Active Networks as a Developing and Testing Environment for Network Protocols

Raouf Boutaba
University of Waterloo
School of Computer Science
rboutaba@ uwaterloo.ca

Andreas Polyrakis
National Technical
University of Athens,
ECE Department
apolyr@netmode.ntua.gr

Alvaro Fernandez Casani
Institute of Particle Physics
of Valencia,
IFIC - CSIC
Alvaro.Fernandez@ific.uv.es

ABSTRACT

Active Networks is a modern network approach in which pieces of code can be downloaded and executed on network devices, affecting in this way their behavior. This approach alters the philosophy of a computer network, makes it resemble to a distributed system and affects not only network protocols, services or applications, but also high-level mechanisms and procedures. One of the affected procedures is the development and testing of new protocols. By exploiting active network properties the development of a network protocol can be simplified to software development. Expensive and time-consuming hardware implementations are avoided, while the code can be developed, shared and tested by individual researchers. Testing can be performed on actual conditions instead of using inaccurate simulations. Early implementations of the protocol, which can be modified easily while the protocol evolves, can be used to obtain useful feedback. This paper describes our experiences of developing and testing of some of the IETF COPS family protocols in an Active Environment.

1. INTRODUCTION

The process of defining a novel network protocol is a hard and complicated task: It can be decomposed into several sub-tasks or steps such as analysis, design, implementation, deployment and testing. Theoretically, these can be independent tasks that follow each other, but in practice the developers have to cycle through these sub-tasks until the final protocol is reached. This can be a hard and long process, especially if the protocol is to become a worldwide standard: In this case, the initial requirements are conceptualized by an initial group of people but during the standardization process new requirements arise, set by people from different backgrounds and with different - or even conflicting - goals. This causes significant modifications and additions to the original idea and it may take years until the final version of the protocol is reached.

Two of the most important tasks in the definition of a protocol are implementation and testing. Through them the protocol is proved to be working as desired, it can be compared to existing protocols and its performance can be assessed. Shortcomings and deficiencies not anticipated during the design are revealed. Usually, an initial prototype is first built and tested and then the protocol is evolved based on that prototype. However, the implementation of a network protocol usually requires modifications on the hardware (or firmware) of the network devices. In most cases, such modifications can only be made by the vendors of the devices, something which excludes independent researchers. Even for vendors, the process is extremely expensive and slow. These facts discourage developers from implementing the protocol on actual network devices and lead them to software simulations. The accuracy and reliability of the results of

the simulations of course are always questionable; and experiments on different simulations sometimes produce disaccording results.

The previous discussion stresses the need for a cost and time effective method to develop and test network protocols on actual network devices. An answer to this problem can be Active Networks, networks the nodes of which can be programmed and their behavior can be dynamically controlled. This radical approach makes the network resemble to a distributed system and allows the deployment of customizable network services and flexible user-aware applications. By taking advantage of the properties of Active Networks, network protocols can be easily deployed upon programmable nodes and thus be tested on real environments. Modifications on the protocols can be easily carried out since these are implemented in software rather than in hardware. Moreover, the network nodes can be programmed by anyone authorized to access the relevant resources, rather than the device vendors only. Hence, the process of developing a novel network protocol can be significantly simplified by Active Networks.

This paper discusses our experiences of developing new network protocols on an active environment. As testing (case-study) protocols we have chosen the IETF protocol for Policy-Based Networking, COPS, its extension COPS-PR, a structure defined by the latter called Policy Information Base (PIB) and a custom PIB that we have defined, the Meta-Policy PIB. This choice was considered as a good one for numerous reasons: COPS and COPS-PR are popular and promising protocols. However, since they are still RFCs, they are not widely supported by vendors. The few existing implementations may not be absolutely compatible to each other or to the current standards, since the protocols are still evolving. On the other hand, researchers that are evolving, testing or extending these protocols can find useful an implementation that can be installed on their devices and be ready to be used or modified. The same applies for developers that want to develop COPS applications or for network operators who are willing to deploy COPS in their network but they do not posses COPS equipment. Last but not least, this would serve the research goals of our team, i.e. the implementation, testing and evaluation of the Meta-Policy PIB that we have defined.

The active equipment used in our experiments comprised two Nortel Accelar (Passport) routers that support the Oplet Runtime Environment *(ORE)*. ***ORE*** is an environment that allows the execution of small java programs that control the behavior of the routers, and it is discussed in more detail in the following sections.

The structure of this paper is as follows: Chapter 2 introduces Active Networks and the Oplet Runtime Environment (ORE). Chapter 3 briefly presents Policy-Based Networking, COPS, COPS-PR and PIBs and outlines the custom PIB that we have defined and implemented. Chapter 4 describes how the previously mentioned protocols and structures have been implemented in ORE and how these have been installed and tested on our servers and active routers. Chapter 5 discusses the advantages of using such a technique to deploy and test novel protocols; finally chapter 6 concludes our work.

2. ACTIVE NETWORKS

2.1. Basic Concepts

The events in the area of computer networks during the last few years reveal a significant trend towards open architecture network nodes, the behavior of which can be dynamically programmed. Modern network protocols become increasingly complex and sophisticated and they demand access to a variety of network resources; and emerging network applications demand advanced computations and complex operations within the network. Active Networks, a technology that allows flexible and programmable open nodes, has proven to be a promising candidate to satisfy these needs.

Active Networks [1], [2], [3] is a relatively new concept, emerged from the broad DARPA community in 1994-95. In Active Networks, programs can be "injected" into the devices making them active in the sense that their behavior can be dynamically defined. Active devices no longer simply forward packets; instead, data is manipulated by the programs installed in the active nodes and devices. Packets may be classified and served on per-application or per-user basis. Complex tasks and computations may be performed on the packets according to their content. The packets may even be altered as they flow inside the network. Hence, Active Networks can be considered active in two ways [2]: First, the active devices perform customized operations on the data flowing through them. Second, authorized users/applications can "inject" their own programs to the nodes, customizing the way their data is manipulated. Due to these features of Active Networks, the open-node architecture is achieved. Custom protocols and services can be easily deployed in the active nodes, making the network flexible and adaptive to the users' and to the network/service administrators' needs.

