Development of Policy Decision Points for Policy-Based Networking: Issues and a Framework

Lisandro Zambenedetti Granville, Maria Janilce B. Almeida, Liane Maragarida Rockenbach Tarouco

Federal University of Rio Grande do Sul (UFRGS) - Institute of Informatics
Av. Bento Gonçalves, 9500 - Bloco IV - Porto Alegre, RS - Brazil
{granville, janilce, liane}@inf.ufrgs.br

Abstract. The development of PDPs is a hard task because currently there is not a widely accepted set of protocols to implement the communication between PDPs and other elements of the PBNM solutions. This paper presents a framework to support the development of PDPs in order to easy the time spent in coding new ones. The framework is based on the IETF approach for policies, which helps in the dissemination of the IETF view of policies.
Keywords: PBNM, PDP, Policy Decision Points.

1 Introduction

Policy Decision Points (PDPs) are translating elements in Policy-Based Network Management (PBNM) architectures [1]. PDPs are responsible for receiving management policies from a Policy Management Tool and translate them into actions that intend to make the managed network act accordingly to the expected behavior expressed in the policies. Naturally, network protocols are needed to transfer policies to PDPs and to configure devices when policies are translated to actions.

Some implementations suggest that PDPs could receive policies via HTTP or FTP, while the IETF prefers the LDAP-based approach [1]. Others investigate policies as scripts transferred via SNMP [2], or as programs transferred to PDPs via Corba RMI [3]. To interact with network devices in order to make them behave as stated in the policies, the COPS [4] and COPS-PR [5] protocols have been defined by the IETF, but SNMP, TELNET/CLI and HTTP/HTTPS seem to be solutions mostly preferred by the network players and administrator. In any case, there is not a widely accepted set of protocols to carry communications in PBNM, which leads to a situation where different PBNM systems cannot easily communicate each other, making integration hard to be achieved.

The lack of standardization turns the development of PDPs a hard task. Developers have to choose protocols that may not be supported by the Policy Management Tool or by the target network devices. Although standardization is slow and hard to be achieved, a framework devoted to the creation of new PDPs may ease their development through the organization and systematization of the several functional parts a PDP is supposed to support. In this paper we introduce a proposal of such a framework based on a set of elements that separates, in logical units, the tasks to be internally executed by a PDP. In order to validate our framework, we have developed PDPs to support QoS and multicast policies that were used to managed the QoS of critical multicast flows of a videoconferencing application in a DiffServ network. Although we have chosen specific protocols to our PDPs, they can be quickly replaced by other protocols without requiring modification in the PDP core software. This way, we believe that the development of PDPs can be faster, and its adaptation in the case of protocols replacement easier.

The remainder of this paper is organized as follows. Section 2 discusses about PBNM, paying special attention to the PDPs in the context of the IETF works. In Section 3 we present our framework, introducing the elements that a developer of new PDPs may reuse and the elements a developer must provide. Section 4 explains the development of a PDP prototype that implements the proposed framework, and Section 5 finally finishes this paper with conclusions and future work.

2 PDPs in the General Policy Architecture

Although there is no standardized Policy-Based Network Management (PBNM) architecture, there is a set of elements commonly accepted as required in a PBNM solution [6] [7]. Fig.1, shows the common referenced PBNM architecture.

In this architecture, the policy management tool allow the network administrator to edit and store management policies into a policy repository. PDPs, on their turn, are intermediate elements that receive policies from the management tool and apply them to the Policy Enforcement Points - PEPs (i.e. the elements inside the network devices where the policies are enforced). Thus, PDPs have at least two communicating interfaces: one with the management tool and another with the PEPs. With the lack of standardization, it is hard to chose the protocols to be used in each interface.

