and a string containing the identifier of the nodeSet where the component represented by this IMO must run. It returns a boolean indicating whether the storage of the MM IMO could be realised successfully.

d public boolean rmMM(String, String)

This method is used to remove from the Database a MM IMO representing a Monitoring Meter component no longer running within the system.

The method receives two strings used to uniquely identify the requested MM IMO. These strings are respectively the identifier of the nodeSet where the component represented by the requested IMO runs and the identifier of the component itself.

It returns a boolean indicating whether the removal of the MM IMO was realised successfully.

E Exceptions

a MonitoringError

This exception is sent by either MM components or the MMCnt class within the DmMs component when an error occurs while monitoring an Individual Statement.

The exception is defined with just one field, which is a string used to provide extra information.

5th Policy processing: Policy enforcement and result processing

The final part of policy processing, domain-independent functionality implemented is that of policy enforcement and result processing. Extensive description of this functionality can be found in the Proposed Model chapter. In this section we will just enumerate and briefly describe the code that has been implemented to develop part of the designed functionality.

The implementation covered the definition of a number of IDL files, the development of methods from the Policy Consumer Manager, Policy Consumers [26], Policy Editor, Decision-making Monitoring system and Database components and a number of exceptions. All system IDL files are included in the appendix.

In concrete, the public and protected methods implemented are detailed in the table below:

[26] The implementation of Monitoring Meter components will be described in the domain-dependant implementation section.

Component Name	Class Name	Method Signature
PCM	PCCnt	protected int enforce(Policy, String[], t_policyId, credential) throws UnableToEnforceP
	PFwCntImpl	protected void pProcSt(t_policyId, int, String)
	PCMCoreImpl	public boolean uninstallP(t_policyId, int)
	PCCCnt	protected boolean uninstPR(t_policyId, boolean)
PCM: PGES	be_PGES	protected static int last(Policy, Group, int)
	first_PGES	protected static int last(Policy, Group, int)
		protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy
	seqNACK_PGES	protected static int last(Policy, Group, int)
	seqACK_PGES	protected static int last(Policy, Group, int)
		protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy
	atomic_PGES	protected static int last(Policy, Group, int)
		protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy
PE	PECoreImpl	public void recvEnfResult(String, t_policyId, int)
DmMs	DLgcImpl	public boolean unregCond(t_policyId)
	MMCnt	protected boolean unregIS(String)
DB	ManagerImpl	public PC getPC(String, String) throws DBObjectNotFound
		public boolean setPC(PC, String)
		public boolean rmMM(String, String)
		public boolean rmPC(String, String)
	PolicyImpl	public credential getPolicyUserCred(t_policyId) throws DBObjectNotFound)
		public PRI getPRI(t_policyId) throws DBObjectNotFound
		public boolean setPRI(t_policyId, PRI)
		public int getPSts(t_policyId) throws DBObjectNotFound
		public boolean modPStatus(t_policyId)
		public boolean removeP(t_policyId)
	PGroupImpl	public boolean rmGroup(int, String)
-	GenRecv	public void recvEnfResult(String, t_policyId, int)

Table 5 - 33. Methods implemented for policy enforcement and result processing

In the following sub-sections we provide a more exhaustive description of the tasks implemented within these methods and the implemented exceptions.

A PCM: PCCnt

a protected Result enforce(Policy, String[], t_policyId, credential) throws UnableToEnforceP

This is a component internal method offered by the PCCnt class of the PCM component to request the enforcement of a policy. More specifically, the PCMCore class uses this method to request the enforcement of a policy by the appropriate Policy Consumer components. The method receives four parameters: the Policy IMO representing the policy to be enforced, an array of strings containing the identifiers of the nodes where the policy must be enforced, the identifier of the policy and the credential of the user introducing the policy[27].

The implemented logic for this method deals basically with finding out what Policy Consumer component must enforce this policy, request the installation of this PC if not already installed and finally, forward to it the enforcement request recompile the results and return them. Further information about these tasks can be found in the design chapter.

The method returns a Result type. This type consists of an integer and an array of strings. The integer indicates the enforcement result. Possible values of this integer are enforced (0), de-enforced because of policies no longer matching (1), de-enforced because removal of the policy (2), enforcement error (3) and undefined (4). The undefined value is used when the enforcement result is not known immediately (e.g. when working over other MANBoP instances). By returning an undefined value, we avoid the PCM to be waiting for an enforcement result that could last a long time. The array of strings is used to include further information related with the enforcement result as error information or others.

When for any reason an error occurs in the enforcement of the policy, the method throws a UnableToEnforceP exception.

B PCM: PFwCntImpl

a public void pProcSt(t_policyId, Result, String)

Public method offered by the PFwCntImpl class of the PCM component to receive the policy enforcement result and react accordingly. More specifically, either the Policy Consumer component or the PCMCore class can use this method to request the realisation of appropriate actions (e.g. user report, policy group execution, etc.). The method receives three parameters: the identifier of the policy enforced, the enforcement result as described in the previous method and a string containing further information in the case there has been an enforcement error.

The implemented logic for this method deals basically with, depending on the enforcement result, updating the policy status information, removing the policy from the database[28], re-start the policy group execution strategy when appropriate and finally, inform upper management layers (e.g. higher-level

[27] Although not being initially included in the design, the credential parameter has been added so that PC components can configure the node even if it has some security rights configured in relation to this user. The same applies to the PC interface (i.e. enforceP method).

[28] This supposes a change in relation to the design where the policy removal could be realised in both the PCMCoreImpl (uninstallP method) and PFwCntImpl classes. In the current proof-of-concepts only the PFwCntImpl can remove policies from the DB. This has been done for sake of simplicity and clarity.

applications, users, higher-level Policy Consumer components). In particular, the enforcement result information message is sent to any higher-level Policy Consumer registered in the current MANBoP instance. Only if there are no higher-level Policy Consumers registered, it forwards the information to any generic receiver[29] registered. Finally, if no generic receiver is registered neither, it forwards the policy result information to the Policy Editor component that will show it through the GUI.

Further information about these tasks can be found in the design chapter.

C PCM: PCMCoreImpl

a public boolean uninstallP(t_policyId, int)

Method offered in the PCM interface. It is used to request the removal of a policy from the system. This method can be called because of several reasons. These reasons are expressed in the integer received as parameter. This integer has four possible values: (0) explicit request from the owner, (1) due to the policy group processing method, (2) policy expiration and (3) to solve a conflict. Aside, the method receives also the identifier of the policy to be uninstalled.

The implemented logic for this method removes all configuration data related with the policy in either the MANBoP instance or the managed nodes. This configuration data ranges from policy reservations information, configuration data in the managed devices, conditions registered in the monitoring system, lifecycle information, etc. Further information about these tasks can be found in the design chapter.

The method returns a boolean that indicates whether all removal tasks could be developed successfully.

D PCM: PCCCnt

a protected boolean uninstPR(t_policyId, boolean)

This is a component internal method offered by the PCCCnt class to request the removal or update of policy resource configuration from the PCC component and the DB. In this case the PCCCnt acts as interface with the PCC component and simply forwards the request to it. The method is defined with two arguments; the first one is the identifier of the policy whose resource information must be removed. The second argument is a boolean that indicates if the resource information should be updated or removed. The resource information must be updated only when the policy is either enforced

[29] Any application must implement the GenRecv (Generic Receiver) interface to be able to receive policy enforcement result reports. This interface is described at the end of the section.

or de-enforced. If the policy is removed, the resource information linked with this policy is also removed.

The method returns a boolean that indicates whether all tasks could be developed correctly.

E PCM: PGES: be_PGES

a protected static int last(Policy, Group, int)

All classes implementing a PGES must offer this method. It implements the functionality to decide if the group policy that has just been enforced is the last one or not. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing the policy enforcement result information. This integer contains the same information as the integer included within the Result type detailed before.

The implemented logic for this method calculates if the enforced policy is the last one according to the best effort execution strategy. That is, the method is completed only if all group policies have been enforced.

The method returns the next step to be taken. Possible values are: 2 this is the last policy of the group and therefore it is completed, 1 the groups is not completed because some policies must be de-enforced and 0 the group is not completed because some policies must be enforced.

F PCM: PGES: first_PGES

a protected static int last(Policy, Group, int)

The *last* method implements the functionality to decide if the group policy that has just been enforced is the last one or not. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing the policy enforcement result information.

The implemented logic for this method calculates if the enforced policy is the last one according to the first correct execution strategy. That is, the method is completed when a correct enforcement result is received.

The method returns the next step to be taken. Possible values are: 2 this is the last policy of the group and therefore it is completed, 1 the groups is not completed because some policies must be de-enforced and 0 the group is not completed because some policies must be enforced.

b protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy

Most of classes implementing a PGES must offer this method; only those with an obvious behaviour are not implemented. For example, the best effort execution strategy does not care about policy enforcement results because it will continue to forward group policies as they are received. It implements the functionality to continue the policy group execution based on the policy enforcement result received. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing the policy enforcement result information as has been described previously.

The implemented logic for this method calculates the group policies that must be either enforced or de-enforced to continue the policy group execution. In the case of this execution strategy, if the group is not finished, it returns the next policy following the group order in case it has been received.

The method returns the group positions of those policies that must be either enforced or de-enforced to continue with the group execution. In case an error occurs while realising these tasks, the method throws an UnProcessablePolicy exception.

G PCM: PGES: seqNACK_PGES

a protected static int last(Policy, Group, int)

In this PGES the *last* method implements the functionality to decide if the group policy that has just been enforced is the last one or not. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing the policy enforcement result information.

The implemented logic for this method calculates if the enforced policy is the last one according to the sequential, not confirmed, execution strategy. That is, the method is completed when the last policy in sequential order has been enforced.

