Using the Script MIB for Policy-based Configuration Management

P. Martinez, M. Brunner, J. Quittek
Network Laboratories
NEC Europe Ltd.
Adenauerplatz 6
69115 Heidelberg, Germany
{*brunner,quittek*}*@ccrle.nec.de*

F. Strauß, J. Schönwälder, S. Mertens, T. Klie
Computer Science Department
Technical University Braunschweig
Mühlenpfordtstraße 23
38106 Braunschweig, Germany
{*strauss,schoenw,mertens,tklie*}*@ibr.cs.tu-bs.de*

Abstract

The IETF has developed several specifications for policy-based configuration management. In addition, an infrastructure for distributed management by delegation has been specified. This paper combines key concepts developed in these two efforts and proposes a policy-based configuration management architecture built upon the distributed management infrastructure. Two prototype implementations for managing differentiated services are discussed and evaluated.

Keywords

Policy-based Management, Distributed Management, Script MIB, Jasmin

1 Introduction

Policy-based management approaches for the Internet are getting close to maturity. Configuring a high number of routers, bridges, or servers by generic rules instead of individual configuration appears to be less complex, less error-prone and more flexible. This paper focusses on configuration management based on Internet Engineering Task Force (IETF) standards. An overview of further approaches to policy-based management can be found in [1]. The IETF has defined a policy framework [2] consisting of management interfaces for entering policies, repositories for storing policies, policy decision points (PDPs) for evaluating policies, and policy enforcement points (PEPs) for enforcing policy decisions.

Based on this framework, the IETF standardized a policy core information model (PCIM [3], PCIMe [4]) that can be used when entering policies, when storing them in repositories, and when evaluating them at PDPs. For the transfer of policy decisions between PDP and PEP the protocol COPS-PR (Common Open Policy Service for Policy Provisioning) [5] was standardized. The structure of configuration information carried by COPS-PR is defined in Policy Information Bases (PIBs). The language for defining PIBs has been standardizes as the Structure of Policy Provisioning Information (SPPI [6]).

In parallel to these efforts to standardize policy-based management, the IETF has created standards for distributing management functions in a network. The motivation was to move from centralized to weakly or strongly distributed management environments in order to improve scalability and flexibility [7]. The IETF Script MIB [8] allows to send management scripts to distributed mid-level managers. Execution of these delegated management scripts can be controlled either by a higher-level manager or by means of other management functions.

In this paper, we look at a PDP as a mid-level manager which generates concrete device configurations from a set of more abstract policy rules. In particular, we explore how the existing distributed management infrastructure as defined by the Script MIB can be used to control the distribution and execution of policy rules in a network with multiple PDPs, each realized by an instance of the Script MIB. In the extreme case, one can imagine a network where all policy-enabled devices contain a local Script MIB agent acting as PDP.

Our approach is related to the work of the IETF SNMPconf working group. This working group defines a new low-level imperative language for expressing policies in terms of sequences of simple statements that manipulate MIB objects. In addition, they define a MIB for transporting policies written in this new language via SNMP from a repository to a PDP at a managed node. The PDP can be combined with a PEP, such that the managed node evaluates and enforces all policies locally, but also the PDP can use SNMP for configuring other, remote PEPs.

While there are certainly benefits to be expected from defining the SNMPconf language, we believe that defining a MIB for the transfer of policies is needless, because the Script MIB already offers all means required to do so. We prove this by implementing two different solutions of Script MIB policy transfer. They were developed in parallel at the NEC Network Laboratories in Heidelberg and at the Technical University Braunschweig.

The first solution is closely related to the SNMPconf approach. Policies are defined as programs. These are downloaded as 'scripts' via the Script MIB and then executed by a common Script MIB runtime engine. The runtime acts as a PDP and sends configuration information to a local or remote PEP.

The second solution builds upon the Policy Core Information Model (PCIM) [3]. Policies are not defined as programs, but as groups of PCIM objects. Such policies stored in a repository are downloaded using the Script MIB directly to managed nodes. There, an interpreter for this special kind of 'scripts' combines PDP and PEP functionality.

Section 2 introduces the background needed to understand the rest of this paper. Section 3 introduces an architecture for policy-based configuration management using the Script MIB. It also introduces two different approaches to implement a policy engine. Sections 4 and 5 describe the two approaches in more detail. An evaluation of the two approaches is provided in Section 6 before we conclude in Section 7.

2 Background

This section introduces core concepts of the IETF policy framework and the IETF distributed management standards. It also briefly reviews core concepts of the Differentiated Service (DiffServ) architecture which has been used as an example application domain in our prototype implementations.

2.1 Policy Framework

The methodology of policy-based network management was developed within the IETF in the context of the Integrated Services model [2], where it was proposed to use a policy framework for the management of admission control to reservations of network resources. The approach however is independent of the particular service model and soon it was recognized that it can equally be applied in a Differentiated Services [9] environment and provides help in the application of IPsec. Moreover, it can be applied favorably to other more general management problems.

Unfortunately, there is no generic IETF policy framework architecture fully agreed upon and specified. But the key functional blocks seem to be commonly recognized: policy management application, policy repository, policy decision point (PDP) and policy enforcement point (PEP), as shown in Figure 1 a).

The policy management application provides the interface to the network administrator to create and deploy policies, store them in the repository and monitor the status of the policy-managed environment. This application performs a simple validation that checks for potential policy conflicts. The policy repository is a storage that is used for policy retrieval performed by the policy decision points. Access to the database is accomplished by a repository access protocol.