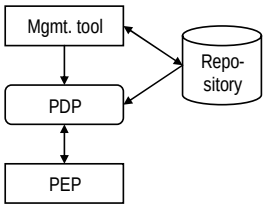

Fig. 1. IETF policy-based network management architecture

2.1 Pushing Policies or Making a PDP Download Them?

There are basically two approaches to send policies to a PDP: the push and pull approaches. In the push approach the management tool downloads a policy from the policy repository and uploads it to the PDP. The upload to the PDP may be executed using FTP, HTTP post messages, a sequence of SNMP set messages, or even proprietary protocols. In the pull approach, the management tool only informs the PDP about the location (URL) of a policy that needs to be retrieved. The PDP, on its turn, downloads the policy from the repository accordingly to the URL previously informed by the management tool. Also in this case, FTP, HTTP, SNMP and proprietary protocols may be used. The IETF particularly prefer the use of LDAP [1]. In this case, the policy repository is implemented by a directory service accessed through LDAP. The policies to be retrieved are identified by their Distinguished Names (DN) within such directory.

In the end, choosing o protocol to transfer policies to a PDP is merely an agreement between a target PDP and the management tool. Although the IETF prefers LDAP, there is no obligation on that. However, the obvious but important point here is the fact that the chosen protocol have to be supported by all PDPs in a managed domain, otherwise the management tool would have to understand a gama of different protocols to transfer policies to the different PDPs of the domain. Taking this statement from a PDP perspective, if a developer is supposed to create a new PDP, he or she must choose as the policy transfer protocol the same one used by the management tool and all the others PDPs in the managed domain. Summarizing, the choosing of a protocol to implement policy transfer to a PDP depends on the already previously protocol supported by the management tool and other PDPs in the same domain.

2.2 Waiting for Devices Requests or Configuring Them?

In order to deploy a policy, PDPs have to communicate with PEPs within devices. Here, also two different approach are found: outsourcing and provisioning.

In outsourcing, a PEP initiates the communication with a PDP asking for policy decisions every time an internal PDP event that require decision occurs. Particularly, outsourcing is interesting in a IntServ/RSVP [9] domain, where RSVP messages request for resources to routers. Every time an RSVP-enable router receives a request, it can contact a PDP to verify if that request can be accepted according to the current policies stored in the PDP. The main important protocol for outsourcing support is COPS [4] which was conceived for this specific purpose.

In the provisioning approach, however, PDPs initiate the communication, indicating to the PEPs how they should behave. Provisioning, as the opposite of outsourcing, is more suitable for DiffServ domains. Since no reservation protocol is expected, routers have to be configured to properly support the QoS requirements of DiffServ aggregates. In this case, a PDP configures DiffServ-enable routers to accomplish the expected behavior. Although COPS-PR has been defined to support the provisioning approach, several other protocols can be used as well. Typical protocols used for device configuration (e.g. Telnet, SSH and even HTTP in Web-enabled devices) may be used in a provisioning operation.

From the developer perspective, the choice of the approach and protocol to be used in the PDP/PEP communication depends on the PEP capabilities and protocol support. If the device that holds the target PEP supports COPS, then the PDP should support COPS as well (and outsourcing). However, if the target PEP can only be configured via SSH, then the PDP should be implemented with SSH support in mind. In the end, the decision about the protocol to be used in the PDP/PEP communication is based on the available protocols supported in the target devices.

2.3 Policies and PDPs Internal Operations

As seen before, external issues drive decisions on the development of PDPs. Internal issues, on their turn, drive implementation decisions too.

According to the IETF approach, a policy is a set of policy rules that are composed by policy conditions and policy actions in a IF-THEN form. The evaluation of the policy conditions of a policy rule must result true in order to all the policy actions to be executed. If the policy conditions evaluates to false, no policy action is executed for the particular policy rule. Further details about the IETF policies can be verified in the definition of PCIM (Policy Core Information Model) [10] and PCIMe (PCIM extensions) [11].

Policy conditions are expressed through two different kinds of variable: the policy attribute variable and policy value variable. A policy attribute variable is compered against a policy value variable of a policy condition. Simple conditions have just one attribute/value comparing, while more complex conditions can be formed through ANDed or ORed attribute/value comparing. Policy attributes describe parameters of the technology the policy is related to (e.g. QoS and security). Some parameters may be related to different technologies at the same time, as the flow description parameters used in both QoS and security policies. Moreover, there are some other policy parameters that may be related to all sort of policies, independently of the target technology the policy is related to. Temporal parameters used to schedule the policy activation are example of such parameters.

Developing a PDP requires the definition of which parameters the PDP will support, and that comes from the technology the PDP is being developed to. However, every PDP should support temporal parameters, despite the target technology. Taking all of this into account, the next section present the proposed framework for rapid PDP development.