The method returns the next step to be taken. Possible values are: 2 this is the last policy of the group and therefore it is completed, 1 the groups is not completed because some policies must be de-enforced and 0 the group is not completed because some policies must be enforced.

H PCM: PGES: seqACK_PGES

a protected static int last(Policy, Group, int)

In this Policy Group Execution Strategy the *last* method implements the functionality to decide if the group policy that has just been enforced is the last one or not. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing the policy enforcement result information.

The implemented logic for this method calculates the group policies that must be either enforced or de-enforced to continue the policy group execution.

The method returns the next step to be taken. Possible values are: 2 this is the last policy of the group and therefore it is completed, 1 the groups is not completed because some policies must be de-enforced and 0 the group is not completed because some policies must be enforced.

b protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy

For this PGES the *resrecv* method implements the functionality to continue the policy group execution based on the policy enforcement result received. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing policy enforcement result information.

The implemented logic for this method calculates the group policies that must be either enforced or de-enforced to continue the policy group execution. In the case of this sequential, confirmed, execution strategy, if the group is not finished, it returns the next policy following the group order in case it has been received.

The method returns the position within the group of the policies that must be either enforced or de-enforced to continue with the group execution. In case an error occurs while realising these tasks, the method throws an UnProcessablePolicy exception.

I PCM: PGES: atomic_PGES

a protected static int last(Policy, Group, int)

The *last* method for the atomic PGES implements the functionality to decide if the group policy that has just been enforced is the last one or not. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing policy enforcement result information.

The implemented logic for this method calculates if the enforced policy is the last one according to the atomic execution strategy. That is, the method is completed only when either all group policies have been enforced correctly or all group policies are uninstalled.

The method returns the next step to be taken. Possible values are: 2 this is the last policy of the group and therefore it is completed, 1 the groups is not completed because some policies must be de-enforced and 0 the group is not completed because some policies must be enforced.

b protected static t_order[] resrecv(Policy, Group, int) throws UnProcessablePolicy

The *resrecv* method for this PGES implements the functionality to continue the policy group execution based on the policy enforcement result received. The method is defined with three parameters. These are the Policy IMO representing the policy that has just been enforced, the Group IMO containing group information and an integer containing policy enforcement result information.

The implemented logic for this method calculates the group policies that must be either enforced or de-enforced to continue the policy group execution. In the case of the atomic execution strategy, if the group is not finished, it returns either the next policy following the group order (in case it has been received) when the result of the previous policy enforcement is correct, or all previously enforced policies when the result of the previous policy enforcement is incorrect.

The method returns the group position of the policies that must be either enforced or de-enforced to continue with the group execution. In case an error occurs while realising these tasks, the method throws an UnProcessablePolicy exception.

J PE: PECoreImpl

a public void recvEnfResult(String, t_policyId, Result)

This method is offered by the "southern" interface of the Policy Editor component, i.e. that offered to the PCM component. The PE offers this interface defined in the GenRecv interface described at the end of this section, in order to be able to receive information about the enforcement of policies and policy groups. This information will be then shown in the GUI offered to the user.

The implemented logic for this method formats the received information so as to show it in the GUI offered to the user. The method is defined with three arguments. These are, the identifier of the policy enforced, the enforcement result and a string containing further information in the case there has been an enforcement error. The enforcement result is given as a Result type that contains an integer and an array of strings. The Result type has been already described at page 277. Further information about these tasks can be found in the design chapter.

K DmMs: DLgcImpl

a public boolean unregCond(t_policyId)

Method offered through the DmMs interface. It is used to request the removal of all monitoring-related information linked with a policy from the

DmMs component. The method receives the identifier of the policy whose monitoring information must be removed.

The implemented logic for this method removes all monitoring data related with the policy in either the MANBoP instance or the managed nodes. Further information about these tasks can be found in the design chapter.

The method returns a boolean that indicates whether all removal tasks could be developed successfully.

L DmMs: MMCnt

a protected boolean unregIS(String)

Internal method of the DmMs component used to request the removal of all monitoring-related information linked with a policy from the DmMs component, Monitoring Meter components and managed devices. The method receives the identifier of the Individual Statement whose monitoring information must be removed.

The implemented logic for this method removes all monitoring data related with the policy in either the MANBoP instance or the managed nodes. Aside, it updates the lifecycle information of the involved Monitoring Meter components accordingly. If any Monitoring Meter is no longer being used it is uninstalled from the system. Further information about these tasks can be found in the design chapter.

The method returns a boolean that indicates whether all removal tasks could be developed successfully.

M DB: ManagerImpl

a public PC getPC(String, String) throws DBObjectNotFound

This method is used to obtain from the Database a PC IMO representing one of the Policy Consumer components currently running with the MANBoP instance.

The method receives two strings used to uniquely identify the requested PC IMO. These strings are respectively the identifier of the nodeSet where the component represented by the requested IMO must run and the identifier of the component itself.

The method returns the requested PC IMO, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

b public boolean setPC(PC, String)

This method is used to store in the Database a PC IMO.

The method receives as parameters the PC IMO to be stored and a string containing the identifier of the nodeSet where the component represented by this IMO must run. It returns a boolean indicating whether the storage of the PC IMO could be realised successfully.

c *public boolean rmPC(String, String)*

This method is used to remove from the Database a PC IMO representing a Policy Consumer component no longer running within the system.

The method receives two strings used to uniquely identify the requested PC IMO. These strings are respectively the identifier of the nodeSet where the component represented by the requested IMO runs and the identifier of the component itself.

It returns a boolean indicating whether the removal of the PC IMO could be realised successfully.

N *DB: PolicyImpl*

a *public credential getPolicyUserCred(t_policyId) throws DBObjectNotFound*

This method is used for retrieving from the Database a user credential linked with a *t_policyId* structure (i.e. a policy identifier).

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy to which the user credential is linked and, hence, locate the credential within the DB.

The method returns the credential structure, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

b *public PRI getPRI(t_policyId) throws DBObjectNotFound*

This method is used for retrieving from the Database a user PRI IMO linked with a *t_policyId* structure (i.e. a policy identifier).

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy to which the PRI IMO is linked and, hence, locate the requested IMO within the DB.

The method returns the PRI IMO, in case it could be found and obtained from the Database. Otherwise, a DBObjectNotFound exception will be raised.

c *public boolean setPRI(t_policyId, PRI)*

Method used to store in the database a Policy Resource Information (PRI) IMO linked with a policy.

The method is defined with two arguments: the identifier of the policy to which the PRI IMO is linked and the PRI IMO itself.

The method returns a boolean indicating whether the object storage has been successful.

d *public int getPSts(t_policyId) throws DBObjectNotFound*

This method is used for obtaining from the Database the status of a policy.

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy. Once the policy is identified, the method logic retrieves the Policy IMO object from the DB and returns its status field. If the Policy IMO could not be found, a DBObjectNotFound exception will be raised.

e *public boolean modPStatus(t_policyId)*

This method is used for modifying the policy status of a Policy IMO stored in the Database.

The method receives a *t_policyId* structure as argument. This structure contains all information to uniquely identify the policy. Once the policy is identified, the method logic retrieves the Policy IMO from the DB, modifies its status field and stores it again.

The method returns a boolean indicating whether all tasks could be developed successfully.

f *public boolean removeP(t_policyId)*

This method is used for removing from the Database policy-related information. In particular, the XML Policy, the Policy IMO, the PRI IMO and the credential are all removed from the Database.

The method receives a *t_policyId* structure as argument. This structure contains information to uniquely identify the policy and the related information to be removed.

The method returns a boolean indicating whether all tasks could be developed successfully.

O *DB: PGroupImpl*

a *public boolean rmGroup(int, String)*

This method is used for removing from the Database group-related information. In particular, all group XML policies and Policy IMOs, policy identifier structures and linked credentials are removed from the Database.

The method receives two arguments to uniquely identify the group information to be removed. These arguments are an integer containing the policy group number and a string with the user name of the user that introduced the group.

The method returns a boolean indicating whether all tasks could be developed successfully.

P PC: XStringFields

XStringFields is an interface included within the Policy Consumer component package. It contains various string fields with text needed to create an XML policy. This allows all PC components needing to create XML policies, particularly those running over other MANBoP managers, to have the common XML strings easily available.

We do not include here the concrete fields contained in the interface since it would take too long and it would not provide any relevant information.

Q GenRecv

The GenRecv is an interface that must be implemented by all applications pretending to receive policy enforcement result reports. This interface defines one single method, described hereafter. This interface is implemented also by those framework components that might also receive policy enforcement result reports, e.g. Policy Editor and Policy Consumer components.

a public void recvEnfResult(String, t_policyId, Result)

This method allows the component implementing it to receive information about the enforcement of policies and policy groups. The component must also be registered at the Naming Services of all MANBoP instances from where it wants to receive these reports.

The method is defined with three arguments. These are: the identifier of the policy enforced, the enforcement result and a string containing further information in the case there has been an enforcement error. The Result type has already been described at page 277.

R Exceptions

a UnableToEnforceP

This exception is sent by either PC components or the PCCnt class within the PCM component when an error occurs while enforcing a policy.

The exception is defined with three fields: one string and two arrays of strings. The string is used to provide extra information about the cause of the error. The arrays contain, respectively, a list of node identifiers where the policy could be enforced successfully and a list of node identifiers where it could not be enforced.

This information is used by the PCCnt class that collects all possible exceptions raised by Policy Consumers and raises a new one with the recompiled information. Finally, the PCMCoreImpl class catches this exception and decides, based on the enforcement type field of the policy (see the Policy IM at pag. 220) whether the policy should be removed from those nodes where it was successfully enforced or not.

6th Adding a node to the managed topology

In addition to the policy-processing functionality just described, we have also implemented management functionality needed to modify the managed topology. More specifically, we have implemented the logic needed to add a new node to the managed topology.