The policy decision point is the point where policy decisions are made. It performs the functions of retrieving and interpreting policies, detecting policy conflicts, receiving policy decision requests from PEPs, and returning policy decisions to them. Triggers to evaluate one or more policy rules can be events, polling, and explicit system/component requests. Please note that the IETF policy framework does not include triggers explicitly. Only conditions (including timer conditions) are included. However, for an implementation of the framework, triggers are a common choice.

The PDP makes policy decisions based on policy conditions that are formed of boolean expressions that may refer to network element attributes. If a condition is evaluated to be true, the according action is executed, which typically (re)configures target elements to enforce the policy. If it is necessary it will translate policy rules into more specific parameters that the PEP could understand. A PDP may control multiple PEPs but each PEP is controlled by one PDP.

The PEP is the target entity that hosts the network elements where policy decisions are actually enforced. It is the target of a policy action being executed when the rule condition evaluates to true. The separation of PEP and PDP (and also of the policy repository) is a logical one based on functionality, and not necessarily a physical separation. PEP and PDP may be combined and co-located.

The IETF framework suggests LDAP [10] as the protocol for repository access and COPS/COPS-PR as the protocol for policy decision transfer. Other mechanisms such as HTTP, FTP, or SNMP may be used as well. However, no protocols are suggested for the communication between the policy management application and the PDPs or among cooperating PDPs. In general, no implementation details such as distribution, platform, protocols or language are prescribed.

The applicability of a policy can be specified by assigning a role to it. The concept of role is central to the design of the entire policy framework. A role is a type of attribute that is used to select one or more policies for a set of entities and/or components from among a much larger set of available policies. The idea behind roles is simple. A policy administrator assigns each resource one or more roles, and then specifies the policies for each of these roles. The policy framework is then responsible for configuring each of the resources associated with a role in such a way that it behaves according to the policies specified for that role.

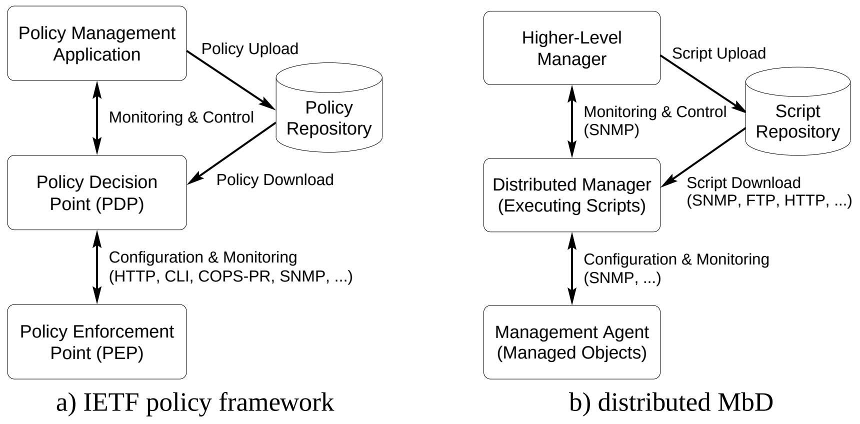

Policy Management Application
Policy Upload
Policy Repository
Monitoring & Control
Policy Decision Point (PDP)
Policy Download
Configuration & Monitoring (HTTP, CLI, COPS-PR, SNMP, ...)
Policy Enforcement Point (PEP)
a) IETF policy framework
Higher-Level Manager
Script Upload
Script Repository
Monitoring & Control (SNMP)
Distributed Manager (Executing Scripts)
Script Download (SNMP, FTP, HTTP, ...)
Configuration & Monitoring (SNMP, ...)
Management Agent (Managed Objects)
b) distributed MbD

Figure 1: Architectures of the IETF policy framework and of distributed management by delegation.

2.2 Distributed Management by Delegation

The delegation of management functions is a well known technique for the creation of scalable and flexible management systems [11]. The IETF Script MIB [8] integrates this approach into the Internet management framework [7, 12].

The Script MIB provides capabilities to (i) transfer management scripts to distributed managers, (i) initiate and terminate the execution of management scripts, (iii) transfer arguments, (iv) monitor and control running management scripts, (v) transfer results and error indications produced by management scripts.

The key components and their interactions are summarized in Figure 1 b). The Script MIB is implemented on the distributed manager and used by a higher-level manager to monitor and control the execution of delegated management scripts. The higher-level manager can monitor and control script execution via SNMP. The distributed manager can download scripts from a script repository using FTP or HTTP. Access to the management agents is subject to the management scripts, also here, SNMP plays an important role.

It should be noted that the Script MIB itself makes no assumption about the format of management scripts and supports arbitrary programming languages and multiple execution environments. Since Figures 1 a) and b) look very similar, it is reasonable to investigate how the two approaches can be combined and integrated.

2.3 Differentiated Services

As an example of an application domain for our system we regard the configuration of DiffServ nodes. According to the DiffServ architecture [9], traffic entering a network is classified and conditioned at boundary nodes of the network, and associated to different behavior aggregates identified by DiffServ code-points (DSCPs) [13]. Within the core network, interior nodes examine only the DSCP field of incoming packets and forward them according to the per-hop behavior (PHB) associated with those DSCPs. A PHB defines how a packet should be treated by a DiffServ node, before being transmitted to the next hop. In order to implement a PHB, a DiffServ node has several components that form building blocks as presented in the informal management model for DiffServ routers [14]. The model includes abstract definitions for traffic classification, metering, marking, dropping, queuing, and scheduling.