3 The Proposed Framework

The proposed framework to support the rapid development of PDPs is presented in Fig. 2.

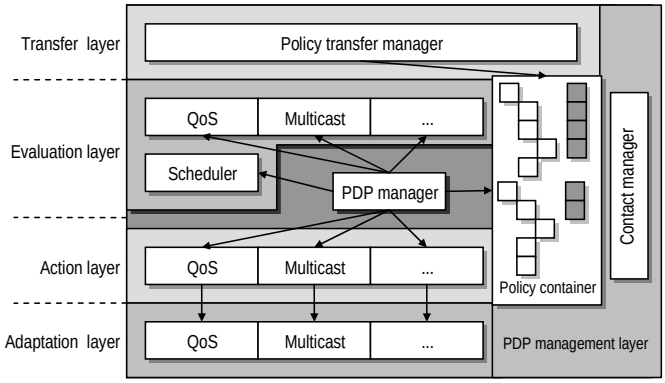

Fig. 2. PDP framework

On the top of the framework resides the **transfer layer**. The first support a PDP is supposed to provide is the ability to receive policies. As we have presented before, policy transfer can be carry out through the push or pull approaches. The **policy transfer manager** is the element of the proposed framework responsible for the implementation of the policy transfer. It encapsulates the support for push or pull approaches and the protocols used in such transfer. Also, the transfer manager has the responsibility to translate the retrieved policies from their original representations to an internal representation, which is the instantiation of objects that stores the policy in the internal memory (the **policy container** is such memory allocated to hold transferred policies).

Transferred policies can only be deployed to specified PEPs. Thus, a mechanism to register PEPs inside a PDP and to establish relationships between registered PEPs and transferred policies has to be provided. The **contact manager** of the **PDP management layer** is the element responsible for such tasks. The contact manager allows the external management tool to register PEPs in the PDP and determines which are the target PEPs for each transferred policy. In Fig. 2, transferred policies (and their hierarchical policy rules) are presented as empty boxes in the policy container. PEPs, on their turn, are presented as filled boxes in the policy container. Further features of the PDP management layer will be explained ahead.

Once a policy is internally stored, it needs to be evaluated. For that, several evaluation elements (or evaluators) are found in the **evaluation layer**. Each evaluation element (e.g. scheduler, QoS, multicast, etc.) is supposed to access a policy rule and evaluate the attributes related to the technology the evaluator is related to. It means that a policy rule is distributed evaluated along the evaluators. For example, if a policy rule has time, traffic and QoS constraints, as the policy presented in Fig. 3, then a scheduler, traffic and the QoS evaluators would be used to verify the policy conditions of that policy. If a policy rule evolves to true after its evaluation executed by the evaluation elements, then the associated policy actions (e.g. priority = high) need to be translated, in order to apply the policy in the target PEPs. The **PDP manager** is the element that coordinates the policy evaluation (calling evaluation elements), and the corresponding action translations (every time a policy rule evolves to true).

Actually, action translation is executed in the **action layer**, where action elements are found. There would be exactly one action element in the action layer for each evaluation element in the evaluation layer. The only exception is the scheduler evaluator that does not have a corresponding action element, because although time constrains in policy conditions

```
if (now >= 6pm and now < 8pm)     and
   (availableBandwidth > 2 mbps) and
   (ipSourceAddr='200.10.248.3')
then
    priority = high
end
```

Fig. 3. A policy example

are common, normally there are no time actions in policy actions. Each action element translates the actions expressed in a policy rule to a call to an interface for the elements in the layer below, that is the **adaptation layer**.

The elements of the adaptation layer are responsible for the deployment of policy actions in the target PEPs using the protocols supported by such PEPs (when in the policy provisioning). For example, a bandwidth reservation would be executed through configuration of a DiffServ PEP every time the QoS adaptation element receives a QoS adaptation call from the action layer. In outsourcing, the adaptation layer elements must wait for policy requests from the target PEPs, and reply such request accordingly with the current set of deployed policies. For example, a COPS reply would be generated by an element of the adaptation layer in response to a COPS request issued by a RSVP-enabled router when a new reservation request arrives to the target PEP. It is important to notice that the policies within the PDP are passed to all the internal elements of the proposed framework. The only point where such policies are not passed is from the action layer to the adaptation layer. In this point, the policy actions are took as parameter for an adaptation element call. It means that the elements in the adaptation layer are not policy-aware; they only receive requests for PEP configuration.