Architecturally, Active Networks can be divided into the discrete (or programmable), and the integrated (or encapsulated) approach [1], [4]. The main difference among them is that in the former the programs are sent to the active nodes through separate, out-of-band channels, while in the latter the code is embedded into the data packets. This imposes some differences in the capabilities of the two approaches: The programmable approach seems more appropriate for cases where the administrators want to modify the behavior of the nodes, e.g. by replacing a protocol or installing a new service. The integrated approach seems more efficient when the network applications require advanced computations or customized manipulation of their packets by the network nodes. In both approaches, though, the architecture usually define some basic primitives in the active nodes that provide critical or commonly used functions such as packet manipulation, access to the environment of the node and navigation schemes, scheduling, storage.

Active Networks introduce radical changes to computer networks, making them resemble to a distributed operating system. These changes can be beneficial for a wide range of applications and tools [2], [4] and can affect almost any of the traditional processes, procedures and mechanisms, one of which is the deployment of novel network protocols.

2.2. The ORE Environment

For the purposes of our experiments, two programmable Nortel Accelar (Passport) routers that support the ***Oplet Runtime Environment (ORE)*** [5], [6] were used. **ORE** is a platform for secure downloading, installation and safe execution of Java code within an embedded Java Virtual Machine (JVM). The downloaded code runs locally on the device on a protected JVM environment and it implements customized software services that include monitoring, routing, diagnostic, or other user specified functions. **ORE** itself is a pure Java software. This means that code written for the routers can also execute on other OS platforms such as Unix or Windows. This property is particularly useful in the development and debugging of the code, which can be carried out with the aid of any modern, sophisticated Java development environment.

The ORE architecture, which is based upon the underlining embedded JVM, consists of the ORE environment, oplets and services. **Oplets** are self-contained downloadable units that encapsulate one or more services, service attributes, authentication information, and resource requirements. Oplets can provide services to other oplets and can depend themselves on other services. The ORE provides means to download oplets, manage the oplet lifecycle, maintain a repository of active services, and track dependencies between oplets and services. The ORE services use APIs that can be used in order to affect the behavior of the router. For instance, the Java Forwarding API (JFWD API) can be used in order to instruct the forwarding engine how to handle packets.

The ORE classifies services into three categories: ORE-specific, system and customized. The ORE specific services are the "Standard Services" that provide APIs for customer service creation, encapsulation and management. The system services provide access to underlying system resources such as forwarding and diverting packets. Finally, the customized services are the user-end APIs.

The client applications (oplets) are developed using the APIs provided by the above three classes of services and utilize them to do particular networking tasks either remotely or locally. *Oplets* are self-contained downloadable units which may specify attributes regarding authentication and resource requirements. ORE oplets may run as standalone applications or can be registered as custom services that run on the network device. Since oplets may depend on (custom or non-custom) services, dependency trees are created and ORE must resolve and download all relative components before installing a custom service or running an oplet.

Safety is an important aspect of the ORE architecture. Safety of the oplets during download is provided by a public-key security infrastructure that enables the downloading of trusted applets. Safety during the execution of the code is accomplished through a safe execution environment with well defined resource limits. Additional mechanisms that provide execution safety are used, such as the strong type-checking of the Java language, bytecode verification, a sandbox environment for executing the bytecode and the security manager provided by the JVM. Since the ORE and its services are constrained to running in the JVM, system stability of the core network device operations is not affected.

The ORE controls allocation of system resources by intercepting allocation calls form the service code to the JVM. To protect itself from denial of service attacks, deadlocks and unstable states, the ORE implements mechanisms for thread safety and revocation. Although it is possible to limit the creation of new threads, once one is created there is no way of limiting the consumption of computing resources because of the JVM limitations. To solve these, extensions for CPU accounting should be supported by the JVM. The ORE uses object revocation to control access to its own resources, and can detect cases where has to revoke accesses to objects that don't already exists because for example they were exported by another oplet that no longer exists.

In our work we implemented certain COPS family protocols and extensions in the form of ORE oplet services. These oplet services were then downloaded and installed on the two active routers that we possess. In this manner, we managed to implement and test novel protocols on real routers and evaluate them on a real environment. In this manner, although real routers were used, the implementation was carried out using conventional Java development tools which reduced significantly the complexity and cost of the implementation process (compared to building new hardware) and gave much more reliable results (compared to using software simulations).

3. POLICY BASED NETWORKING AND THE COPS PROTOCOL

3.1. Policy Based-Management

Policy-Based Networking (PBN) has emerged as a promising paradigm for network operation and management [7]. It is based on high-level control/management policies, [8], [9], [10] i.e. rules that describe the desired behavior of the network in a way as independent as possible of the network devices and topology. Two basic entities are distinguished: the ***Policy Enforcement Points (PEPs)*** and the ***Policy Decision Points (PDPs)*** [11]. The PEPs typically reside on the managed devices. Their role is to enforce the commands that they receive as configuration data from the PDPs. The PDPs process the high-level policies along with other data such as network state information, and generate configuration data for the

PEPs. The configuration data for each PEP are generated according to its specific role, capabilities and limitations. If the network state or policies change, the PDP may readjust the behavior of the devices, by sending updated configuration data.

The key concept in PBN is that by describing goals ("what"), rather than procedures ("how") (which happens with the traditional management techniques), the policies are separated from the network details. The high degree of abstraction and the automation implied from this model make the network easier to be managed and ensure consistency in the behavior of the devices across it. Besides, the dynamic binding of the policies with the network details that takes place in real-time at the PDPs allows new types of policies to be constructed and gives extra flexibility to the network managers. PBN is illustrated in Figure 1.