The device-level configuration of the elements is realized through a configuration and management interface via one or more management protocols, such as SNMP or COPS, or via other configuration mechanisms such as command line interfaces running over SSH or TELNET. Some of the management interfaces are currently being standardized within the IETF. The IETF DiffServ working group is developing a DiffServ MIB [15] and a DiffServ PIB [16] for managing DiffServ devices via SNMP and COPS-PR. Additionally, the Policy Framework working group focuses on a CIM information model for DiffServ nodes [17].

3 Architecture

This section describes our general policy-based configuration management architecture using the Script MIB shown in Figure 2. It contains four kinds of components: policy manager, policy repository, PDP, and PEP.

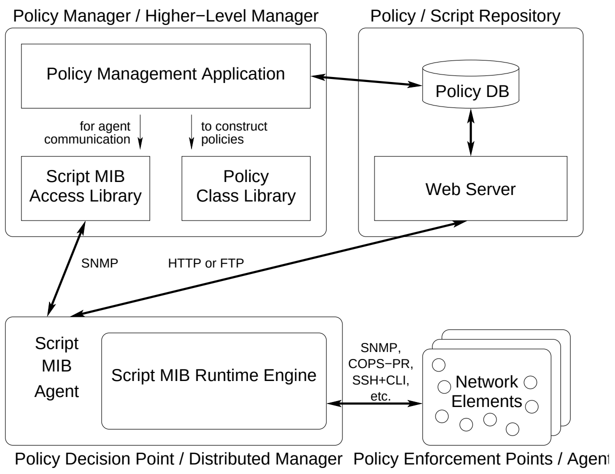

Policy Manager / Higher−Level Manager
Policy / Script Repository
Policy Management Application
Policy DB
for agent communication
to construct policies
Script MIB Access Library
Policy Class Library
Web Server
SNMP
HTTP or FTP
Script MIB Agent
Script MIB Runtime Engine
SNMP, COPS−PR, SSH+CLI, etc.
Network Elements
Policy Decision Point / Distributed Manager
Policy Enforcement Points / Agent

Figure 2: Architecture of the Script MIB based configuration management system.

The policy manager controls the entire policy-based management system. It provides a user interface which allows administrators to construct and edit policies and to store them in the policy repository. To control and monitor the execution of policies, the management application communicates with the agent(s) via SNMP.

The policy manager must be aware of the roles associated to each network element, since it performs a policy selection in order to know which policies each agent must retrieve. The concept of roles is used to select the policies which apply to a certain network element and therefore need to be downloaded to the Script MIB agent. In our architecture, the policy management application needs to keep track of the roles of the network elements controlled by the Script MIB agent, and what policies apply for what role.

The Script MIB agent acts as PDP, retrieving policies from an HTTP or FTP server that is acting as a policy repository. We are regarding the runtime engine of the Script MIB agent as a policy engine that evaluates the policies delegated to it as 'scripts'. A policy evaluation may require monitoring network elements and executing actions by sending configuration messages to the network elements.

There are two manager-agent relationships, one between policy manager and PDP and one between PDP and PEP. For communication between policy manager and PDP, SNMP and the Script MIB are used. For communication between PDP and PEP several alternatives can be used: SNMP, COPS-PR, SSH/CLI, or a local interface, if PDP and PEP are co-located.

While we assume only one instance of policy manager and policy repository, there may be multiple PDPs and for each PDP there may be multiple PEPs. The architecture covers three levels of PDP distribution shown in Figure 3: (a) centralized with just a single PDP, (b) weakly distributed with several PDPs, but much less than the number of PEPs, and (c) strongly distributed with one PDP per PEP.

For centralized policy management there is no specific advantage in using the Script MIB compared to other policy management systems. For weakly distributed policy management we gain scalability. The central PDP in (a) may become a bottleneck, if the number of policies and/or the number of PEPs increase too much. In (b) the bottleneck is removed by distributing the load to as many PDPs as required.

In case of strongly distributed policy management, all network elements hosting a PEP also host a PDP. Here, no standardized communication between PDP and PEP is required anymore, because local proprietary communication may be used. The

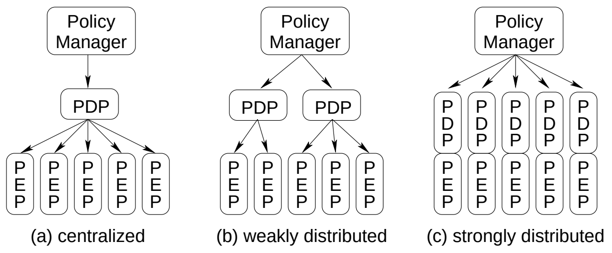


Figure 3: Different levels of PDP distribution.

scalability is still higher than in (a) because policy information is considered to be more condense than the configuration information derived from it and because the policy manager now can select which policy to send to which network element. For example, in a DiffServ network management system with all routers hosting PDPs and PEPs, core routers would only receive and evaluate core router policies, while edge routers receive different and potentially more complex policies.

3.1 Two Approaches for a Policy Engine

In order to make the Script MIB agent behave as a PDP, we must regard the runtime engine designed to execute scripts as a policy execution engine that evaluates policies. The following sections describe implementations of two different approaches for the handling and processing of policies with the Script MIB.