3.1 Elements Availability

Each of the elements of the proposed framework fits into one of the following categories: provided, provided and replaceable, user provided for technology, user provided for PEP.

Provided elements are those originally provided by the architecture and that can not be removed or replaced. They are composed by: scheduler (in the evaluation layer), PDP manager and policy container. These elements are the core of the architecture, and are not supposed to be replaced by other elements. **Provided and replaceable** elements are provided, but can be replaced if desired. Policy transfer manager and contact manager are those provided and replaceable elements. A policy transfer manager that supports pull, for example, could be replaced by another that supports push. Even more, one could replace a pull or push transfer manager by another more sophisticated one that could support both pull and push approaches. In the case of the contact manager, it could, for example, originally support an SNMP interface for the PDP external management. If SNMP is not suitable for some reason, such contact manager could be replaced by another that would support another protocol (e.g. HTTP).

User provided for technology elements are those that belong to the evaluation and action layer. Such elements (exception for the scheduler) have to be provided by the developer of a new PDP in order to support a specific policy target technology, such as QoS, security or multicast. One important point is that these elements are supposed to be provided without any concern about the target PEP protocol, i.e., user provided for technology elements are supposed to be protocol independent. The user provided for technology elements found in the evaluation layer are those that evaluate attributes of the target technology, while those found in the action layer translate technology-related actions to configuration requests send to the adaptation layer elements.

Finally, **user provided for PEP** elements are found in the adaptation layer and complements the functionalities of the action layer elements. User provided for PEP elements provide specific support for the protocol used to contact each target PEP, i.e., such elements are PEP protocol-aware. They receive configuration requests from action elements and translate such requests to messages that configures a target PEP using the configuration protocol supported by such PEP. In the case of outsourcing, the user provided for PEP elements have to have the ability to wait for PEPs requests issued through a specific protocol (e.g. COPS) as well. User provided for PEP elements have also to be provided by the developer of PDPs.

3.2 Internal Policy Representation

In this section we present the details about the operations of the proposed framework. To a better understanding, internal policy representation and storage need to be verified. As presented before, a transferred policy is stored in the policy container, which is a memory that holds the policies that the PDP must evaluate and deploy. The policies on the container are structured as a set of objects and their relationships. Since this work is strongly based on the IETF PCIMe policy modelling, policy rules hierarchies are possible. In this case, a policy is composed by a set of policy rules that can also be composed by other sets of policy rules. Fig. 4a presents an example of a policy that has an internal rules hierarchy.

More internal policy rules are only evaluated if more external policy rules have already evaluated to true. It mean that not all policy rules of a hierarchical policy need to be evaluated: only those whose parent already evaluated to true.

Once a policy rule evaluates to true, its policy actions have to be applied, and the descent policy rules evaluated. If a policy rule in true is evaluated back to false, then all policies rules in the descent order of the hierarchy will be deactivated.

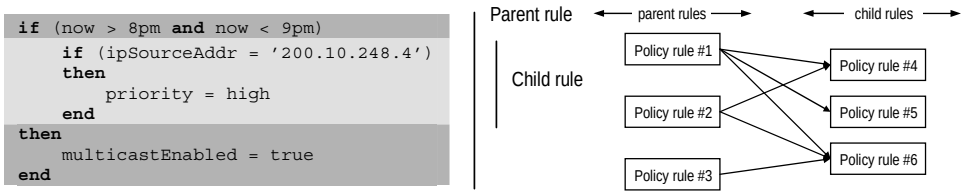

Fig. 4. (a) An hierarchical policy example, and (b) Policy hierarchy with rules reuse

In the end, each policy rule may be in the activated or deactivated state. If a policy rule is active, it means that its parent have evolved to true; if it is deactivated that is because its parent is evaluated as false. Active policies must be evaluated, while not active policies must not. It means that we need a way to express policy activation (which is a concept absent in the policy definition). A simple one bit flag can be used to that.