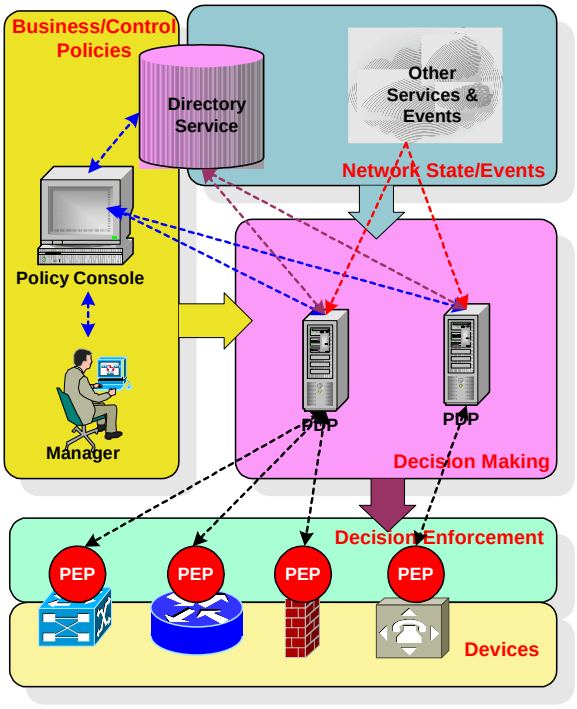

Figure 1: *Policy-Based Networking*

3.2. COPS

The IETF ***Resource Allocation Protocol (RAP)*** working group [12] attempts to standardize the communication between PDPs and PEPs through the ***Common Open Policy Service (COPS)*** [13] protocol and its extensions. COPS has already received significant attention and applications based on it have emerged [14], [15], [16].

The COPS protocol is designed to communicate self-identifying policy-related information, exchanged between the PDP and the PEP. In COPS, each PEP may support one or more clients of different client-types; different client-types exist for the different policing areas (security, QoS, admission control, accounting, etc). By supporting the appropriate clients-types, the PEP provides a way to control the various management aspects of the device. Such client-types are currently being developed by IETF [17]. These client-types are considered as extensions of the base COPS protocol, since they define details for the format and semantics of the configuration data that is exchanged between the PDPs and the PEPs. COPS for Policy Provisioning (COPS-PR) [18] is one of those client-types.

3.3. COPS-PR

RAP has developed ***COPS for Policy Provisioning (COPS-PR)*** [18] as an extension (or, client-type) of COPS. COPS-PR was initially biased towards Differentiated Services (DiffServ) policy provisioning [19]. However, it appears to be suitable for several other management areas (accounting [20], IP filtering [21], [18], security [22], etc.).

As its name implies, COPS-PR operates in a provisioning mode. The clients connect to the appropriate PDP, report their capabilities and limitations and request the initial policies to be downloaded into them. The PDP processes the request of each client and, according to the global policies and network state, generates and downloads the appropriate configuration data into the PEPs, which serve all incoming events

according to these data. If the network state or policies change, the PDP may update these configuration data, in order to keep the behavior of the managed devices consistent.

3.4. The Policy Information Base (PIB)

In COPS-PR, each client has to maintain a special database, called ***Policy Information Base (PIB)*** [23], where all the received configuration data are stored. The PIB is a structure similar to a MIB, and can be described as a conceptual tree namespace, where the branches represent structures of data, or Provisioning Classes (PRCs), and the leaves represent instances of these classes, called Provisioning Instances (PRIs). PIBs are defined by COPS-PR only as abstract structures; the details of each PIB (PRCs and their semantics) are specified in separate standard documents (such as internet-drafts or vendor private documents). Different PIBs are defined in order to cover the various management areas (Differentiated Services, accounting, security etc). PIBs are defined in a high abstraction level in order to hide the details of the underlying hardware and provide to the PDP a unified way to control the behavior of the devices regarding a specific management area, across the entire network.

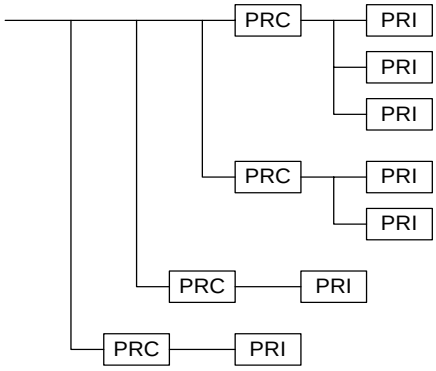

Figure 2: PIB structure

PRIs are identified within the PIB through a PRI identifier (PRID). Policies are formed as a set of PRIs in the PIB; by adding or removing PRIs, the PDP can implement the desired policies, which will be enforced at the device.

PIBs are defined using the ***Structure of Policy Provisioning Information (SPPI)*** specification [24].

3.5. The Meta-Policy PIB

The ***Meta-Policy PIB*** [25] is an attempt to push some of the PDP functionality and intelligence towards the PEP through standard COPS-PR procedures. The basic idea is that modern network devices (especially active ones) have the resources to perform some processing, and thus some of the intelligence of the PDP in the PBN model can be distributed into them. This is performed through "meta-policies", policies that are sent to the PEP by the PDP and control the former by enforcing "traditional" network policies on the device. The key concept in our framework is that meta-policies in the meta-policy PIB control the content of other PIBs on the device; by altering the contents of these PIBs the policies implemented by the device are affected. The device, after receiving the meta-policies, can monitor several events and take policing decisions on behalf of the PDP in an independent and decentralized manner, making in this way the COPS model more robust, reliable and fault tolerant.

The Meta-Policy PIB is defined according to the IETF specifications (i.e., using SPPI) It comprises 14 Provisioning Classes (which can be envisioned as tables), grouped into five groups:

- The Capabilities Group contains the Provisioning classes (PRCs) that store the capabilities and limitations of the PEP (as far as the meta-policy PIB is concerned). The PRIs of these classes are reported to the PDP when the PEP connects.
- The Base Meta-Policy Group contains the classes that form the meta-policies, define their relative priority in case of conflicts, and report their status.
- The Condition Group provides classes for forming the conditions of the meta-policies.
- The Action Group includes the PRCs that define the actions of the meta-policies.