In concrete, the public and protected methods implemented are detailed in the table below:

Component Name	*Class Name*	*Method Signature*
PCM	PCMCoreImpl	public void addN(String, String)
DB	TopologyImpl	public Link getLinkObj(String, String) throws DBObjectNotFound

Table 5 - 34. Methods implemented for the node addition mechanism

A PCM: PCMCoreImpl

a public void addN(String,String)

This method is used to add a new node on the managed topology. The method might be used by the network operator to update the managed topology with the addition of a new managed device.

The main tasks developed as consequence of calling this method are first, the creation of the corresponding IMOs and their storage within the DB. Second, the recalculation of the routing algorithm with the new information.

The information received as method parameters are two strings containing respectively, the path where the file describing the new managed topology, including the new resources, is located and the path where the file describing how to access the new node is located.

B DB: TopologyImpl.

a public Link getLinkObj(String, String) throws DBObjectNotFound

The getLinkObj method is used to obtain a Link IMO from the Database.

The method receives two parameters that uniquely identify the link. Both parameters are strings. The first string indicates the source node of the link.

The second string contains the identifier of the link. Such identifier is unique for links in the source node.

The method returns the Link IMO retrieved from the Database if any is found. Otherwise, a DBObjectNotFound exception is raised.

7th Domain-dependant components

The domain-dependant functionality implemented can be classified in three types of components: Policy Conflict Check (PCC) components, Monitoring Meter (MM) components and Policy Consumer (PC) components.

All implemented components from the same type share the same interface, the component interface; thereby, the implemented methods are exactly the same for all components of the same type. However, the functionality implemented inside the method is different depending on the component goal, the functional domain and the underlying devices.

Not all methods defined in the component's interface have been implemented. The table below summarises the interfaces implemented for the proof-of-concepts per component type.

Component Name	*Class Name*	*Method Signature*
PCC	PAnImpl	public boolean checkConfl(t_policyId)
	PCCCore	public static boolean findResources(Policy, t_prvp)
MM	MFactImpl	public boolean[] monIS(String, t_simpleCond, credential) throws MonitoringError
		public boolean sMonIS(String)
		public void recvMonResult(String, t_policyId, boolean)
PC	Mapper	public Result enforceP(Policy, int, credential, String[], int) throws UnableToEnforceP
		public void recvEnfResult(String, t_policyId, Result)

Table 5 - 35. Domain-dependant methods implemented

In the following paragraphs we will extensively describe the logic implemented for each of these methods and for all implemented functional domain components.

A PCC: PAnImpl

a public boolean checkConfl(t_policyId)

This method is used to initiate the conflict checking mechanism. Such mechanism verifies that no inconsistencies exist between the new policy and the previously introduced policies. This method is called during the policy-processing tasks by the Policy Consumer Manager component to be sure that there will be no inconsistencies before continuing the process.

The only argument received in the method is the identifier of the new policy against which conflict checks must be realised. The method returns a boolean indicating whether there was any conflict.

The method has been implemented just for the network-level PCC: the *PCC_0* component.

The logic implemented simply obtains the corresponding Policy IMO from the Database based on the policy identifier received and extracts from it the policy validity period. With this information, it contacts the *findResources* method from the *PCCCore* class, described in the following sub-section, to request the conflict checks.

B PCC: PCCCore

a public static boolean findResources(Policy, t_prvp)

The *findResources* method is called by the PAnImpl class for requesting the resources needed by a policy either for an established interval or at enforcement time and verifying that there are no inconsistencies with other policies.

The method has two input parameters. The first one is the Policy IMO while the second is the times during which the policy will be enforced.

The result is a boolean that indicates if any conflict was found.

As the *checkConfl* method just described this method has been implemented just for the network-level PCC. The implemented logic finds out all managed devices involved in the policy and verifies that these devices exist within the managed topology.

C MM: MFactImpl

a public boolean[] monIS(String, t_simpleCond, credential) throws MonitoringError

The *monIS* method is called by the DLgcImpl class to request the monitoring of an Individual Statement (IS) from a policy condition.

The method is defined with three input parameters. The first one is a string containing the unique identifier of the IS to be monitored. The simple condition with the IS information is included next. Finally, the last parameter is the credential of the user requesting the monitoring. Such credential is used to monitor the resources with the user's access rights.

Additionally, the method throws a MonitoringError exception if a problem occurs during the monitoring of a policy. This exception contains a field providing more details about the cause of the problem. The exception will be caught by the DLgcImpl class, which will react in accordance.

This method has been implemented in all Monitoring Meter components developed for the proof-of-concepts. In the following paragraphs we will explore the concrete logic in each of these implementations.

The logic implemented in the BWMM_0_ABLE component for this method monitors the used bandwidth in an ABLE router interface. It does this by creating, compiling and sending an active packet for the ABLE router based on the information contained in the simple condition received. Additionally, it waits for value changes in the simple condition evaluation coming from the active packet in the ABLE router. These value changes are sent via a socket created at a particular port. When the IS (i.e. simple condition) evaluated value changes, it contacts the DLgcImpl class to report the new value. When this new value changes the global policy condition evaluation the policy enforcement is triggered.

The logic implemented for the BWMM_1_ABLE component, the same Monitoring Meter component but for the element-level, is basically the same. The only difference is that the network-level BWMM_0_ABLE component has the possibility of monitoring several ABLE nodes, while the BWMM_1_ABLE, for obvious reasons, does not contemplate this possibility.

The third BWMM component implemented is the BWMM_2_MANBoP component. This component is the one installed at the network-level manager when working over element-level managers. The logic implemented for the *monIS* method in this component creates an element-level monitoring policy based on the simple condition information and forwards it to the appropriate element-level MANBoP managers. Additionally, it registers as higher-level Monitoring Meter component at the Naming Services of these element-level stations so that it is informed when the condition is met.

Other MM components developed that implement this method are the TimeMM components. These components have been implemented for the scalability evaluation scenario (described at the end of the chapter). The TimeMM_0_CISCOX and TimeMM_1_CISCOX components have been implemented with exactly the same logic. The X stands for the number of the simulated type of CISCO router being monitored starting from 2601 (see page 309 for more information). Initially, the logic contained in these components realised the monitoring of the used bandwidth in a CISCO router interface during an established period of time. The time used in the scenario was big enough to last all along the scenario. Additionally, independently on what the monitored value was they do not report that the condition is met. They polled the value every five seconds. Nonetheless, the logic implemented at last, just simulated this behaviour.

The TimeMM_2_MANBoP component implemented contains exactly the same logic as the BWMM_2_MANBoP component for this method. It creates the element-level monitoring policy information based on the condition information received, forwards the monitoring policy to the appropriate element-level stations and registers at the Naming Services running at these stations as higher-level monitoring meter.

b public boolean sMonIS(String)

This method is used by the DLgcImpl class to request the end of the monitoring of an Individual Statement.

The method is defined with a single parameter, which is the identifier of the IS that should no longer be monitored. The method returns a boolean indicating whether the requested action could be realised successfully or not.

Only the BWMM_0_ABLE and BWMM_1_ABLE components implement this method. In both cases the logic is the same. They both stop the active packet running in the ABLE node, close the socket through which they received condition value changes and exit the method.

c public void recvMonResult(String, t_policyId, boolean)

The *recvMonResult* method is used to report monitored values into a higher-level Monitoring Meter component.

Three parameters are defined. The first parameter is an integer containing the identifier of the target node from where the reported monitored value has been obtained. The policy identifier of the monitoring policy that causes, with its enforcement, this report is the second parameter. Finally, a boolean is included as the last parameter. This boolean contains the new evaluated value for the monitored Individual Statement.

Among the Monitoring Meter components implemented for the proof-of-concepts, just the BWMM_2_ABLE component implements this method. The logic contained at the component informs the DLgcImpl class that the Individual Statement is met if there is only one monitored node, or if there are many monitored nodes and either all monitored values meet the condition or the condition establishes that just one node meeting the condition suffices. The BWMM_2_ABLE component contains a table that maps policy identifiers of created monitoring policies with the Individual Statement identifiers that are being monitored with those policies.

D PC: Mapper

a public Result enforceP(Policy, int, credential, String[], int) throws
UnableToEnforceP

The *enforceP* method is used by the Policy Consumer Manager component to start the enforcement of a policy in one or more target nodes. More specifically, this method translates policy action values into commands understandable by the underlying device, configures the device with these commands and returns the result.

The method expects five input parameters. The first one is the Policy to be enforced. Second, an integer containing a sequence number that is used to differentiate between the enforcement of the same policy at different target nodes of different types, that is at different nodeSets. Third, the credential of the user is included to enforce the policy with the user's access rights. The fourth parameter is an array of strings. Each string contains a node identifier of a device where the policy must be enforced. Finally, an integer containing the number of the action among all actions included in the policy that must be enforced by the component.

The method defines a Result type as return parameter. This type contains an integer indicating whether the policy enforcement was successful and an array of strings that might include further information related with the policy enforcement.

If an unexpected error occurs while enforcing the policy the method throws an UnableToEnforceP exception. This exception will provide more information about the cause of the problem and, optionally, the target node where the problem occurred.

All Policy Consumers must implement this method. For the proof-of-concepts we have implemented several PCs. In the following paragraphs we will explain the logic included in this method for each of them.

The first Policy Consumer implemented in the proof-of-concepts is the QoSPC_0_FAIN. The logic for the *enforceP* method included in this component mainly creates or activates a Virtual Environment (VE) for a service provider over one or more FAIN nodes [FAIN03d] based on the parameters received in the policy. Furthermore, the specified computational and forwarding resources are allocated to the created VEs. To carry out this functionality the component uses the facilities that FAIN nodes offer. If the enforcement is successful the component includes in the Result type returned two strings containing respectively, the Virtual Environment identifier assigned by the node and the Virtual Network identifier that came in the policy itself.