The first approach represents policies by program code. This matches the typical use of the Script MIB. A policy or a group of policies are represented by a program that is passed as a "script" to a Script MIB agent. At the agent the program is executed by a runtime engine for the used programming language. This runtime engine must provide a way of accessing the network elements to be configured by the policies, e.g. by offering a specific library.

The second approach represents policies by objects. A policy or a group of policies are represented by a set of objects. Again, the set is passed as a "script" to a Script MIB agent. There, the objects are evaluated by a specific policy runtime engine. The objects representing policies conform to PCIM, they contain data only and no code. Also, they are specific to an application domain, e.g. QPIM [18] for QoS. The policy runtime engine must contain implementations of all PCIM policy classes that are to be evaluated, and it must have access to the network elements to be configured by these policies.

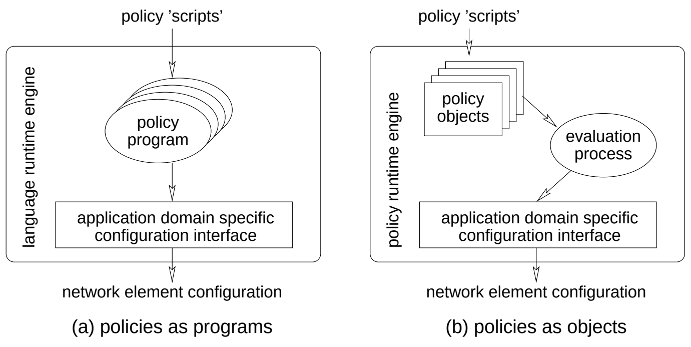


Figure 4: Two different approaches to policy-based management with the Script MIB.

4 Approach 1: Policies as Programs

This approach, shown in Figure 5, is based on an existing Script MIB runtime engine. Policies are represented as scripts written in a language supported by that runtime engine. A general policy management language extension provides interfaces for deriving and implementing policies, rules, conditions, actions, network elements, event generators and events.

Domain specific language extensions provide abstract interfaces to network elements of a specific policy application domain. They allow policy scripts to retrieve element attributes and event notifications and to correlate them to make policy decisions, so that they can in turn be used to configure network elements. Drivers realize the mapping between the domain specific interfaces and the underlying device-level mechanism to actually configure the network elements.

We developed a prototype implementation of these libraries based on the Jasmin Script MIB implementation [19] . This prototype is described in the following section, while Section 4.2 gives a simples policy script example. A class diagram of the libraries along with the example is shown in Figure 6.

4.1 Implementation

The `policyMgmt` package contains the classes and interfaces `Policy`, `Condition` and `Action` that are usually implemented by policy scripts: A `Policy`-derived class represents the main class that is executed when the policy script is started. It registers a number of newly instantiated `Rule`s that in turn register implementations of the `Condition` and `Action` interfaces. Since these classes can be programmed individually, they can benefit from the whole Java code flexibility.

Rules are triggered by `Event`s, which are raised by `EventGenerator`s. Typi-

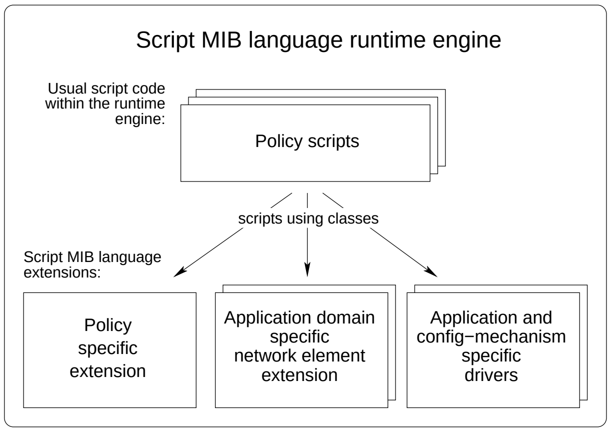
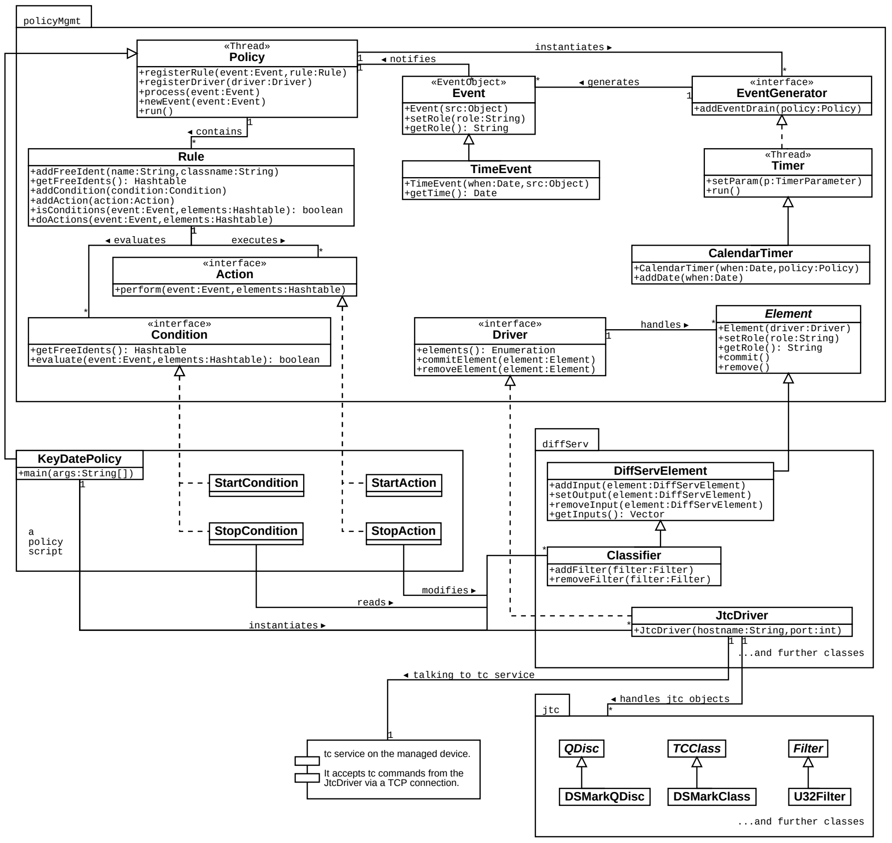