With reuse in mind, the IETF has defined the policy modelling in a way that a policy rule can be reused in several different parent police rules. In this case, a single policy rule may have more than one parent (Fig. 4b). In the internal PDP policy representation, each policy rule consumes only one portion of memory, i.e., there is only one copy of each policy rule. The problem in this case is to determine whether a policy rule with more than one parent is active or not. If at least one parent evolves to true, then the policy rule should be activated. If all parents evolve to false the policy rule needs to be deactivated. In order to track which parents of a policy rule have evolved to true or false, instead of a single one bit flag we must use an array of a structure that identifies a parent and the activation state of the policy rule if only that parent would be present. The final activation state would be computed by the logical OR of all individual activation state per parent. Fig. 5 presents the policy activation table for rules ♯4 and ♯6 from Fig. 4a. Supposing that rules ♯1 and ♯2 are false and policy ♯3 is true, then that rule ♯4 would no be activated because none of its parent rules is true. Rule ♯6, on the other hand, would be activated because at least one of its parent rules is true, in this case the rule ♯3.

Policy rule #4 activation table

Parent policy rule	Policy rule #1	Policy rule #2	Activate?
Would activate?	False	False	False

Policy rule #6 activation table

Parent policy rule	Policy rule #1	Policy rule #2	Policy rule #3	Activate?
Would activate?	False	False	True	True

Fig. 5. Computing the policy rule activation state

Another important point is the changing of the policy evaluation result, which is different from the activation state. An activated policy may evaluate to true or false, while a deactivated policy is not evaluated at all. A just active policy has its evaluate state initially set to false before the first real evaluation. After the evaluation, carry through the elements of the evaluation layer, the evaluation state may keep false. In this case, no action is executed. As soon as the policy evolves finally to true, the actions needs to be executed in the PDP. After that, if the policy stills evaluates to true, no further actions is executed. However, if the policy evolves back to false, then new actions is executed in order the give to the PEPs the opportunity to to rollback to the previous state before the policy first evolved to true. Thus, actions is only executed when the evaluation of a policy rule changes to a new value (from false to true, or from true to false). In this case, we also need another flag that indicates the current policy evaluation state. In the end, each policy rule has some additional information needed in the evaluation process. Fig. 6 presents such additional information.

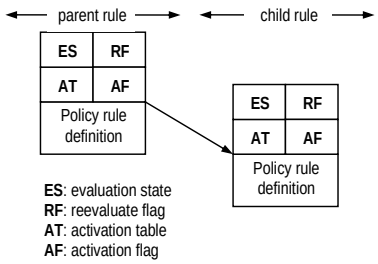

Fig. 6. Policy rule additional information

The evaluation state (ES) flag is the one used to store the result of the last evaluation of an active policy. For deactivated policies the ES flag has no meaning. The reevaluate flag (RF) is used to indicate that a new process of evaluation is required in order to update the ES flag. The used of the EF is further explained in the next subsection. The activation table (AT) stores information like those previously depicted in Fig. 5 and the activation flag (AF) is the result of a computed activation table, and indicates whether a policy rule is active or not.

3.3 PDP Internal Engine

The internal engine of the PDP has its main logic coordinated by the PDP manager element. The sequence of execution in the engine follows the steps below.

First, the PDP manager scans the policy container for active policy rules - those that have the activation flag (AF) in true (every top policy rule for each policy is always active, since they do not have a parent policy). For each active policy rule, the PDP manager extracts the rule conditions and register them in the corresponding evaluation elements. For example, in the case of the policy from Fig. 3, which has time, traffic and QoS constrains, the PDP manager registers the time conditions in the scheduler evaluator and the traffic and QoS conditions in the QoS evaluator. In the case of Fig. 4, which has only time constrains in the top policy rule, the PDP manager registers the rule only to the scheduler evaluator.