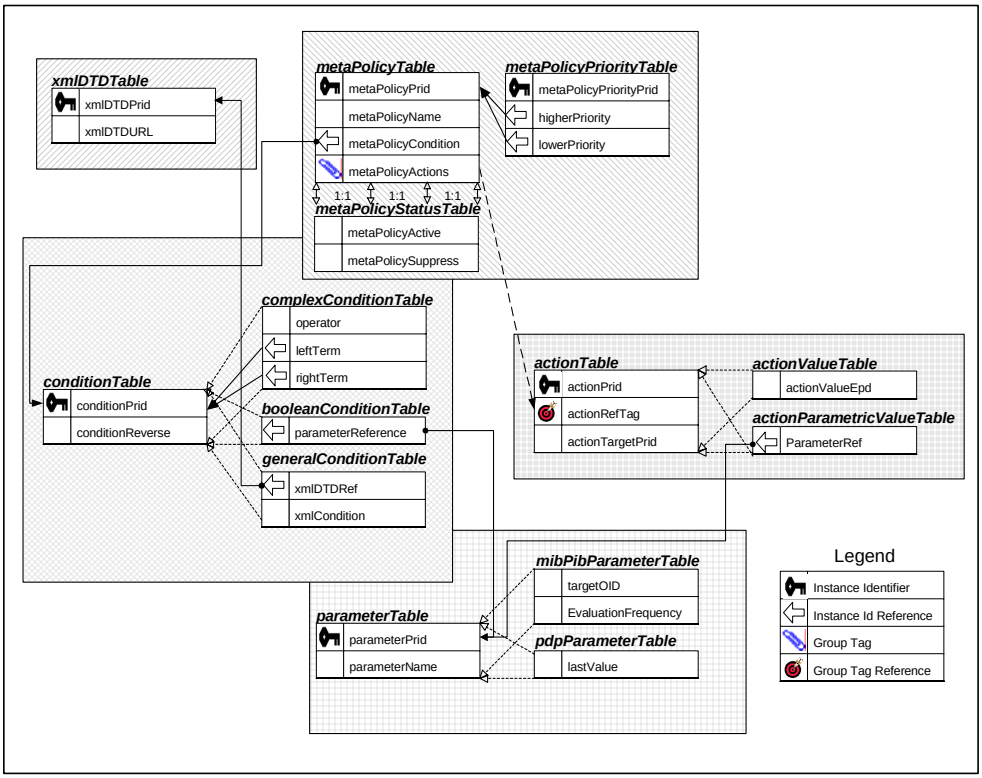

Figure 3: *The Meta-Policy PIB*

- The Parameter Group contains the PRCs where the parameters and their evaluation methods are stored.

Details of the meta-policy PIB can be found at [25]. Presenting these details here is out of the scope of this paper. As far as this paper is concerned, any PIB (either developed within IETF or not) could serve as a case study. The implementation of any PIB would require the implementation of a module that implements the respective PRCs (classes), stores instances of these classes (PRIs), and of course interpret them by modifying the configuration of the device accordingly.

4. CASE STUDY: IMPLEMENTING COPS-PR IN AN ACTIVE ENVIRONMENT

4.1. General

For the purposes of our experiments, two programmable Nortel Accelar routers were used, which were able of running small Java programs (***oplets***) on the ORE environment.

The implementation and testing was carried out in the following manner: Initially, ORE oplets that implemented the COPS functionality were developed and installed as services on the routers. Based on the already installed COPS service, additional oplets that implemented COPS-PR were built and installed as well. Both protocols were extensively tested and their correctness was assured by validating the exchanged messages though log files. Next, a small filtering PIB was defined and built. This PIB provided the routers with a minimal policing functionality, which would be controlled though COPS-PR. Finally the meta-policy PIB was built and installed on the routers as well. A simple PDP was also developed, which implemented the server side of the COPS and COPS-PR protocols. The PDP would send meta-policies to the meta-policy PIB through COPS-PR, and the meta-policy PIB would in turn modify the filtering PIB. At that time, we realized that the complexity of the exchanged information made monitoring through the initial log files inefficient. For this reason, graphical monitoring and debugging tools were also developed which monitored the COPS-PR connection and the contents of the two PIBs of the PEP. All coding (oplets, PDP server and extra tools) was performed in Java.

From a logical point of view, the most fundamental concept of our framework is policies. Policies are represented as a set of rules composed by conditions and actions that are executed when the conditions are met. A condition is a set of expressions that determine if the policy should be enforced; actions define what must be done when the conditions are met, i.e. how network resources are managed, allocated and controlled.

From an architectural point of view, the implementation follows the PBN model. The main components are the ***Policy Decision Point (PDP)*** and the ***Policy Enforcement Point (PEP)***, as described earlier in this paper. Policies are processed by the PDPs and distributed to the PEPs which enforce them. For simplicity purposes, we did not implement a ***Policy Console***. Instead, policies were written using a custom xml-based language, defined for this purpose and they were stored in pre-defined storage schemas in the ***Policy Repository***. In our implementation, the storage schema directly derives from the definition schema, described before. As a Policy Repository, a Directory Service is typically utilized and the PDP fetches the policy information through it. However, for simplicity reasons again, our implementation integrates the policy repository within the PDP.

As far as communications are concerned, the PDP and PEP communicate through COPS-PR. The base COPS protocol was first implemented, and then COPS-PR functionality was added. COPS-PR provides to the PDP a means of modifying the content of any PIB. The contents of the PIB are referenced and accessed through the PRI identifier, which uniquely defines a PIB (through its prefix) and a leaf on its tree (through its suffix). The protocol implementation was common on both the PDP and PEP (although each of them is allowed to utilize different protocol functions).

With regard to the implemented PIBs, two custom PIBs were implemented. The first PIB was a small filtering PIB which could store and force simple IP access-lists. The second one was the meta-policy PIB which could store meta-policies upon the filtering PIB. This means that meta-policies (stored and implemented by the PEP) could control the content of the filtering PIB, which in turn would affect the IP access-lists of the device. Although the PDP should be aware of the content and the semantics of the PIBs of the PEPs that it controls, the PIBs were only necessary to be implemented at the PEP side.

4.2. PDP Implementation

In general, a PDP is a complicated entity which performs advanced computations and transformations on the network policies, monitors the network and controls the PEPs. The goal of our experiments was to test the previously described protocols and PIBs; not to build a fully functional PDP. For this reason, a dummy PDP was initially built. This PDP would fetch the pre-processed policy data from the policy repository

(which represent policies and meta-policies) and would apply simple transformations to them in order to create the COPS-PR data that should be sent to the PEPs.

In order to communicate with the PEP, the PDP utilizes the COPS and COPS-PR implementation described in the next paragraphs, which adhere to the IETF specifications (except certain minor COPS simplifications, mentioned in § 4.6).

Since the PDP is written for test purposes only, the implementation details are not discussed here.

4.3. PEP Implementation

The role of the PEP is to open a session to the PDP, wait for configuration data and install them to its PIB. Also, the PEP must perform the actions defined by the contents of its PIB, i.e. it must check the contents of the filtering PIB and modify the access-lists of the router accordingly and it must check the contents of the meta-policy PIB and when certain conditions are met, modify the contents of the filtering PIB accordingly.