Script MIB language runtime engine
Usual script code within the runtime engine:
Policy scripts
scripts using classes
Script MIB language extensions:
Policy specific extension
Application domain specific network element extension
Application and config−mechanism specific drivers

Figure 5: The standard Script MIB runtime engine is executing policy ‘scripts’ that use policy-supporting language extensions.

cal events are time-based (`TimeEvent` raised by `Timer`), e.g. periodic or calendar-based one-shot timers (`CalendarTimer`). So far, our implementation is restricted to time events, but the architecture is open for other kinds, such as monitoring events. The abstract class `Element` is the parent of all network elements modeled by domain specific packages. Elements are handled through a `Driver` interface.

Our targeted application domain is the configuration of DiffServ nodes. Hence, our domain specific class package is named `diffServ` and contains element classes to represent classifiers, filters, meters, actions, droppers, queues, schedulers, etc., that can be created, deleted and modified. In accordance to the DiffServ MIB all these data path elements can be plugged together (with certain restrictions) through methods provided by the common parent class `DiffServElement`, which in turn is a child class of the `policyMgmt.Element` class.

The class `JtcDriver` represents the adapter between the protocol independent



policyMgmt
«Thread» Policy
+registerRule(event:Event,rule:Rule)
+registerDriver(driver:Driver)
+process(event:Event)
+newEvent(event:Event)
+run()
◂ notifies
instantiates ▸
«EventObject» Event
+Event(src:Object)
+setRole(role:String)
+getRole(): String
◂ generates
«interface» EventGenerator
+addEventDrain(policy:Policy)
◂ contains
Rule
+addFreeIdent(name:String,classname:String)
+getFreeIdents(): Hashtable
+addCondition(condition:Condition)
+addAction(action:Action)
+isConditions(event:Event,elements:Hashtable): boolean
+doActions(event:Event,elements:Hashtable)
TimeEvent
+TimeEvent(when:Date,src:Object)
+getTime(): Date
«Thread» Timer
+setParam(p:TimerParameter)
+run()
◂ evaluates
executes ▸
«interface» Action
+perform(event:Event,elements:Hashtable)
CalendarTimer
+CalendarTimer(when:Date,policy:Policy)
+addDate(when:Date)
«interface» Condition
+getFreeIdents(): Hashtable
+evaluate(event:Event,elements:Hashtable): boolean
«interface» Driver
+elements(): Enumeration
+commitElement(element:Element)
+removeElement(element:Element)
handles ▸
Element
+Element(driver:Driver)
+setRole(role:String)
+getRole(): String
+commit()
+remove()
KeyDatePolicy
+main(args:String[])
a policy script
StartCondition
StartAction
StopCondition
StopAction
diffServ
DiffServElement
+addInput(element:DiffServElement)
+setOutput(element:DiffServElement)
+removeInput(element:DiffServElement)
+getInputs(): Vector
Classifier
+addFilter(filter:Filter)
+removeFilter(filter:Filter)
modifies ▸
reads ▸
instantiates ▸
JtcDriver
+JtcDriver(hostname:String,port:int)
...and further classes
◂ talking to tc service
◂ handles jtc objects
jtc
tc service on the managed device.
It accepts tc commands from the JtcDriver via a TCP connection.
QDisc
TCClass
Filter
DSMarkQDisc
DSMarkClass
U32Filter
...and further classes

Figure 6: Class diagram of (a) the policy management package `policyMgmt`, (b) the DiffServ domain specific package `diffServ`, and (c) the Linux tc specific driver `jtc`. A policy script (d) `KeyDatePolicy` makes use of these components.

classes of the `diffServ` package and the device specific configuration mechanism. In our prototype we have implemented support for the Linux 2.4 “tc” (traffic conditioning) subsystem through the Java class package `jtc` which represents the tc data structures. This allows the `JtcDriver` to manage tc configurations mapped from the model supported by the `diffServ` package. In order to write tc configuration to the kernel, we have implemented a simple TCP service that accepts tc commands sent by the `JtcDriver`.