The QoSPC_1_FAIN component contains almost the same logic as the QoSPC_0_FAIN component, as it is the element-level component from the same functional domain. The main difference is that the mapping of policy action values to FAIN node commands is slightly faster since the received action values are more detailed than network-level ones. For the rest, the logic contained in this method is mainly the same.

The logic implemented for the method in the QoSPC_0_CISCO2600 component is oriented to the modification of the routing tables of CISCO2600 routers [CISCO2600]. The component maps the received policy action values into CLI commands introduced in the router via telnet. More specifically, the component receives a flow specification, a list of target nodes to be managed and a list of next hop addresses. Then, the component accesses each of the target nodes and modifies the routing table with the new next hop address for that flow. Finally, the component returns the Result type with the global enforcement result on all target nodes.

The element-level QoSPC_1_CISCO2600 Policy Consumer component contains almost the same logic as its network-level equivalent just described. The only difference is that in this case only the CISCO2600 router managed by the element-level manager where the component runs is configured. Hence, the component creates the corresponding CLI commands and sends them to the router to add the new next hop address for the specified flow. It returns the result of the policy enforcement on the managed node.

The QoSPC_2_MANBoP is the last component for the QoSPC functional domain that has been implemented in the proof-of-concepts. This component is installed when needed, at the network-level station when running over element-level MANBoP managers. The method logic implemented translates all network-level QoSPC functional domain policies defined for the proof-of-concepts into the corresponding element-level QoSPC policies. Once the policies are created it forwards them to the appropriate element-level managers to be processed. Additionally, it registers at the Naming Services for these element-level managers as receiver of policy enforcement results. Finally, it returns an unspecified result value to avoid having the PCM blocked waiting for the enforcement result. The real enforcement result value will not be known until element-level policies are enforced and inform the component of the enforcement result via the *recvEnfResult* method that will be described in the following sub-section.

The DelegationPC_0_MANBoP and DelegationPC_1_MANBoP have been implemented with the exactly the same logic in this method. Both components contain logic for registering new users in the management station and for creating restricted functional domains for these users. Particularly, they look at the action type to be enforced. When it is a *newUser* action, they create the new User IMO in the Database with the information received in the policy. They also create and store the Schema IMO that allows the user to access the restricted functional domains specified in the policy. When the policy action is a *FDRestriction* action, the components first obtain the user name and password from the policy conditions. Then, they verify that they are correct. Afterwards, they obtain the permitted restricted functional domains and check that the functional domain to be restricted is among the ones permitted. If so, they download the XML Schema to be restricted and finally, modify the XML Schema to introduce the restrictions. Finally, they return the enforcement result.

The DelegationPC_2_MANBoP component in addition to the logic just described for the other DelegationPC components implements the logic for translating network-level delegation policies into element-level ones. The translation is carried out only if the delegation policy applies at the element-level. Translated element-level policies are forwarded to the corresponding element-level stations where they will be enforced. Additionally, the component registers at the Naming Services of these stations as receiver of enforcement result reports. Finally, the component returns the unspecified result value.

The method logic implemented in the ServicePC_0_FAIN component deals with the deployment and configuration of active services in FAIN nodes. In particular the active services supported are those used for the proof-of-concepts: the duplicator and the transcoder active services. When the policy action is a *ServiceDeployment* action the component contacts the ASP system [FAIN03c] in the FAIN active node to request the deployment of the active service with the appropriate parameters. These parameters are calculated taking into account the policy action values. Then, the component stores the IOR of the deployed service and returns it with the correct enforcement Result type. When the policy action is a *ServiceConfiguration* action the component uses the IOR of the component to configure it based on the received action values. If the configuration is successful, a correct enforcement result is returned.

The ServicePC_1_FAIN component carries out almost the same functionality. The main difference with its network-level equivalent is that configuration parameters received in the policy action are much more detailed and hence, the translation of these parameters into service parameters is a bit faster.

In the case of the ServicePC_2_MANBoP component, the logic included in the method is the translation logic. This component translates all network-level service policies supported in the scenario into element-level ones. Then, it forwards the element-level policies to the corresponding element-level stations. Finally, the component registers at the Naming Services running in that element-level stations as receiver of enforcement result reports and returns the unspecified result value.

The last Policy Consumer component implemented for the proof-of-concepts is the MonPC_1_ABLE component. This component enforces monitoring policies introduced by higher-level Monitoring Meter components. The logic implement for this method in the component is oriented towards informing the corresponding network-level MM component of a new value in a monitored condition. More specifically, the component looks in the Naming Service if the MM component to be informed is registered. If so, it uses the *recvMonResult* method from that MM to inform it about the new monitored condition value in the managed device. Afterwards, and if no error occurs, the component simply returns a successful enforcement result.

b *public void recvEnfResult(String, t_policyId, Result)*

This method is called by lower-level MANBoP managers to inform about the enforcement result of a policy. More specifically, this method receives the enforcement result from all underlying devices where a policy has been enforced and based on these results it constructs the global enforcement result. Then, it informs the PCM of the final enforcement result of the policy.

The method has been defined with three parameters. The first parameter is a string containing further information if an error has occurred during the policy enforcement. Then, the policy identifier of the policy being enforced is also introduced. The last parameter is the Result type containing the enforcement result and any further information of interest.

Three Policy Consumer components among those implemented for the proof-of-concepts include logic for this method. These components are the QoSPC_2_MANBoP, DelegationPC_2_MANBoP and ServicePC_2_ MANBoP components. In all three cases the implemented logic in this method is exactly the same. For every enforcement result received they add the result information into the global result. Just one incorrect enforcement value in any of the element-level enforcement results received sets the global enforcement result value to incorrect enforcement. Also, the method logic compares the number of enforcement results received with the number of element-level managers to which the policy has been forwarded. When all enforcement results have been received, the PCM component is informed about the global enforcement result.

Section V.5 – Proof-of-concepts demonstration description

1st Introduction

In order to evaluate the proof-of-concepts implementation just described, we have envisaged two scenarios that run over testbeds prepared in our laboratory. Each of these two scenarios is targeted towards an evaluation goal. More specifically, the first scenario is oriented towards assessing the functional requirements of the system: the system flexibility, delegation mechanism, etc. On the other hand, the second scenario is targeted to assess the system scalability.

Along this sub-section we will describe in detail both scenarios as well as the testbeds where they are going run. We will first describe the scenario targeted towards evaluating the functional requirements and then the scenario prepared for the scalability assessment.

2nd Functional assessment scenario (first scenario)

A Description

The scenario starts when a service provider contacts a network operator because he wants to offer to his customers a webTV service. The webTV service consists on the broadcasting of a video program in the Internet that customers are able to watch irrespectively of their terminal capabilities.

Customers interested in the service will subscribe directly to the service provider.

The service provider needs from the network operator a Virtual Active Network (VAN), containing computational and forwarding resources, wherein he can deploy services that may be customised to meet customer requirements.

The service provider and the network operator negotiate a Service Level Agreement (SLA) that establishes the exact resources that will be assigned to the service provider and where will them be assigned. These resources are:

i. Two Virtual Environments in two different FAIN active nodes [FAIN03d]. The computational resources assigned to the VEs must be enough to run the service provider's active services.

ii. The SLA also establishes the allocation of certain forwarding resources enough for the transmission of the video data with an acceptable quality.

iii. To guarantee the forwarding resources assigned to the service provider the network operator assigns an ABLE router [ABLE] to monitor the used bandwidth, if a threshold is exceeded the traffic from the service provider is routed through a backup route.

iv. The service provider is allowed to introduce some types of service policies into the management system with which it will install and configure his actives services freely.

v. The active services from the service provider will be registered and stored in the active service repository owned by the network operator.

Once the SLA negotiation is concluded, the network operator must configure his managed network to fulfil all points of the agreement. The configuration is executed by enforcing five policies, four of them are grouped in a policy group that must be enforced atomically. These four policies will create the VAN for the service provider and delegate the management functionality (i.e. restricted types of service policies) to the service provider. The VAN creation encompasses the creation and activation of the VE in the two FAIN active nodes, the delegation of the restricted service management functionality and the reservation of the necessary resources. The fifth policy is processed individually. This policy is responsible of guarantying the forwarding resources[30]. It re-routes the service provider flows through a backup route when a threshold is exceeded in the initial route. However, when the network operator tries to introduce the policy, the management framework, more specifically the Policy Conflict Check component, will refuse the enforcement

[30] This policy is enforced over a CISCO router. For this reason many times along the document the reader might see references to this policy as: *'the policy enforced over a CISCO router in the scenario'*.

of this policy. The reason is that one of the nodes in the backup route is not registered as part of the managed network inside the management system. This part of the scenario has been included exclusively to demonstrate the capability of the system of adding and removing managed elements dynamically. The network operator will add the node in the managed topology with an administrative command and introduce the policy again. This second time the policy is introduced successfully. Finally, the network operator registers and stores the service provider's active services in the service repository.

When the managed network and the management framework are both configured as established in the SLA, the service provider will start the creation of the webTV service.

In the scenario, the service provider has two customers that have previously subscribed to use this service. Hence, the service provider creates the webTV service to satisfy these two customers. One of the customers has subscribed for the reception of the video in MPEG format while the second one wants to receive it in H.263 format. Therefore, the service provider will create a service topology as the one shown in the figure below.