Internally, each evaluator has a list of the policy conditions that have been previously registered by the PDP manager, and that are supposed to be evaluated. An execution thread, per evaluation element, constantly checks the registered conditions in the internal list and compute a final result. If a change in the evaluation result occurs, the evaluation element then accesses the corresponding policy rule in order to set its reevaluate flag (RF) to true. It indicates that the whole set of conditions of the policy rule needs do be verified and the evaluation state (ES) flag needs to be recomputed. It is important to note that each evaluator runs its own thread, and each thread accesses the policy additional information (Fig. 6) to set the reevaluate flag every time a new evaluation state is required to be computed. However, the new evaluation state is not computed by the threads of the evaluation elements: it is computed by the thread of the PDP manager. Every time the PDP manager finds an active policy rule in the scan process presented before, it also checks if the reevaluate flag is up. If so, the PDP manager checks the current result of each policy condition (previously evaluated by the evaluators) and compute the new evaluation state flag ANDing and ORing the conditions status accordingly with the rule conditions composition. Then, the reevaluate flag is set to false since the need evaluation is finished.

If the just computed evaluation state value is different from the previous value, then an action should be executed, either to deploy the policy rule to the target PEPs or to remove such rule from the target PEPs. If the new value evolved to true, then the PDP manager sets its entry in the activation table (ET) of the descent rules to true; if the new value evolved to false, the entry in the activation table of the descent rules must be set to false. In the last case, if the deactivated descent rules have been already deployed (i.e. the evaluation flag was true), actions to rollback the target PEPs need to be executed as well. In the end, it means that policy actions are deployed only if the policy rule is active (activation flag is true) and the evaluation of the rule evolved to true (evaluation flag is true). However, policy actions are removed from the target PEPs either if the rule evolved to false (evaluation flag is false) or if the rule is deactivated (activation flag is false). Thus, once a policy rule evolves to false its own actions need to be removed, their descent rules are deactivated and the actions related to those descent rules with the evaluation flag in true are removed from the target PEPs as well.

4 Implementation

We have developed a prototype based on the previous presented framework. The prototype was implemented in Java an maps most of the elements of the framework to Java classes and interfaces grouped in packages.

4.1 Classes, Interfaces and Packages Organization

Three packages holds the classes that form the framework prototype: policyContainer, pdpDaemon and userProvided.

The **policyContainer** package holds classes used to store retrieved policies as a set of object instances. Such classes were defined based on the CIM, PCIM and PCIMe specifications. When a policy is retrieved by an implementation of a policy transfer manager, such manager instantiate several objects of the policyContainer classes that holds, inside the PDP, all policy elements, such as policy rules and rules hierarchy, policy rule conditions, values, actions, additional information (Fig. 6) and so on. Thus, the remainder classes of the prototype have access to the transferred policies accessing the object of the policyContainer classes. As a final remark, the policyContainer package does not define any Java interface (in opposite with the other packages presented below).

The **pdpDaemon** package holds the core elements of the framework: `pdpScheduler`, `pdpManager` and `pdpDaemon` (Fig. 7a). The `pdpScheduler` implements the scheduler element of the evaluation layer of the framework. It is responsible for checking time constrains in the transferred policies. Since time constrains are not technology-dependent, the

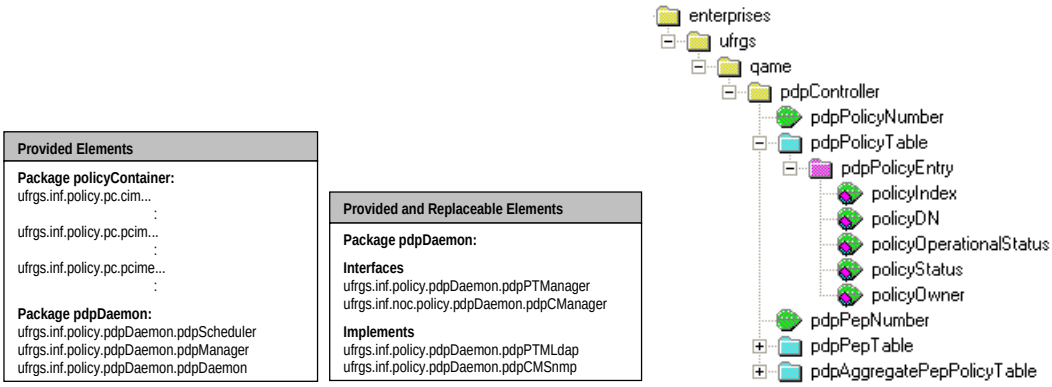

Fig. 7. (a) Provided elements, (b) Provided and replaceable elements, and (c) MIB for policy transfer

`pdpScheduler` does not need to be a replaceable element. The `pdpManager` implements the key manager of the whole framework coordinating all the other elements. Finally, the `pdpDaemon` implements the entry-point of the PDP software.