4.2 Example

The following example shows a policy script that contains a pair of rules: a `StartRule` that gets triggered by a `CalendarTimer` at a specific point in time, e.g. at 23:00 on December 31, and a `StopRule` that gets triggered some time later, e.g. at 2:00 on January 1. The conditions evaluate unconditionally to true in both cases. The `StartAction` doubles a specific bandwidth parameter and after the critical time period the `StopAction` resets it to the start value. This might be reasonable to allow mobile phone traffic to carry the expected increased bandwidth demand during that time. The Java code below shows the essential parts of a policy script that supports this scenario.

```
public class KeyDatePolicy extends Policy {
    public class StopCondition implements Condition {
        public Hashtable getFreeIdents() { return new Hashtable(); }
        public boolean evaluate(Event event, Hashtable elements) {
            // check whether the bandwidth is ok again...
            return checkBandwidth(elements);
    }   }
    public class StartAction implements Action {
        public void perform(Event event, Hashtable element) {
            defaultRate = tokenBucket.getRate();
            tokenBucket.setRate(defaultRate * 2);
            try { tokenBucket.commit(); } catch (IOException e) {}
    }   }
    public class StopAction implements Action {
        public void perform(Event event, Hashtable element) {
            tokenBucket.setRate(defaultRate);
            try { tokenBucket.commit(); } catch (IOException e) {}
    }   }
    public KeyDatePolicy(String[] args) {
        // setup the driver and the general DiffServ config...
        // setup the start and stop calendar timers...
        Date startDate = formatter.parse("31.12.2001 23:30:00");
        Date stopDate  = formatter.parse("01.01.2002 02:00:00");
        CalendarTimer startTimer = new CalendarTimer(this, startDate);
        CalendarTimer stopTimer  = new CalendarTimer(this, stopDate);
        startTimer.start(); stopTimer.start();
        // setup the policy rules
        Rule startRule = new Rule(); Rule stopRule  = new Rule();
        startRule.addCondition(new AlwaysCondition());
        startRule.addAction(new StartAction());
        stopRule.addCondition(new AlwaysCondition());
        stopRule.addAction(new StopAction());
        this.registerRule(startTimer, startRule);
        this.registerRule(stopTimer, stopRule);
    }
    public static void main (String[] args) {
        KeyDatePolicy policy = (new KeyDatePolicy(args));
        policy.start();
        try { policy.join(); } // ...
}   }
```

5 Approach 2: Policies as Objects

In our second approach, we coded policies not as programs running independently of each other, but as objects (or sets of objects) that are executed by a single policy evaluation process, see Figure 7. Therefore, the runtime engine cannot be anymore an interpreter of a common programming language. A special runtime engine for policy objects is required.

In order to use existing standards as much as possible, we chose PCIM and standards derived from PCIM as information model for the policy objects. Implementations of the classes these objects are instances of, must be available at the runtime engine. The policy evaluation process executes policy actions by using application domain specific interfaces.

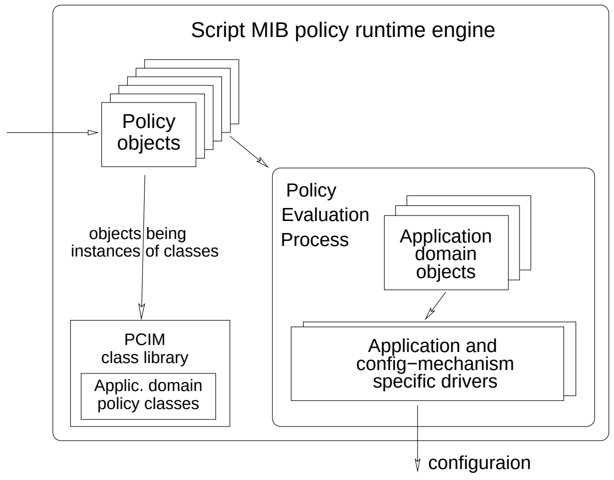

Script MIB policy runtime engine
Policy objects
objects being instances of classes
PCIM class library
Applic. domain policy classes
Policy Evaluation Process
Application domain objects
Application and config−mechanism specific drivers
configuraion

Figure 7: Implementation of the policy runtime engine.

The functions that the policy runtime engine must offer are:

- Monitor domain specific attributes using domain specific objects that represent the configuration state of the network elements controlled by the Script MIB agent. These objects might use driver functions to provide an abstract interface to configure the underlying specific domain implementation.
- Receive orders from the Script MIB agent to add, remove, enable or disable policies.
- Handle the triggering of policies. The event that triggers the evaluation of a policy is encoded within the delegated policy object.
- Identify the set of target domain objects to which the triggered policy applies. The role of the policy is specified within the policy object.
- Compare the values of selected domain object attributes with the conditions specified within the triggered policy object.

- Execute the actions of a policy rule after the corresponding condition was evaluated to true. In general these actions are performed by modifying certain attributes of the domain objects regarding the information encoded in the policy action.
- Prioritize policies according to their priority attribute.

5.1 Implementation

We restricted the implementation of this approach to the configuration of local managed objects. This implies that each PEP has a co-located PDP that exclusively manages the PEP. For implementing the policy runtime engine, we built upon the Java language runtime engine already used in the approach described in Section 4. We replaced the default Java class loader with a class loader for serialized Java objects.

Policy classes for the PCIM package were implemented as a one-to-one mapping of PCIM and QPIM to Java objects. QPIM is the IETF QoS Policy Information Model [18], which conforms with PCIM and its extensions [4]. Note that PCIM and QPIM are pure information models. Therefore, the classes of the PCIM package do mainly contain data structures, but no specific methods. All code for evaluating policy objects is contained in a `PolicyEvaluator` class defining the main loop of the policy evaluation process. It contains all functions listed as required above.

As an interface for configuring the local DiffServ implementation, we developed a class package for the DiffServ application domain which provides a virtual representation of the traffic conditioning and per-hop behavior applied to data flows at a DiffServ router. These classes and their relationships are based on the DiffServ Informal Management Model [14] and the Information Model for Describing Network Device QoS Datapath Mechanisms [17].