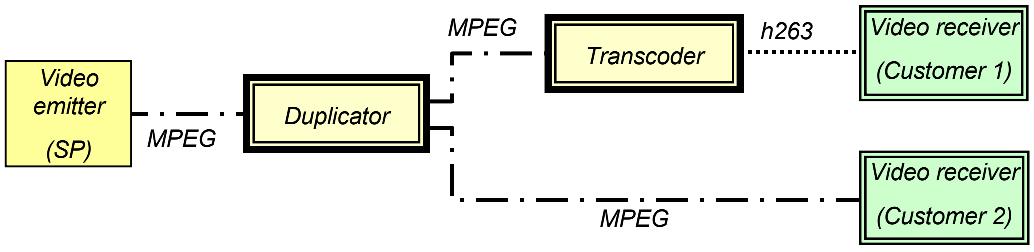

Figure 5 - 4. WebTV Service topology

To create such a service topology the service creator will introduce a policy group of service policies that must be enforced atomically. The policy group consists of four policies. Two policies will be enforced in the FAIN active node that is closer to the video emitter. These policies will serve to request the deployment and configure the duplicator active service. The duplicator active service receives the video data, duplicates it and forwards it to two destinations specified. The other two policies will be enforced in the second FAIN node to request the deployment and configure the transcoder active service. The transcoder active service receives video data in MPEG and codes the video information with a new video format, in this case H263. Both active services will be deployed from the active service repository owned by the network operator.

Once everything is set up and running both customers are capable to see the video in their respective format. The bandwidth is also monitored periodically to assure the video quality.

At this point, interfering traffic starts flowing through the same links as the video flow to Customer 2. Soon, the threshold of the video quality assurance is reached. This causes the enforcement of the policy that will re-route the service provider flow avoiding the interfering traffic.

The process just described will be carried out two times, each time being one half of the whole scenario. During the first half of the scenario, the first time the above process is carried out, the management infrastructure will consist only on a MANBoP network-level station. In the second half, the management infrastructure will be formed by a MANBoP network-level station working over MANBoP element-level managers. There will be one element-level manager per managed device.

The objective of running the process described two times is showing the system's capability of simple creation of different management infrastructures. Furthermore, we will analyse and compare the management system behaviour with both management infrastructures.

In the following sub-section we will go into details and describe this scenario as a sequence of steps.

B Steps

a Step 1: Creation of the Virtual Environments

This step encompasses the enforcement of the first policy from the VAN creation policy group. This policy is enforced in the two FAIN active nodes and causes the instantiation of a Virtual Environment as well as the allocation of the corresponding resources to the service provider in each node.

The policy group information carried inside the policy specifies that the policy group must be processed at the network-level. Hence, the other group of policies are stored at the network-level station until the correct enforcement of this policy is received.

The policy has no condition that needs monitoring, hence it is enforced directly. The policy action specifies the type of Execution Environment (EE) that will be instantiated (in this case a JVM), the quality of service class assigned to the service provider and the identifier for the VAN. There are three levels of QoS: bronze, silver and gold.

The policy is enforced by the QoSPC component running inside the corresponding active node. This QoSPC will map the received policy action attributes to node interface commands to carry out the requested action. Since this is the first QoS domain policy processed in the scenario, the download and installation of the QoSPC will be requested on each FAIN active node where it must be enforced. Additionally, when the management system infrastructure is that of a network-level over element-level managers, a QoSPC component realising policy translation between QoS network-level policies and QoS element-level policies will also be installed at the network-level management system. Furthermore, the Policy Conflict Check component (PCC) will also be dynamically installed in all those management stations where the policy is processed. The PCC component is always installed with the first policy processing in a management station and every time a newer version is needed.

b *Step 2: Registration of the Service Provider*

If the previous step is completed successfully and the enforcement confirmation is received at the network-level station, the second policy from the VAN group is processed.

This second policy is a delegation policy that will register the service provider as authorised user of the management system and grant him access to the restricted functional domain that will be created with the next policy.

The delegation policy action contains a field called *'Applies'* that determines where the delegation must be done; i.e. at network-level, at element-level or both. In this case the delegation policy specifies that the functionality must be delegated at both levels, obviously just when both levels are used in the management infrastructure created.

Other policy action fields included in the policy are the user name and password from the service provider and the restricted functional domain to which it will have access.

The policy has no conditions, therefore it is enforced directly on those management stations within the management infrastructure where the service provider is allowed to use such functionality. The policy is enforced by the DelegationPC component. Again, this policy is the first delegation policy processed in the scenario. Thereby, the processing of this policy causes the dynamic download and installation of the DelegationPC component at every management station within the management infrastructure. The actual enforcement of the policy causes the creation of the appropriate user information in the management station's Database.

c *Step 3: Delegation of a restricted Service Functional Domain*

Once the service provider is correctly registered in the system, the processing of the third policy of the group begins. This policy will create the restricted XML Schema assigned to the service provider in each management station where the functionality has been delegated to him.

The policy condition need not be monitored. The two policy conditions included are the service provider's user name and password.

The policy action includes those policy fields that must be restricted. In this case, the policy fields restricted are the allowed target nodes, the list of actions from the service domain permitted and two policy action fields. These two policy action fields are the VAN name which is restricted to be exclusively the name of the service provider's VAN and the active services names that might be used, which are restricted to *'duplicator'* and *'transcoder'*.

The policy is enforced by the DelegationPC component on those management stations within the management infrastructure where the service provider has obtained the delegated functionality.

d Step 4: Activation of the Virtual Environments

When the enforcement confirmation from the previous policy is received the last policy of the group begins its processing. This policy will activate the Virtual Environment previously created for the service provider and definitively allocate the corresponding resources. This policy is included as the last of the policy group to avoid taking network resources before time and before being sure that the other group policies will be enforced correctly.

The policy includes no conditions needing monitoring. Thus, the policy is enforced directly by the QoSPC component running in the FAIN active nodes. The policy action fields are the same as those included in the Virtual Environment creation policy.

Only when the enforcement confirmation of this policy is received, the last one from the group, the user is informed about the correct enforcement of the group. If any of the group policies would have failed, previously enforced policies would have been removed and the user would have been informed about the impossibility of correctly enforcing the group.

At this point the service provider can start using and configuring his VAN. The processing of the quality assurance policy has been moved to later steps on the scenario in order to keep all quality assurance steps together. Moreover, the policy would not be needed until then.

e Step 5: Deployment of the duplicator active service

To prepare his Virtual Active Network (VAN) to offer the WebTV service, the service provider will introduce a policy group in the management system. This policy group is formed by four service policies that will create the service topology shown in Figure 5 - 4. The four service policies will be authorised with the service provider's access rights. That is, they will be validated against the restricted XML-Schema.

The first group policy will be processed immediately. This policy will request the deployment of the duplicator active service in the FAIN active router that is closer to the video emitter.

The policy contains no condition, so it will be enforced immediately. As long as the policy is not refused in the authorisation process.

The policy action is a '*ServiceDeployment*' action permitted to the service provider. The policy action has many fields. The first field is the name of the service provider's VAN where the active service must be deployed. This field is restricted to the VAN identifier of the service provider's VAN. The second action field is the name of the active service that must be deployed, in this case *'duplicator'*, which is one of the two allowed values. Finally, the last field is the identifier of the EE where the active service must be installed, in the policy the value is *'JVM'*. This last field is not restricted, however at VAN creation's time the only EE specified for that VAN was the JVM.

The policy is enforced by the ServicePC running on the corresponding FAIN active node. As this is the first policy from the service domain enforced in the scenario the download and installation of the ServicePC will be requested for correctly enforcing the policy. The ServicePC components will be installed at those FAIN active nodes where the policy is going to be enforced. Additionally, a ServicePC translating service network-level policies into service element-level policies will be installed at the network-level station when the management infrastructure is that of a network-level over element-level managers.

The enforcement of the policy is done by mapping the policy fields into commands send to the Active Service Provisioning (ASP) facility implemented in FAIN [FAIN03c].

f *Step 6: Deployment of the transcoder active service*

As soon as the correct enforcement confirmation of the previous policy is received the processing of the second service group policy begins. This policy requests the deployment of the transcoder active service inside the second FAIN active node.

The policy is validated against the service provider's restricted XML Schema. If the authorisation process does not refuse the policy its processing continues normally.

The policy is very similar to the previous one. As the previous one it has no conditions, which causes that the enforcement process is started immediately.

The policy action fields are also very similar to those in the previous policy, the only change is that the service name field value is *'transcoder'* instead of *'duplicator'*.

The enforcement of the policy causes the dynamic installation of the ServicePC component that will enforce the policy at the second FAIN active node. This Policy Consumer (PC) component was not installed as a consequence of the previous policy enforcement simply because it was not needed, as this FAIN active node was not contacted as result of the previous policy enforcement. As in the previous policy, the enforcement is done by contacting the FAIN ASP system available in the node.

g Step 7: Configuration of the duplicator

The reception of the correct enforcement confirmation from the previous policy starts the processing of the third service group policy. This policy will configure the duplicator with the appropriate information according to the service topology that is being created.

This service policy has no conditions either. Hence, as long as the policy is not refused by the authorisation mechanism the policy enforcement process is started immediately.

The policy action of the policy is a *'ServiceConfiguration'* action, which is one of the two action types delegated to the service provider. The action fields are first, the identifier of the VAN where the service to be configured is located. The only allowed field is the identifier of the service provider's VAN. The second action field is the named of the service to be configured. In this policy the service name is obviously *'duplicator'*, which is also allowed. Finally, the rest of the policy action fields are *'ConfigurationInfo'* fields. A service configuration action might have from zero to, potentially, an infinite number of *'ConfigurationInfo'* fields. These fields are service-specific as they contain the parameters that actually configure the service. In this case, the policy action contains five *'ConfigurationInfo'* fields providing information about the address of the video emitter and the addresses of the two destinations of video data.