The pdpDaemon package also defines replaceable elements (namely, the policy transfer manager and the contact manager) through two interfaces: `pdpPTManager` and `pdpCManager`. We provide in our original software two implementations for these interfaces: `pdpPTMLdap` and `pdpCMSnmp` (Fig. 7a). The `pdpPTMLdap` implements a transfer manager that is able to download policies from a directory using LDAP (as suggested by the IETF). Our provided policy transfer manager only supports the pull transfer approach. However, if push is required, the developed of a new PDP can replace our `pdpMTMLdap` coding another transfer manager that implements the `pdpPTManager` interface.

We also provide a contact manager through the `pdpCMSnmp` class. In this case, the contact manager support SNMP and is implemented as an SNMP agent that exposes PDP internal information to an external, SNMP-enable policy management tool. In the current implementation the `pdpCMSnmp` supports the PDP-MIB partially presented in Fig. 7c. Again, if another contact manager is required (e.g. a contact manager via HTTP) the PDP developer can built its own implementing the `pdpCManager` interface.

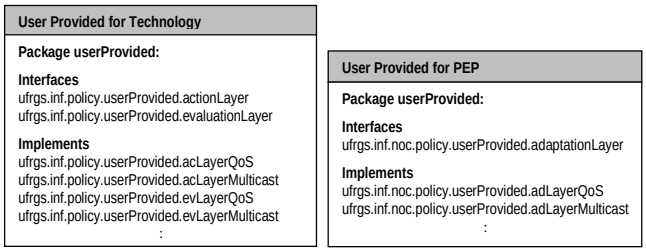

Fig. 8. (a) User provided for technology elements, and (b) User provided for PEP elements

The last package available is **userProvided**. It was defined to support the logic of the evaluation, action and adaptation layers of the framework. User provided for technology elements are supported through the implementation of the `evaluationLayer` and `actionLayer` interfaces (Fig. 8a). User provided for PEP elements are supported through the implementation of the `adaptationLayer` interface (Fig. 8b). `evaluationLayer`, `actionLayer` and `adaptationLayer` are generic interfaces to be used in all implementations of evaluation, action and adaptation elements, independently of the final technology to be supported (e.g. QoS and multicast) or the target PEP protocol (e.g. SNMP, COPS, etc.).

4.2 The Provided QoS and Multicast Example Support

In the current implementation we provide support for QoS and multicast policies through the classes `evLayerQoS` and `evLayerMulticast` (for the evaluation layer elements), `acLayerQoS` and `acLayerMulticast` (for the action layer elements) and `adLayerQoS` and `adLayerMulticast` (for the adaptation layer elements).

The `evLayerQoS` class is able to evaluate traffic and QoS parameters (e.g. IP source and destination and bandwidth and lost rates). The `evLayerMulticast`, is able to evaluate multicast-related parameters such as the multicast routing protocol used. Both classes receive policy rule conditions to check QoS and multicast parameters, and set the reevaluate flag each time new values are found. If a QoS and multicast policy rule evolves to another state, e.g., from false to true, the PDP manager implemented in the `pdpManager` class calls the action elements ”behind” the `actionLayer` interface.

The `acLayerQoS` and `acLayerMulticast` are these elements behind the `actionLayer` interface. As soon as they are called they try to implement the policy rule actions calling the specific protocol again behind the `adaptationLayer`. Here, the `adLayerQoS` and `adLayerMulticast` implement the support for the PEP target protocol. Currently, we suppose that a PDP as a whole are only able to access a PEP used one specific protocol per `adaptataionLayer` element. It means that the QoS support in the adaptation layer would be only able to configure PEP using just one protocol. In our implementation, the `adLayerQoS`, for instance, can configure CBQ in Linux-based DiffServ routers only through Telnet. Actually, this limitation can be bypassed implementing more sophisticated adaptation layer classes.