We tested this implementation in a DiffServ test-bed containing Linux routers with the NEC Linux DiffServ implementation [20] developed jointly with the University of Bern. The local driver translates calls from DiffServ Java configuration methods into a respective set of commands and profile updates.

5.2 Example

Regarding the example introduced in Section 4.2, the policy is represented by one policy rule. It increases the capacity of the Expedited Forwarding (EF) PHB on each router in a specific time period (see Figure 8).

Once the policy object (a Java object) is received by the Script MIB, the `PolicyEvaluator` checks the policy rule in order to derive triggering events for that particular policy from the `PolicyTimePeriodCondition` and stores them in its internal event table. The event table keeps track of when which policies need to be triggered. In the current implementation, we store an event only for the start time of the time period, because we do not role-back changes after the time period has expired. Therefore a different policy rule is needed for restoring the original state after the policy rule is not valid anymore. The policy rule itself is stored in the list of policy rules and awaits to be triggered by the policy evaluator's time management.

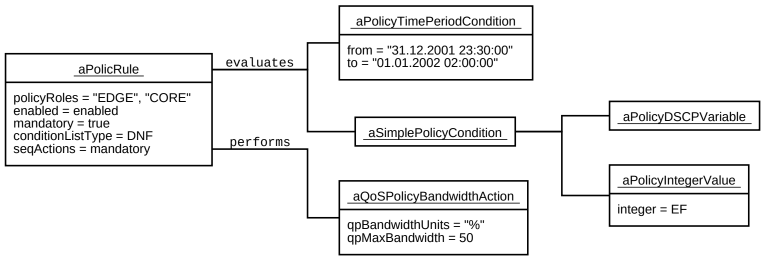


Figure 8: HappyNewYear policy object.

When the policy is triggered, the `PolicyEvaluator` reads the value of the `policyRoles` property and identifies the set of interfaces that match the roles. Then the `PolicyEvaluator` evaluates for each of these interfaces the condition. In this case, the DiffServ element objects which are in the data path of the EF PHB are searched, and the scheduling element of the path is configured by the action in case EF is configured on the interface. In this example, the `QoSPolicyBandwidthAction` increases the maximum bandwidth, the EF PHB is allowed to use.

After all the changes on DiffServ element objects by one policy rule are made, the pertinent accessor functions are called. The new parameters are translated into *tc* commands (and for other examples in configuration file changes) of the underlying NEC Linux-DiffServ implementation.

6 Evaluation

The Script MIB-based policy management architecture we introduced in Section 3 can be used for a wide range of policy-based management systems including centralized single PDP solutions, weakly distributed solutions with high scalability, and strongly distributed solutions with one PDP per PEP.

Our architecture covers the main functionality of the SNMPconf target without requiring a new MIB to be developed. When used with one PDP per PEP, also no standard for transferring configuration information, such as a Policy Information Base module, is required. By using the SNMP Script MIB, our architecture integrates well into existing Internet management frameworks and offers high security as shown in [12].

Comparing the two approaches we investigated for realizing the architecture, policies as programs (PaP) and policies as objects (PaO), we observed the following differences: Definitely, PaP is much more **expressive and flexible**, because each policy is coded explicitly as a program, that can be tailored individually and that is only limited by the programming language and the runtime system. In contrast, PaO is tightly bound to a set of predefined (standardized) policy classes that must be used for policy definition. Yet, the predefined set should in general be sufficient for all (or most) useful and required policies, because this is a key issue when defining the set.

While PaP has much more flexibility and expressiveness, PoP should have exactly as much as is required.

Safety, i.e. protection against defining wrong or bad policies by mistake, is obviously higher for PaO, because only attributes of policies and associations between policies can be set. In general, the typical trade-off between expressiveness/flexibility and safety applies. The more expressive a language is, the higher are chances of making mistakes, for example coding infinite loops in policies.

Because the Script MIB does not offer means for specifying the priority of a script, **prioritization of policies** cannot be supported by our first approach. In the second approach, the runtime engine can easily check the priority attribute of policies and schedule their evaluation accordingly.

Since PaP uses independent processes or threads for each policy, it does support **concurrent evaluation of policies**. Our PaO implementation just supports sequential evaluation of policies. In general, it would be possible to implement concurrent evaluation also for PaO, for example by switching between policies, when the current one waits for I/O, but we estimate the implementation effort rather high.

User-friendliness very much depends on the editor/toolkit used for generating policies as programs or objects. A PaO editor must support creation of policy objects, setting their attributes, and associating them with other objects. A PaP toolkit must assist the user in writing policy programs. For PaP standard programming language toolkits may be used, although specific support for managing the policies is desirable when larger numbers of policies are to be written.

The separation of policies into independent ”scripts” inhibits **re-use of ”scripts”** among different policies with PaP, because each ”script” runs its own policy decision engine that cannot access data structures of other ”scripts”. Only policies coded into the same ”script” may share code. This is different for PaO. Here, all objects are evaluated by the same decision engine. Several policies can share conditions or components of conditions as well as actions or components of actions.

For our implementations of the two approaches, we found similar **”script” sizes** for corresponding policy implementations. For PaO, the ”scripts” contained superfluous data, because the policy objects contained values for all their attributes including unused ones. For PaP only the values of required attributes are coded in set operations, but here there is a lot of program code for the policy implementation that is very similar for each policy, but that cannot be omitted.