The main class of the PEP implementation is the class **PEP**. This class is responsible for initializing the PEP and loading all necessary classes and components (such as the COPS-PR protocol classes or the classes of the PIB). The **COPSComm** class is in charge of the basic communication over the network with the COPS protocol. This class is responsible for connecting to the PDP through a TCP/IP socket and for accepting and handling of all COPS messages and objects. The **ClientHandler** class (described also in §0) is used in order to load any PEP COPS clients (of one or more client-types). Each **ClientHandler** is able to handle different client-specific issues. We have only implemented COPS-PR as client-type (§4.7), but other clients-types can be supported (such as COPS-RSVP). The **COPSPRClient** class implements the COPS-PR client. It utilizes the COPS-PR protocol implementation (described in §4.7), and it handles all the COPS-PR objects and messages. In order to install the COPS-PR data in the appropriate PIB, the **COPSPRClient class** uses the **PIBController** class, which loads and controls PIBs. The COPSPRClient object send PRIs and their values (EPDs) to the PIBController object, which is responsible for adding, replacing or removing these data from the appropriate position of the appropriate PIB.

The implementation of each PIB is responsible to maintain its data and to take the appropriate actions that these data dictate. That is, for the filtering PIB to update the IP access-lists of the router; and for the meta-policy PIB to monitor several events and when the meta-policy conditions are met, to alter the contents of the filtering PIB.

4.4. The PEP Viewer and the COPS-PR Viewer

The graphical ***PEP Viewer*** serves as monitoring console for the PEP status. It monitors the PEP and it reports the contents of the PIBs, in terms of installed policies and meta-policies. It also reports which meta-policies are active and enforced in real-time, the meta-policy conflicts and their relative priorities. Through the same tool, the values of the meta-policy Parameters can be checked, as well as certain MIB values. In this way, the system administrator can have a concrete view of the status and the operation of the PEP (Figure 4).

Another graphical tool that we have implemented is the ***COPS-PR Viewer***. This tool can be run either at the PDP or the PEP side and captures all exchanged COPS and COPS-PR messages. This tool was useful in order to debug the COPS and COPS-PR protocol implementation.

Each of these tools is implemented both as JAVA applet and application, so it is possible to execute them either locally or remotely through a web browser.

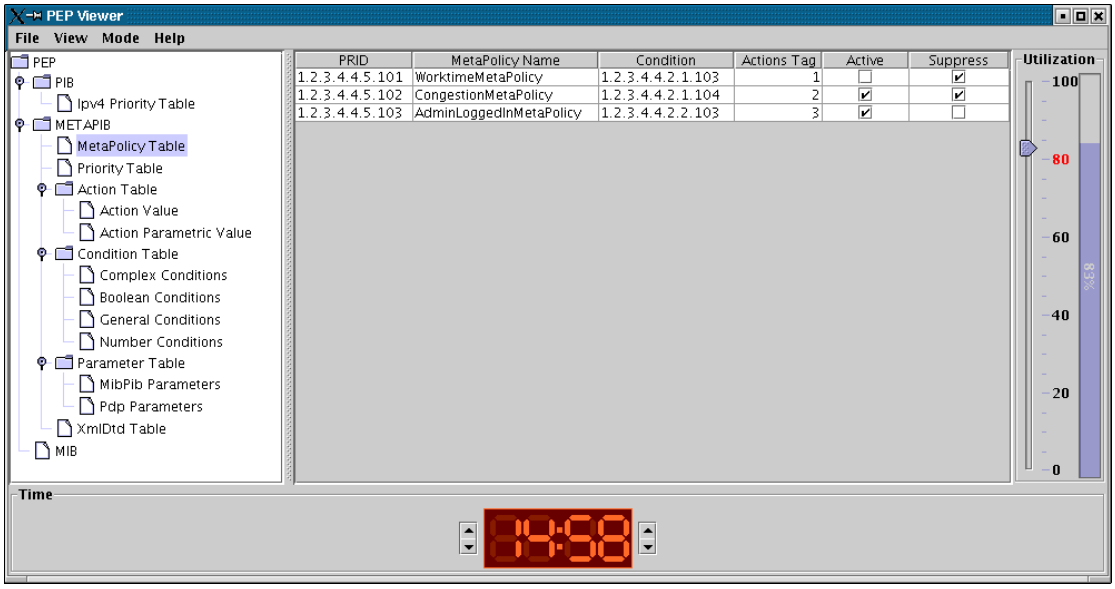

Figure 4: *The PEP Viewer*

4.5. Policy Repository

As stated before, our implementation integrates the policy repository into the PDP. A policy modeling language based on ***xml*** has been defined in order to model, define and store high-level network policies into the policy repository. This model, although simplistic enough, is adequate for the types of policies that our experiments utilize (i.e. IP filtering policies, meta-policies)

4.6. Implementing COPS

As a first step, we implemented the COPS protocol as the base protocol to convey policy information between PDPs and PEPs. COPS itself does not define precisely the format and semantics of the exchanged configuration data: it just provides a way to exchange those data. COPS defines the necessary messages that need to be exchanged, such as messages that open and close connections, carry policies or report errors. Each message is composed of specific COPS objects. However, the exact content of these objects is not described by COPS: the content is defined per client-type basis in additional documents. In this way, COPS can be used to exchange policy data for any client-types, without knowing the semantics of the exchanged data. For a deeper understanding of the protocol, the reader can refer to [13].

The protocol is also implemented in JAVA, consisting in a number of classes that represent the various COPS objects and messages. All COPS classes extend the general **CopsObject** class. This class includes the basic COPS object header (that defines the object and sub-object type, the length of the object and other identifiers) and the object data field. Depending of the type of the object, the type and sub-type fields are set to the respective values. The data field encapsulates the data. The length field is set to the number of octets of the entire object (header plus data). The class also implements methods common to all objects, e.g. for transmitting the object over a stream. Based on that class, the basic COPS objects are defined: Context Object (**ContextObj),** Client Specific Information Object **(ClientSIObj),** Decision Object (**DecObj),** Error Object (**ErrorObj),** Handle Object **(HandleObj),** Keep-Alive Timer Object (**KaTimerObj),** Last PDP Address **(LastPDPAddrObj),** PEP Identification Object **(PepIdObj),** Reason

Object **(ReasonObj)** and Report-Type Object (**ReportTypeObj)**. The objects In-Interface, Out-Interface and LPDP Decision are not used by COPS-PR, and the objects PDP Redirect Address, Accounting Timer Object and Message Integrity Object were not critical for the correct operation and testing of the protocol, and were omitted from our implementation. However, it is not hard for someone interested in them to extend our implementation by including them.