The policy is enforced by the ServicePC component running at the corresponding FAIN active node. This component keeps the IOR (CORBA Object reference) of all active services that it has installed within the active node. Therefore, the ServicePC component uses the IOR from the transcoder to contact it directly and introduce the parameters contained in the policy. If a service provider wants its active service to be managed with the MANBoP management system, that active service must offer a common interface. However, the network operator might wish to create a special Policy Consumer component for managing one particular active service, with a different interface, owned by the operator himself or even from by one of his service providers, but it will not usually be the case.

h Step 8: Configuration of the transcoder

Finally, if the duplicator active service could be configured correctly, the processing of the last service group policy is started. This policy will configure the transcoder active service with the necessary parameters to fit the service topology being created and the '*Customer 1*' requirements.

As the rest of service group policies, the policy contains no conditions. If the authorisation is successful, the policy enforcement begins.

The policy action fields are almost the same as in the previous policy. The only difference is in the *'ServiceName'* field, which in this case is *'transcoder'* and obviously the *'ConfigurationInfo'* fields. The configuration information contained in these fields tells the transcoder the source and destination of the video data as well as the video and audio quality of service classes. In this case, the video source is the address of the duplicator, while the destination of the coded video is the customer's address. Finally, the video and audio quality of service classes have three possible values: bronze, silver and gold.

The policy is enforced by the corresponding ServicePC in the appropriate FAIN node. The ServicePC will use the IOR of the transcoder service to contact it and configure it with the information contained in the policy.

The confirmation of the correct enforcement of the policy causes that the management system informs the user of the successful enforcement of the service policy group.

At this point, the webTV video service is ready and both customers are able to see the video channel in the respective coding formats.

In the following scenario steps, we will describe the quality of service assurance part of the scenario.

i Step 9: Quality assurance policy refused

Once the service is up and running the only missing part of the scenario is that of assuring the bandwidth QoS to the service provider. In order to do that the network operator introduces in the management system a policy that monitors the throughput in an ABLE node that is part of the VAN assigned to the service provider. When a certain threshold is reached the policy is enforced causing the service provider traffic to be re-routed through a backup route. However, to test the MANBoP framework's capability of dynamically adding or removing managed elements to the managed topology, one of the nodes in the backup route has not been included within the managed topology when the management system is booted. In consequence, when the network operator tries to introduce the quality assurance policy the management framework refuses the policy introduction with the following error message: *'One or more nodes involved in the policy are not included within the managed network topology'*.

It is the PCC component that when realising the static conflict checking detects the conflict and refuses the policy processing with that error message.

The network operator must first register the new managed device in the managed network to be allowed to introduce the quality assurance policy.

j Step 10: Addition of a node to the managed topology

This step is done by the network operator via an administrative command supported by the Policy Consumer Manager (PCM) component. With administrative we refer to the fact that it is a command oriented to the administrator of the network, usually the network operator. This command is implemented in the *addN* method that has already been described in Section

V.4 – Implemented Code. This method receives as parameters the new managed topology information and the information to correctly access the new underlying device (either the managed device or its correspondent element-level manager).

After requesting the managed device addition in the network-level station and all element-level managers (when they are included in the management infrastructure) managing neighbouring devices to the one added, the network operator might try again the introduction of the quality assurance policy.

k Step 11: Quality assurance policy accepted

At this point the network operator introduces the quality assurance policy in the management system and this time it is not refused by the conflict check component.

The policy contains a policy condition that needs to be monitored to decide when the policy must be enforced. This policy condition is an interface bandwidth condition (*'IFBWCond')* that establishes a threshold. When the threshold is reached the condition is met and the policy enforcement process will be triggered. The policy condition value field contains four strings that contain information for doing the monitoring correctly. These four strings indicate the IP address of the router being controlled by the ABLE node that must be monitored, the router type (in this case a CISCO 7200 router [CISCO7200]), the interface that must be monitored and the bandwidth threshold.

To realise the monitoring the BWMM component is dynamically downloaded and installed in the system. This Monitoring Meter (MM) when the management infrastructure is formed by a network-level manager only, will directly realise the monitoring by sending an active packet to the ABLE active router.

When the management infrastructure is that of network-level over element-level managers, the BWMM component is also installed at the network-level station. However, this time the BWMM running at the network-level creates an element-level monitoring policy with the policy condition information received. This monitoring policy is forwarded to the corresponding element-level manager that will realise the monitoring through the BWMM component installed at the element-level.

l Step 12: Re-routing of service provider's flows

At this point we will boot a traffic generator that will start sending packets through the monitored interface. This will cause that soon the threshold established by the policy condition is reached.

When the management infrastructure consists of a network-level station only, it is directly the BWMM running at this station the one informed by the active code running in the ABLE router that the condition is met.

On the other hand, when the management infrastructure is formed by network and element-level managers, the active code running at the ABLE router will inform the BWMM of its attached element-level manager that the condition is met. This will cause the enforcement of the element-level monitoring policy. A Monitoring Policy Consumer (MonPC) that is dynamically downloaded and installed within the element-level station will enforce this policy. The enforcement of the policy is simply informing the network-level BWMM that the condition is met.

Once the BWMM knows that the quality assurance policy condition is met it triggers the policy enforcement that will cause the re-routing of the service provider's flows through the backup route. The enforcement of the policy will be done by the QoSPC component.

When running a network-level only management infrastructure the management station is running directly over the managed devices and hence all dynamically installed components interacting with the managed devices are specific for that type of devices. Up to know only QoSPC components for FAIN routers have been installed and they are running on the active routers themselves. Hence, a QoSPC component for managing CISCO 2600 routers (the one that is going to be reconfigured to change the route) is dynamically downloaded and installed in the station.

On the other hand, when the management infrastructure is a network-level over element-level management infrastructure, components running at the network-level station have only one type of underlying devices: MANBoP element-level managers. For this reason, the needed QoSPC component for processing the policy would have been previously installed since it is the same as when processing VAN QoS policies. Nevertheless, at the corresponding element-level manager the QoSPC component for managing CISCO 2600 routers will do have to be dynamically downloaded and installed to enforce the policy.

The policy action fields of the quality assurance policy contain the necessary information to re-route service provider's flows through the backup route. These fields are the flow source (optional) and flow destination values to identify the service provider's flows and the hops addresses through which these flows must routed, that is, the backup route.

The policy is enforced by the corresponding QoSPC component by configuring the CISCO router using CLI commands. These CLI commands are introduced in the router opening a telnet session with it.

After the enforcement of this policy, service provider's video flow to Customer 2 will be re-routed through the backup router skipping the interfering traffic.

All these twelve steps are done twice in the scenario, once for each management infrastructure considered.

C Testbed

The managed topology created prepared for this scenario is shown in the figures below. The first figure shows the managed topology before the node addition and before the service provider's flow is re-routed. The second figure shows the managed topology at the end of the scenario.

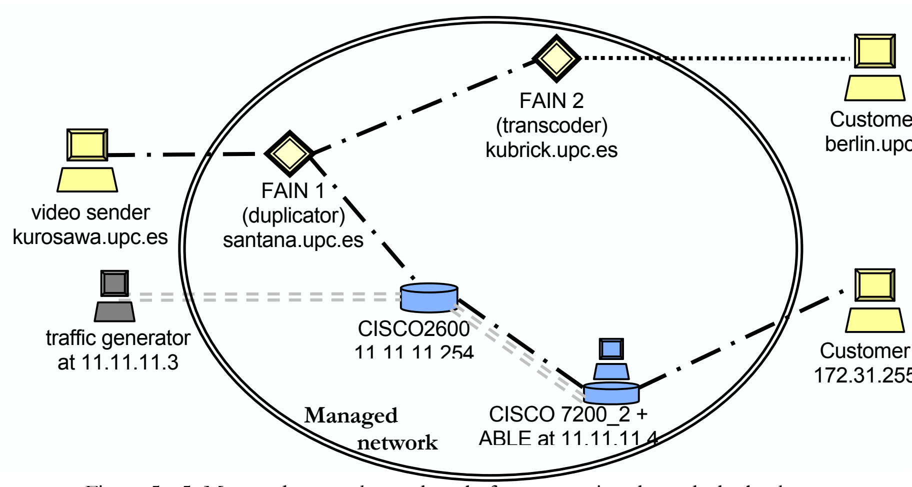

Figure 5 - 5. Managed network topology before re-routing through the backup route

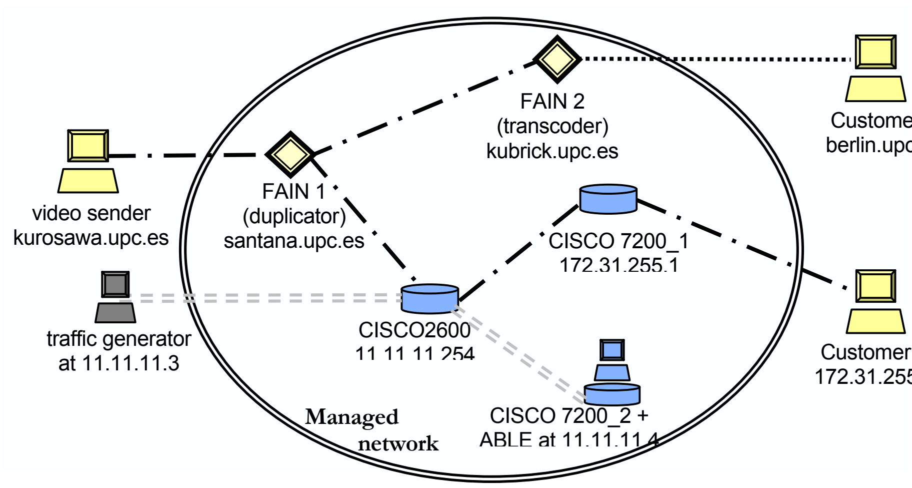

Figure 5 - 6. Managed network topology after re-routing through the backup route

In the figures below we show the whole topology including the management stations. The first figure shows the topology with the network-level only management infrastructure and the second the topology with the network-level over element-level management infrastructure.