4.3 PDP Configuration Files

The provided implementation supports QoS and multicast as presented before. However, since implementations of other supports are expected, the PDP needs to be configured to work with the proper classes that provides the implemented new supports. That is achieved through the configuration of PDP using simple configuration files that point to the location and description of the classes to be loaded. The `pdpDaemon.conf` file holds the data relate to the `pdpPTManager` and `pdpCManager` interfaces (Fig. 9a).

```
#pdpDaemon.conf
#Implemented Class based pdpDaemon Interfaces

ufrgs.inf.policy.pdpDaemon.pdpPTMLdap implements pdpPTManager
ufrgs.inf.policy.userProvided.pdpCMSnmp implements pdpCManager
```

```
#pdpDaemon.conf
#Implemented Class based pdpDaemon Interfaces

ufrgs.inf.policy.pdpDaemon.pdpPTMLdap implements pdpPTManager
ufrgs.inf.policy.userProvided.pdpCMSnmp implements pdpCManager
```

Fig. 9. (a) pdpDaemon.conf, and (b)pdpProvided.conf

The second configuration file (`userProvided.conf`) holds the description and location of the classes that implement the `userProvided` package. In this case, several implemented classes are allowed for each interface provided (Fig. 9b).

5 Conclusions and Future Work

In this paper we have presented a framework and a prototype to support the development of Policy Decision Points (PDPs) in Policy-Based Network Management (PBNM) systems. We implemented the prototype in Java because we wanted a operating-system independent platform, since PDPs are expected to be run in several different environment, from Windows or Linux hosts to Java-based routers.

We have initially supported QoS and multicast as target technologies for the prototype. However, any other technology can be supported through the implementation of the presented `pdpEvaluation`, `pdpAction` and `pdpAdaptation` interfaces. Comparing the development of new PDP using the proposed framework against the development of PDP from scratch, we believe that our solution brings real advantages because de PDP developer does not need to implement all the elements of the PDP: he or she only needs to provide the elements absent in the original implementation that is required in the particular environment of the developer. For example, the developer does not need to support PDP and management tool communication if SNMP-base communication is enough. Policy transfer support also does not need to be developed if LDAP support is accepted (as expected). Even more important, our framework supports policies described in the IETF approach, which is not simple to be understand and implemented from scratch. Thus, we believe that the main contribution of this work is the potential for the dissemination of the IETF policy approach along PDP developers.

Although the framework easily supports policy provisioning, outsourcing is expected to be implemented with more difficulties, because it is based on a bottom-up communication sequence. Today, there is no explicit message receiving service in the framework that could help in the development of outsourcing PDPs. This point is currently under investigation and further improvements on the framework must be designed to accomplish such drawback as future work.

References

1. A. Westerinen et al., "Terminology for Policy-Based Management", RFC 3198, IETF, November 2001.
2. P. Martinez et al., "Using the Script MIB for Policy-based Configuration Management", IEEE/IFIP NOMS, 2002.
3. H. Guo and T. Becker, "Programmable resource control in global active IP networks", IEEE LCN, 2000.
4. D. Durham, J. Boyle et al., "The COPS (Common Open Policy Service Protocol)", RFC 2748, IETF, January 2000.
5. K. Chan et al., "COPS Usage for Policy Provisioning (COPS-PR)", RFC 3084, IETF, March 2001.
6. P. Flegkas et al., "Design and Implementation of a Policy-Based Resource Management Architecture", IEEE/IFIP IM, 2003.
7. L. Granville et al., "PoP - An Automated Policy Replacement Architecture for PBNM", IEEE POLICY, 2002.
8. H. Mahon et a., "Requiriments for a Policy Management System", <draft-ietf-policy-req-02.txt>, IETF (expired draft).
9. R. Braden, D. Clark and S. Shenker, "Integrated Services in the Internet Architecture: an Overview" RFC 1633, IETF, June 1994.
10. B. Moore et al., "Policy Core Information Model – Version 1 Specification" RFC 3060, IETF, February 2001.
11. B. Moore, "Policy Core Information Model (PCIM) Extensions" RFC 3460, IETF, January 2003.
12. M. Wahl et al., "Lightweight Directory Access Protocol (v3): Attribute Syntax Definitions" RFC 2252, IETF, December 1997.