COPS messages are defined as specific combinations of COPS objects. The base class is the **CopsMessage** which contains a header and a valid sequence of COPS objects. The header is mandatory and it is implemented as a separate class (**CopsHeader)**. It stores information like the client type, the message identifier, the length of the entire message, flags, and some other information and provides methods to access and modify those data, transmit the header, etc.

By extending the **CopsMessage** class, all COPS messages are defined: Client-Accept (**CatMessage)**, Client-Open (**OpnMessage),**Client-Close (**CCMessage),** Decision (**DecMessage), Request (ReqMessage),** Delete Request State (**DRQMessage),** Report State (**RPTMessage),** Synchronize State Request (**SSQMessage),** Synchronize State Complete (**SSCMessage)** and Keep-Alive (**KAmessage)**

As described previously, the details of several COPS objects is not defined by the protocol itself, but by the COPS client types. For this purpose we have defined a **ClientHandler** class which provides a way to load several clients of different COPS client-types into the PEP. One such client-type is COPS-PR, the implementation of which is described on the next paragraph (4.7).

Apart from the described classes, several other classes have also been implemented in order to perform several actions (manipulation of these objects, TCP/IP connectivity etc). Description of these classes is out of the scope of this paper.

4.7. Implementing COPS-PR

As an extension of the COPS protocol we implemented the COPS-PR extensions for policy provisioning.

COPS-PR is an extension of COPS in the sense that it further defines specific COPS messages and objects. More specifically, COPS-PR further describes the format of the Request, Decision and Report State Messages and it defines five objects (named Complete Provisioning Instance Identifier, Prefix PRID, Encoded Provisioning Instance Data, Global Provisioning Error, PRC Class Provisioning Error and Error PRID) which are encapsulated within COPS Decision Objects or COPS Client Specific Information Objects.

Similarly to the COPS implementation, a **COPSPRObj** class has been defined that extends the CobsObj and serves as the base class for all the COPS-PR objects. Based on that class, the classes that represent the COPS-PR objects were defined. The most important of them are the **PRIDObj** and the **PPRIDObj** which describe the PRI of the **PIB,** the **EPDObj** which encodes the policy data**,** and the **ErrorPRIDObj** which conveys errors. All policy data is encapsulated in the EPDObj using BER encoding. These data will be installed on a specific PRI of the PIB. The PRI is described by the PRIDObj that identifies the destination.

4.8. Implementing the PIBs

4.8.1. The Filtering PIB

As a first step, a simple custom ***filtering PIB*** was implemented. It has a single PRC that describe source and destination IP addresses and ports of network flows that are allowed/ denied by the network device.

Each PRI of this PRC describes a flow that should be allowed or denied. When a PRI is added to the PIB, a line is added in the access-list of the device. When a PRI is removed, the corresponding line is deleted.

The purpose of implementing this PIB is to serve as a subject for the operations of the meta-policy PIB.

4.8.2. The Meta-Policy PIB

The ***Meta-Policy PIB*** is composed by a group of JAVA classes that represent the defined PRCs: **MetaPolicy, MetaPolicyCondition, MetaPolicyBooleanCondition, MetaPolicyComplexCondition, MetaPolicyAction, MetaPolicyActionValues, MetaPolicyActionParametricValue, Parameter, PdpParameter, MibPibParameter, MetaPolicyPriority and MetaPolicyXmlDtd.** All meta-policy functionality is orchestrated by a coordinator class that is called **MetaPIB**. This class is in charge of adding and removing PRIs in the PIB, checking their correctness and monitoring and enforcing the meta-policies.

When the PEP receives a COPS-PR with a PRI identifier that refers to the meta-policy PIB, the respective data are extracted from the EPD object and placed in the described PRI. The data in the PRIs form conditions and actions of meta-policies. The conditions of the meta-policies are monitored and when they are met, the actions are enforced. The actions are COPS-PR-like data, which describe PRIs in other PIBs in the PEP (in our case they refer to the filtering PIB) and data that must be installed or removed to these PRIs.

4.9. Transforming the Conventional Java Code into ORE Oplets

In order to take advantage of the modern sophisticated Java Development environments, we initially developed and debugged our coda as conventional Java code, and then we transformed it to ORE oplets. The transformation process was simple, requiring only minor modifications in order to implement the minimum API that ORE demands.

More specifically, ORE demands its oplets to implement the minimal API by ***implementing*** an ORE class named "***ManifestOplet***". By ***implementing*** the methods of this class, oplets can be loaded, register as a service, use other ORE services or oplets, start, stop and generally by manipulated by the ORE environment. One of the methods that need to be implemented is the ***startService()*** method which is executed when the ORE starts the oplet service. Apart from the minimal API described above, oplets can implement other ORE APIs that can be used in order to affect the behavior of the router (routing, switching, MIB values etc). In our case, the Filtering PIB oplet had to implement APIs that affect the packet filters of the router.

The described transformations are achieved in the following manner: The initial, conventional Java bytecode was executed in a conventional JVM by loading the class ***StartPEP*** that implemented the ***public static void main()*** method. This class had to be replaced by a new one (***PEPOplet*****)** which ***implemented*** the ***ManifestOplet*** class. Then, the contents of the ***main()*** method of the initial ***StartPEP class*** were copied into the ***startService()*** method of the ***PEPOplet*** with minor modifications. In this manner, the ***PEPOplet.startService()*** had the same behavior with the ***StartPEP.main()***. After these modifications the Java code could be executed within the ORE environment (which, in turn, could be running on a router, on a Unix System or elsewhere).

4.10. Installing the Components into the Router as ORE Services

In order to load the oplets into the router, ORE itself must be downloaded and started first. In order to do so, a special boot image containing JVM is initially loaded to the router through a TFTP server. Then, through the JVM the ORE .jar files are also downloaded, unpacked and executed.