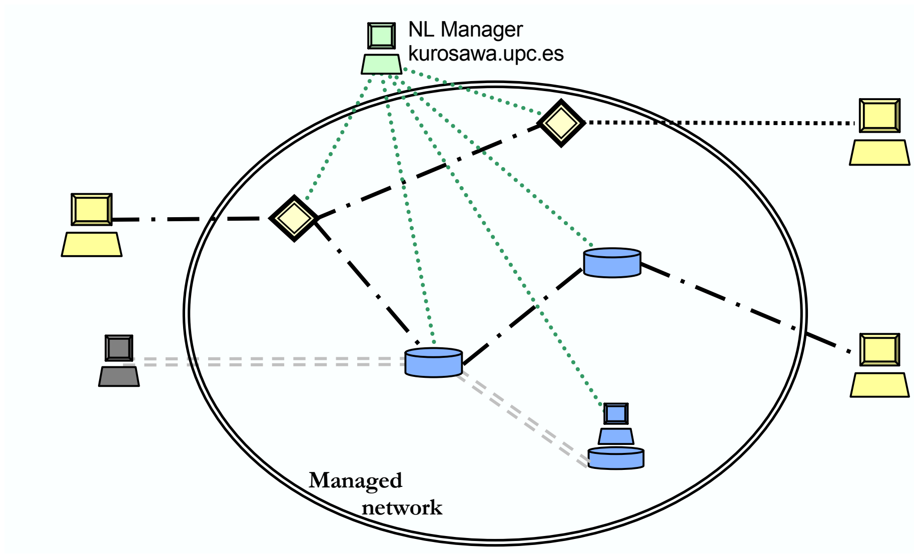

Figure 5 - 7. Testbed topology with the network-level only management infrastructure

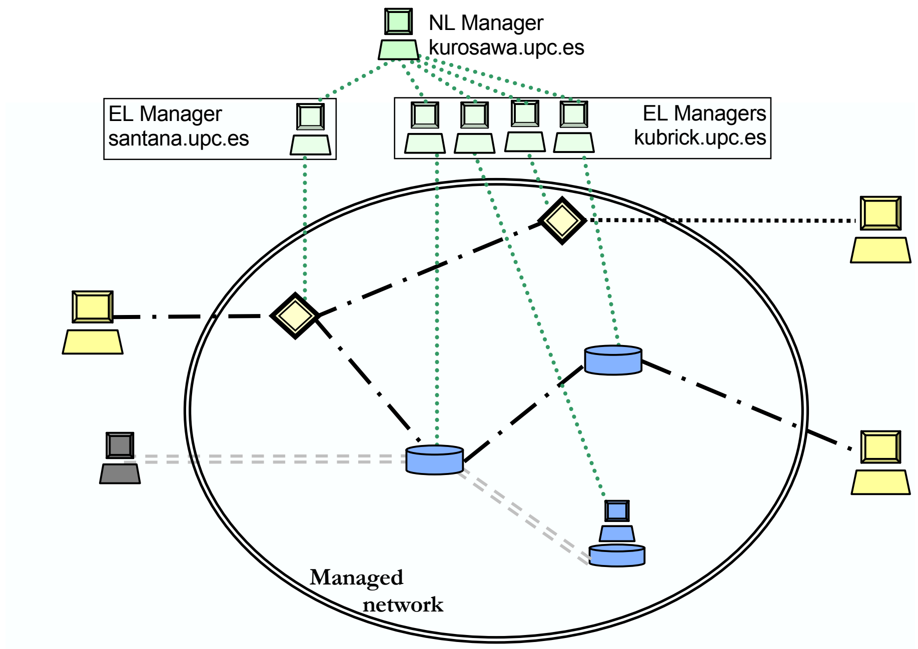

Figure 5 - 8. Testbed topology with the network-level over element-level managed topology

All nodes are physically located at building D4 inside the North Campus of the Universitat Politècnica de Catalunya (UPC). Indeed, except Kurosawa, Santana and Berlin, which are installed in a separate room all the other nodes are located in the same room. Finally, all links between nodes are Fast Ethernet links except those between CISCO routers, which are connected through Serial interfaces.

After having a clear picture of all the nodes involved in the testbed we provide in the table below the properties of each testbed node.

Node	*IP*	*Computer properties*	*OS*	*Testbed-related installed software*	*Roles played*
Kurosawa.upc.es	147.83.106.111	PC Pentium IV 1500 512MB RAM HD 30 GB	Windows 2000 Server	MANBoP, CIA, JMF [SunJAVAd], Eclipse profiler [Profiler]	• Network manager • Code server • Video server • Performance profiler
Santana.upc.es	147.83.106.104	PC Pentium IV 1500 512MB RAM HD 30 GB	Linux Debian	MANBoP, CIA, FAIN ANN, ASP, Eclipse profiler	• FAIN ANN • EL Manager for santana.upc.es • Performance profiler
Kubrick.upc.es	10.0.4.4	PC Pentium III 667 384 MB RAM HD 14GB	Linux Debian	MANBoP, CIA, FAIN ANN, ASP	• FAIN ANN • 4 EL Managers: (kubrick, 11.11.11.4, 11.11.11.254, 172.31.255.1)
11.11.11.4	11.11.11.4	PC Pentium 150 32MB RAM HD 800 MB	Linux Debian	ABLE	• ABLE Router
CISCO2600	11.11.11.254	CISCO 2600 Router	IOS	-	• Passive router
CISCO 7200_1	172.31.255.1	CISCO 7200 Router	IOS	-	• Passive router
CISCO 7200_2	172.31.255.2	CISCO 7200 Router	IOS	-	• Passive router under ABLE
172.31.255.3	172.31.255.3	CISCO 7200 Router	IOS	-	• Video flow destination
Berlin.upc.es	147.83.106.75	PC Pentium III 450 128MB RAM HD 1,6 GB	Windows 98	JMF	• Video flow destination
11.11.11.3	11.11.11.3	PC Pentium 150 32MB RAM HD 2 GB	Linux Debian	MGEN [MGEN]	• Traffic generator

Table 5 - 36. Testbed-nodes properties

As can be seen in the table the two nodes with more computing power are the ones responsible for obtaining the performance evaluation data.

3rd Scalability assessment scenario (second scenario)

A Description

To assess the scalability of the system, let's imagine that the service provider's flows cross more and more passive routers of different types where, in order to assure the video quality the monitoring of bandwidth at their corresponding interface must be done. Now, let's imagine that the network operator decides to assure the flow quality introducing a quality assurance policy for each monitored node. This would create a situation where each policy causes the monitoring of the corresponding node to decide when it must be enforced. Nevertheless, this time the monitoring is done by periodically polling the passive routers with CLI commands via a telnet connection since the monitoring is done over passive routers and not ABLE routers as before.

As more and more policies are being introduced, more and more Monitoring Meter components are dynamically installed to carry out the polling of the information. As each MM component is using certain computational and forwarding resources, the station where these components run has less free resources as the number of MMs grows.

Before going on with the description of the scenario let's clarify the reason of some adopted decisions. Particularly, the reason why the monitored nodes must be passive nodes of different types is that it is the only way of flooding the management station with MM components. If the managed devices were active or programmable routers, the MM components would not be installed inside the management station but on the routers themselves, or at least more efficient ways for monitoring them would be available. In the same way, if they were all passive routers of the same type, for example CISCO 2600 routers, the monitoring for all routers could be done by a single MM component, with a significant reduction of the manager load. Thereby, we need that the monitored devices are passive routers of different types, which must be therefore monitored in different ways and thus, need different MM components to do the monitoring.

In a more realistic scenario, where we have many devices of the same type, and some active routers, the manager station would be able to support the monitoring of much more devices that in this fictitious scenario.

The network operator will progressively introduce more and more quality assurance policies until the maximum number of 100 is reached or until the management system gets blocked. Obviously, we do not have in our laboratory the possibility of creating a testbed with one hundred routers of different types. In consequence, we will cheat the MANBoP system bootstrapping it with a testbed as the one described, although Monitoring Meter components will be all monitoring the same CISCO 7200 router. Monitoring Meters will poll the used bandwidth in one of the routers interface at five seconds intervals.

Furthermore, even if we could not use the CISCO router to open one hundred telnet sessions, we would simulate the expected behaviour of Monitoring Meter components based on the data obtained from a single monitoring on the router.

This scenario, where the network operator progressively introduces one hundred policies that need to be monitored by one hundred different MM components, will be realised twice. The first time with a management infrastructure of a single network-level management station. The second time, the management infrastructure will consist of a network-level station working over as many element-level managers as managed devices (i.e. one hundred).

B Steps

The steps for this scenario are very simple. Indeed, one single step is repeated up to one hundred times.

In this step the network operator introduces a quality assurance policy that causes the monitoring of the used bandwidth on a passive router's interface to decide when the policy must be enforced. If the threshold were reached in the monitored bandwidth, which will never happen, the policy would be enforced causing the re-routing of the service provider's flows.

The policy condition contains the information needed to correctly monitor the underlying device, as the device interface to be monitored and the threshold.

A new Monitoring Meter component will be dynamically downloaded and installed to monitor the corresponding managed device. Each MM component will actually monitor the same CISCO 7200 router with the IP address 11.11.11.253, although it will never inform that the threshold is reached whatever the monitored value is.

C Testbed

The testbed in this scalability scenario is physically simpler than in the previous scenario although the MANBoP system sees a much more complex one.

In the figures below we can see the testbed for both the network-level only management infrastructure and the network-level over element-level management infrastructure.