After the ORE is successfully downloaded and started, the oplets themselves must be downloaded and started. In order to do so, the ORE environment is accessed via telnet and specific commands that download and execute the oplets are issued. After the successful completion of the previous steps, the services that the oplets implement are installed on the router and affect its behavior.

4.11. Testing and Results

In order to test the entire model, simple test scenarios were designed:

First of all, the integrity of the COPS implementation was tested. All types of COPS messages and objects were exchanged in order to ensure that our implementation conformed to the IETF standards. After verifying the correct operation of the COPS protocol, similar tests were run upon our COPS-PR implementation.

The next step was to implement the filtering PIB and control it through PDP commands, sent to the PEP through the COPS-PR protocol. Several actions were performed to ensure the correct operation of the PIB (and the underlying protocols) when data are added, replaced or deleted from it.

In parallel, the language for defining policies and meta-policies was also defined and implemented at the PDP and got tested. The use of this language allowed the definition of test scenarios that the PDP would read, implement and enforce to the PEP.

In the same time, the PEP and the COPS-PR Viewers were implemented. Through them we were able to monitor the exchanged COPS-PR messages, the process of installing the policies (and later on, meta-policies) and the content of the PIBs of the PEP.

Finally, the Meta-Policy PIB was implemented and tested, and the PEP Viewer was enhanced to support meta-policies. With simple scenarios, we demonstrated that meta-policies were installed into the PEPs, and through these meta-policies the PEP could self-control the filtering PIB. Our scenarios utilized meta-policies with conditions that included parameters like the time of the day or the load of the interfaces, which the PEP was able to analyze by itself (through additional oplets) without the intervention of the PDP.

4.12. Status and Availability of the Code

The webpage and the source code of the project reside at ***Sourceforge***, and can be found at [27]. Currently the latest version of the ***metapib package*** is the 1.6, and this release includes all the features that we have described in this paper. Note that the code has been tested only against our components; hence the proper communication with third party implementations of the protocol is not guaranteed.

5. DISCUSSION

5.1. Active Networks as a Developing and Testing Environment for Network Protocols

Although the primary goal of our research was to implement and test the functionality of the meta-policy PIB rather than to examine the impact of Active Networks in the development of network protocols, our work gave us significant experience in the latter and several important conclusions were extracted.

The use of an Active Network environment as a means to develop and test new network protocols simplifies the procedure to standard software development and testing, which has numerous advantages:

First of all, the hardware rigidity is overcome and the protocol can be implemented and tested on an actual network device. In the past, only hardware vendors could enjoy such privileges. Independent researchers had to build simulations, which were always questionable and inaccurate, despite the great effort and time spent on them. However, with Active Networks no simulations are needed. Experiments can be materialized on real equipment and in certain cases the experiments can even be conducted on production networks. Such experiments can produce reliable, non-deceiving results, useful for the evaluation and evolution of the protocol. As a subsequent effect, the researchers can conduct research independently of the needs and guidelines set by the vendors.

In addition, traditional Software Engineering techniques can be employed in order to increase the quality and performance of the evolving protocol. For instance, a quick-and-dirty prototype can be initially developed, on which the protocol gradually evolves. The initial prototype only implements the basic functionality, giving feedback regarding the performance and effectiveness of the protocol, and exposing non-anticipated problems. Based on the results of the initial prototype, a newer version is built, which repairs and tweaks the previous version and adds functionality. The procedure is repeated until the final product is reached. With this method, or other similar methods traditionally used in Software Engineering, the final product (protocol) is reached more quickly and its quality and performance is increased significantly.

Besides, the development and testing process results to a ready-to-use product. The final version of the produced software is the protocol itself and it is ready to be used on production-line devices. Provided certain compatibility between different vendors, all or part of the code can be used in order to implement the same protocol on devices of different manufacturer as well. This saves significant effort and resources from the procedure of implementing and testing a new protocol in hardware. Besides, the use of standard software tools (development environments, debugging tools) increases the quality and correctness of the final product.

If the developed protocol is to become a standard, the software nature of the protocol implementation significantly simplifies the procedure of standardization. Independent researchers can share the code, test them on their equipment, create variations and compare them. The evolution procedure can address a wider research community rather than being practically restricted to a group of people or vendors.

5.2. Evaluation of the ORE Environment

The previous section discussed the general advantages of using Active Network environments for developing and testing new protocols. However, the use of ORE results in additional advantages.

ORE is a pure Java software. This means that ORE oplets are encoded in Java, and ORE itself runs within a JVM. The use of a popular language like Java for writing oplets has several advantages:

First of all the development process is significantly simplified. The learning period is minimized since tons of books and other material for learning Java is available and many experienced Java programmers exist. Even more, code is easy to develop and debug. Any Java Development environment can be used, as long as the ORE package is first imported into it. This has enormous impact on the development cycle: Writing code, and especially testing and debugging it, can become extremely hard and tedious without the use of sophisticated tools. Besides, third party Java Libraries or code can be easily reused. For instance, our implementation utilizes certain third-party XML classes. It is worth mentioning that the entire development period for the described oplets was limited to only nine man-months.

Code maintenance and sharing is also made easier since any Java programmer can understand, modify and extend the code. Contribution and continuation of ones' work is much simpler, and the validity of the results of the evaluation of the protocols is strengthened by the fact that the code can be easily understood by a wide range of people.

Finally, the JVM acts as an execution sandbox and prevents oplet malfunctions or security issues from affecting the core functions of the device.

ORE is not the only way to inject conventional code (i.e. code written with conventional tools) into actual routing devices. Other similar approaches also exist, each one of which has its own advantages and disadvantages. A similar, quite popular approach is to use Linux/FreeBSD boxes as active routers: Their software routing mechanism can be modified to achieve "active" behavior. The cost of such boxes is significantly smaller that a hardware router and a combination of tools and languages can be used in order to develop and test a novel protocol. The advantage of ORE over such an approach is that protocols built as ORE oplets can also be used as production protocols on production networks, since they can reside on hardware routers. Software routers are rarely used on production networks, and hence their role in novel protocol deployment is mainly constrained to evaluation and testing purposes.

6. CONCLUSION

This paper demonstrated how the process of defining and testing a new network protocol can be simplified by taking advantage of the properties of Active Networks. This was achieved by describing our experiences of using two programmable routers for deploying and testing two IETF RFC protocols. The case study protocols were the IETF COPS protocol, its extension COPS-PR, and two custom PIBs (data structures utilized by the latter). Although the purpose of our experiments was not to evaluate the effect of Active Networks in the deployment of new protocols, several useful deductions were reached: The Active Environment made the process of developing and testing more easy and quick; neither expensive hardware changes nor inaccurate and time consuming simulations were necessary; common development and debugging tools were used; and the code (which is written in Java) is available for any researcher, vendor or administrator who wants to study, extend or use it.

Of course, there are certain tradeoffs as well. The increased CPU and memory utilization on the network devices is one of them; the incompatibility among different Active Network architectures is another. However, as the CPU and memory resources of the network devices increase and the Active Network technology becomes more mature, we expect that such tradeoffs will be eliminated.

7. REFERENCES

[1] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden; “A Survey of Active Network Research”; *IEEE Communications Magazine,* Vol. 35, No. 1, pp80-86. January 1997.

[2] Konstantinos Psounis; “Active Networks: Applications, Security, Safety, and Architectures”; *IEEE Communications Surveys*, Vol. 2, No. 1, First Quarter 1999.

[3] Smith, J.M., Calvert, K.L., Murphy, S.L., Orman, H.K., and Peterson, L.L.; “Activating networks: a progress report”; *Computer* Vol. 32 4, April 1999, Page(s): 32 –41.

[4] D. L. Tennenhouse and D. J. Wetherall; “Towards an Active Network Architecture”; *Computer Communication Review*, Vol. 26, No. 2, April 1996.

[5] “An Overview of the Oplet Runtime Environment (ORE)” www.openetlab.com/ore.latest/doc/ore/overview.html

[6] “Integrating Active Networking and Commercial-Grade Routing Platforms”; *Usenix Workshop on Intelligence at the Network Edge*, March 2000

[7] Susan J. Shepard; “Policy-based networks: hype and hope“; IT Professional, Vol. 2, No. 1, January-February 2000, pp.12 -16

[8] R. Boutaba, K. El-Guemhioui, P. Dini; "An Outlook on Intranet Management"; *IEEE Communications Magazine, Special issue on Intranet Services and Communication Management*, October 1997, pp.92-97

[9] R. Boutaba, S. Znaty, "An Architectural Approach for Integrated Networks and Systems Management"; *ACM-SIGCOM Computer Communication Review*, Vol. 25, No 5, October 1995, pp. 13-39

[10] M. Sloman; "Policy Driven Management For Distributed Systems"; *International Journal of Network and Systems Management*, Vol. 2, No. 4, December 1994, pp. 333-360

[11] A. Westerinen, J. Schnizlein,J. Strassner, Mark Scherling,Bob Quinn,Jay Perry, Shai Herzog, An-Ni Huynh, Mark Carlson, Steve Waldbusser; "Terminology"; *IETF, Internet-Draft*, draft-ietf-policy-terminology-02.txt, November 2000 (http://www.ietf.org/internet-drafts/draft-ietf-policy-terminology-02.txt)

[12] “Resource Allocation Protocol (rap)”; http://www.ietf.org/html.charters/rap-charter.html

[13] D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, A. Sastry; "The COPS (Common Open Policy Service) Protocol"; *IETF, RFC 2748*, January 2000; (http://www.ietf.org/rfc/rfc2748.txt)

[14] “Policy Based Networking Products, Design and Architecture”; IPHighway, White paper, January 2000.

[15] “Intel COPS client Software Development Kit”; http://www.intel.com/ial/cops/

[16] “COPS Download Page”; http://www.vovida.org/protocols/downloads/cops/

[17] “Internet Engineering Task Force”; http://www.ietf.org/

[18] K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, A. Smith; "COPS Usage for Policy Provisioning"; *IETF, RFC 3084*, March 2001 (http://www.ietf.org/rfc/rfc3084.txt)

[19] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, A. Smith, F. Reichmeyer; “Differentiated Services Quality of Service Policy Information Base”; *IETF, Internet-Draft,* draft-ietf-diffserv-pib-03.txt, March 2001 (http://www.ietf.org/internet-drafts/draft-ietf-diffserv-pib-03.txt)

[20] D. Rawlins, A. Kulkarni, K. Ho Chan, D. Dutt, “Framework of COPS-PR Policy Information Base for Accounting Usage”; *IETF, Internet-Draft,* draft-ietf-rap-acct-fr-pib-01.txt, July 2000 (http://www.ietf.org/internet-drafts/draft-ietf-rap-acct-fr-pib-01.txt)

[21] J. Ottensmeyer, M. Bokaemper, K. Roeber; “A Filtering Policy Information Base (PIB) for Edge Router Filtering Services and Provisioning via COPS-PR”; *IETF, Internet-Draft,* draft-otty-cops-pr-filter-pib-00.txt, November 2000 (http://www.ietf.org/internet-drafts/draft-otty-cops-pr-filter-pib-00.txt)

[22] M. Li, D. Arneson, A. Doria, J. Jason, C. Wang; “IPSec Policy Information Base”; *IETF, Internet-Draft,* draft-ietf-ipsp-ipsecpib-02.txt, March 2001 (http://www.ietf.org/internet-drafts/draft-ietf-ipsp-ipsecpib-02.txt)

[23] M. Fine, K. McCloghrie, J. Seligson, K. Chan; S. Hahn, R. Sahita, A. Smith, F. Reichmeyer; "Framework Policy Information Base", *IETF, Internet-Draft,* draft-ietf-rap-frameworkpib-04.txt, November 2000 (http://www.ietf.org/internet-drafts/draft-ietf-rap-frameworkpib-04.txt)

[24] K. McCloghrie, M. Fine, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, F. Reichmeyer; "Structure of Policy Provisioning Information (SPPI)"; *IETF, Internet-Draft,* draft-ietf-rap-frameworkpib-06.txt, February 2001 (http://www.ietf.org/internet-drafts/draft-ietf-rap-frameworkpib-06.txt)

[25] A. Polyrakis and R. Boutaba; "The Meta-Policy Information Base"; *IEEE Network Magazine,* Special issue on Policy Based Networking, Vol.16 No.2, March/April 2002, pp 40-48.

[26] “The ORE FAQ”; http://www.openetlab.org/docs/openetlab/ORE-FAQ.htm

[27] “Project: Meta-Policy Information Base”; http://sourceforge.net/projects/metapib