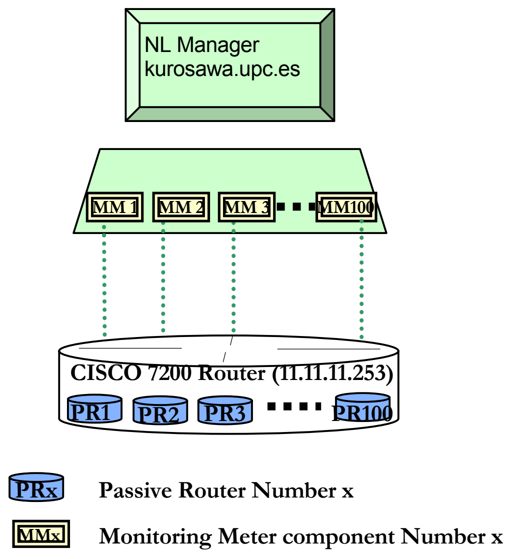

Figure 5 - 9. Scalability scenario testbed for the network-level only management infrastructure

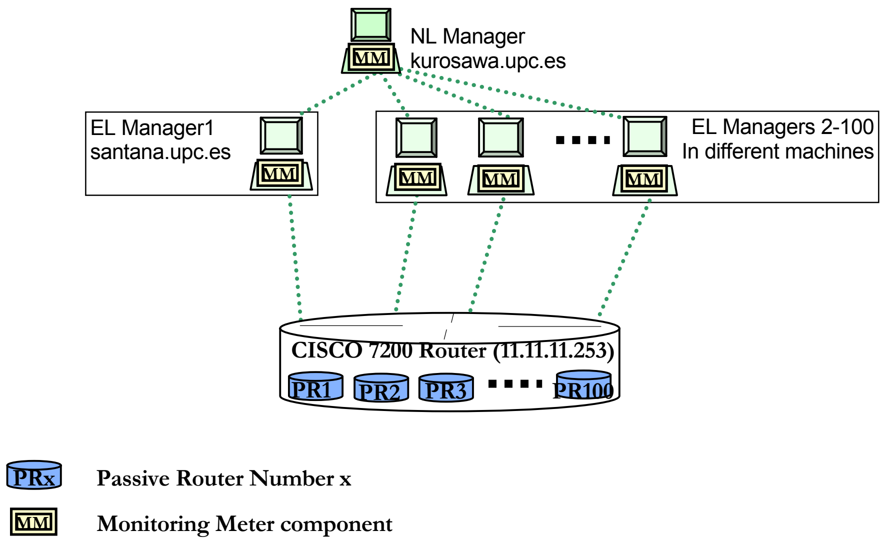

Figure 5 - 10. Scalability testbed scenario for the network-level over element-level management infrastructure

Some testbed properties that may be worth explaining appear in the figures. The first one is noticing how we have represented the fact that all passive routers are physically the same CISCO 7200 router receiving all requests. The second important aspect is that we do not have the possibility of using one hundred computers in our laboratory for introducing one hundred element-level managers. Thereby, from the second to the last element-level manager, they are run at different machines (as much as we need to boot them all), because we don't really mind the load at these computers since the profiling will exclusively be done at santana.upc.es and at kurosawa.upc.es and the performance data obtained is not affected by the load in the other computers. The computers that might be used for that goal are kubrick.upc.es, candanchu.upc.es and satriani.upc.es.

The description of the nodes location and the network connections between them given for the previous scenario applies equally in this one.

In the table below we show the main characteristics of each node in the testbed.

Node	*IP*	*Computer properties*	*OS*	*Testbed-related installed software*	*Roles played*
Kurosawa.upc.es	147.83.106.111	PC Pentium IV 1500 512MB RAM HD 30 GB	Windows 2000 Server	MANBoP, CIA, Eclipse profiler	• Network manager • Code server • Performance profiler
11.11.11.253	11.11.11.253	CISCO 7200 Router	IOS	-	• Passive router
Santana.upc.es	147.83.106.104	PC Pentium IV 1500 512MB RAM HD 30 GB	Linux Debian	MANBoP, CIA, Eclipse profiler	• EL Manager 1 • Performance profiler
Kubrick.upc.es	10.0.4.4	PC Pentium III 667 384MB RAM HD 14GB	Linux Debian	MANBoP, CIA	• Several EL Managers
Candanchu.upc.es	147.83.106.70	PC Pentium III 667 256MB RAM HD 14GB	Linux RedHat 7.2 / Windows 2000	MANBoP, CIA	• Several EL Managers
Satriani.upc.es	147.83.106.105	PC Pentium III 667 256MB RAM HD 14GB	Linux RedHat 7.2 / Windows 98	MANBoP, CIA	• Several EL Managers

Table 5 - 37. Testbed nodes properties

Again the nodes responsible for obtaining the evaluation information in the testbed are santana.upc.es and kurosawa.upc.es

Section V.6 - Conclusions

Along the chapter we have reviewed the components and functionality implemented for the proof-of-concepts as well as the scenarios with which this functionality will be evaluated.

First of all, we have described in detail the naming conventions followed during the implementation. These conventions affect the names of MANBoP packages, the names of directories where Database information is stored, the names of the classes registered in the Naming Service and finally, the names of the code that might be dynamically installed in the framework.

The naming conventions for both the MANBoP packages and the Database directories have been defined with the goal of naming them as simply and understandably as possible. On the other hand, the naming conventions for the Naming Service and the dynamically installable code are more complex because they are oriented towards facilitating the extensibility of the framework and the interactions between components.

A normal user will not need to know any of these conventions to use MANBoP normally. However, following the specified naming conventions is essential for modifying the framework or implementing new dynamically installable components for new functional domains.

The second aspect covered in this chapter is the description of the Information Model defined for MANBoP. We have described first the Information Model part related with policies and their fields and afterwards, the part containing information needed by the framework to work (i.e. managed topology information, user information, manager information, etc.).

Moreover, the policy Information Model part has been split in those objects containing functional domain independent information and those objects (i.e. policy conditions and policy actions) that have been implemented for functional domains used in the scenarios defined in the Thesis for the proof-of-concepts.

The policy structure used in MANBoP is based on the IETF Policy Core Information Model (PCIM) though simplified by defining as mandatory only those features essential for policy processing. Hence, the size of policies is considerably reduced (around five times smaller than following the PCIM model) and their processing is simpler.

On the one hand, we have simplified the definition of policy conditions and policy actions, that needed the use of several classes following the IETF PCIM, into just one class for each simple policy condition and each policy action. On the other hand, we have modified the way policy sets (or policy groups) are specified to permit the individual introduction of policies of a group into the framework. In other words, we do not need to introduce into the framework the whole policy group at the same time.

Furthermore, we have included some new fields, not considered in the IETF PCIM, to fulfil the specific requirements of this Thesis. Two examples of these fields are the *schemaId* and the *user* fields needed by the extensibility and delegation mechanism respectively.

Besides the Policy Information Model, a number of Information Model Objects (IMOs) have been defined in MANBoP to support the normal behaviour of the framework. These IMOs can be grouped in objects containing information about the managed topology, objects containing user information and objects containing information about the management system itself.

Managed topology IMOs contain information about network elements and links forming the managed network as well as about the resources available and used within the network. This information is used by MANBoP instances to find out whether resources requested by a user can be allocated and where they can be found within the managed topology.

The IMOs containing user information indicate what policies or policy groups has introduced each user, what are the access rights of the user and what are the resources assigned to a user through the enforcement of one or more policies.

In addition to the managed topology and user information, each MANBoP instance keeps information about the instance itself. In particular, it stores information about the location of the instance within the management infrastructure and about the components dynamically installed in the system. Indeed, the information about the location of the instance within the management infrastructure together with the functional domain requested is used to identify the components that should be dynamically installed.

Along the chapter, we have also described the code that has been implemented to proof the concepts proposed in the Thesis. The code has been implemented in JAVA (JDK 1.4.1) and over a CORBA platform (OpenORB).

In some cases the implemented functionality does not match exactly the designed functionality. These differences are most of the times due to the fact that we have tried to ease the implementation of the proof-of-concepts, and other times because we have tried to enhance the performance of the system. However, these changes are always small, as parameter types changes in some methods, while the expected behaviour from components and the sequence diagrams followed in every process designed have been fully respected in the implementation.

The election of the functionality implemented has been guided mainly by two concepts. The first one is to choose that functionality needed to achieve and assess the main objectives of the Thesis. The second concept was to keep the implemented functionality in a reasonable scale and not dealing with aspects that are not covered in this Thesis, as advanced conflict checking or traffic engineering algorithms.

It is obvious that not including these functionality in the implementation has an impact over the performance figures, since the system is more lightweight and performs slightly better. However, it is also true that an implementation oriented to an optimum performance of the system would probably compensate the performance degradation derived from including all functionality.

The last item described in the chapter is the proof of concept demonstration scenarios. Two scenarios have been described for the evaluation of the management framework proposed. The reason for defining two scenarios is that the definition of a scenario suitable for evaluating all aspects of the MANBoP framework (e.g. flexibility, extensibility, scalability, etc.) was a complex task. Furthermore, the resulting scenario would have been difficult to implement, process, evaluate and understand. By defining two simple scenarios linked among them, we simplify the comprehension of the scenarios as well as the testbed setup.

The first scenario described is oriented to assess all MANBoP properties except its scalability, which is addressed exclusively in the second scenario. More specifically, the second scenario is focused in the evaluation of the framework behaviour when the managed network grows. As we do not have in our laboratory the number of devices necessary to simulate a managed network of that size, we have configured the management system to send all management commands for an imaginary big network towards one single device in our laboratory.

In the following chapter we will analyse the implemented proof-of-concepts system by running the scenarios described in this chapter. We will also evaluate the MANBoP framework based on the results obtained and on very concrete criteria that will be also described in detail in the next chapter.