Concerning **runtime requirements** PaP and PaO differ significantly for large numbers of policies, because PaP create a thread or process for each policy. Our implementation based on heavy-weighted Java threads consumed a lot of memory (approx. 5MB per ”script”). The overhead might be less for other runtime systems, but in principle the additional time requirement for creating new threads/processes and the additional memory requirement remain.

The **implementation effort** of the runtime system appeared to be comparable for PaP and PaO. For PaP more effort was required for thread control, while for PaO the policy evaluation and prioritization consumed additional man power. However, there is a big difference in re-usability of the runtime engine. While the PaO engine is only usable for policy evaluation, the PaP engine includes a full runtime engine for scripts in a standard programming language, that can be used for other (management) purposes as well, or that is already available anyway. In the latter case, the implementation effort for PaP would of course be much lower than for PaO.

The evaluation shows advantages and disadvantages for both approaches. Summarizing, we think that policies as programs is the more open approach, which allows more deviations from the pure policy-based management approach and which is more flexible in unforseen situations. In contrast, PaO appears to be the better choice for well defined and dedicated policy-control problems, particularly, when PCIM-based solutions are desired.

7 Conclusions

This paper introduces an architecture for policy-based configuration management using the Script MIB. The architecture matches exactly the IETF framework for policy-based management as well as the IETF framework for distributed management by delegation. Thereby, it offers several PDP distribution models for customized scalability and flexibility. The architecture shows an alternative to the approach chosen by the IETF SNMPconf working group, but it is completely based on standards that already exist.

We found, implemented, and evaluated two different ways of implementing our architecture: one representing policies by programs and one representing policies by objects. Both have proven to be feasible and suitable when applied to DiffServ network management. Policies as programs are more flexible using a standard programming language, while policies as objects seamlessly integrate into systems based on the IETF policy framework. An example policy demonstrates the differences between the approaches.

In the future, the Java-based solution might be replaced by one specific to policy definitions, e.g., we could integrate a runtime engine for a policy specification language such as Ponder [21] at the Script MIB agent.

References

[1] G.N. Stone, B. Lundy, and G.G. Xie. Network Policy Languages: A Survey and a New Approach. *IEEE Network Magazine*, 15(8):10–21, January 2001.

[2] R. Yavatkar, D. Pendarakis, and R. Guerin. A Framework for Policy-based Admission Control. RFC 2753, Intel, IBM, U. of Pensylvania, January 2000.

[3] B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. Policy Core Information Model – Version 1 Specification. RFC 3060, IBM, LongBoard Inc, Cisco Systems, January 2001.

[4] B. Moore, L. Rafalow, Y. Ramberg, Y. Snir, J. Strassner, A. Westerinen, R. Chadha, M. Brunner, and R. Cohen. Policy Core Information Model Extensions. Internet Draft <draft-ietf-policy-pcim-ext-06.txt>, IBM, Cisco Systems, Telcordia Technologies, NEC, Ntear LLC, November 2001.

[5] K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, and A. Smith. COPS Usage for Policy Provisioning (COPS-PR). RFC 3084, Nortel Networks, Intel, Cisco, IPHighway, PFN, Allegro Networks, March 2001.

[6] K. McCloghrie, M. Fine, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, and F. Reichmeyer. Structure of Policy Provisioning Information (SPPI). RFC 3159, Cisco Systems, Nortel Networks, Intel, Allegro Networks, PFN, August 2001.

[7] J. Schönwälder, J. Quittek, and C. Kappler. Building Distributed Management Applications with the IETF Script MIB. *IEEE Journal on Selected Areas in Communications*, 18(5):702–714, May 2000.

[8] D. Levi and J. Schönwälder. Definitions of Managed Objects for the Delegation of Management Scripts. RFC 3165, Nortel Networks, TU Braunschweig, August 2001.

[9] S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. An Architecture for Differentiated Services. RFC 2475, Torrent Networking Technologies, EMC Corporation, Sun Microsystems, Nortel UK, Bell Labs Lucent Technologies, Lucent Technologies, December 1998.

[10] M. Wahl, T. Howes, and S. Kille. Lightweight Directory Access Protocol (v3). RFC 2251, Critical Angle Inc., Netscape Communications Corp., Isode Limited, December 1997.

[11] Y. Yemini, G. Goldszmidt, and S. Yemini. Network Management by Delegation. In *Proc. International Symposium on Integrated Network Management*, pages 95–107, April 1991.

[12] J. Schönwälder and J. Quittek. Secure Internet Management By Delegation. *Computer Networks*, 35(1):39–56, January 2001.

[13] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, Cisco Systems, Torrent Networking Technologies, EMC Corporation, December 1998.

[14] Y. Bernet, S. Blake, D. Grossman, and A. Smith. An Informal Management Model for Diffserv Routers. Internet Draft <draft-ietf-diffserv-model-06.txt>, Microsoft, Ericsson, Motorola, Allegro Networks, February 2001.

[15] F. Baker, K. Chan, and A. Smith. Management Information Base for the Differentiated Services Architecture. Internet Draft <draft-ietf-diffserv-mib-16.txt>, Cisco Systems, Nortel Networks, Allegro Networks, November 2001.

[16] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, C. Bell, A. Smith, and F. Reichmeyer. Differentiated Services Quality of Service Policy Information Base. Internet Draft <draft-ietf-diffserv-pib-05.txt>, Cisco Systems, Nortel Networks, Intel, Allegro Networks, PFN, November 2001.

[17] B. Moore, D. Durham, J. Halpern, J. Strassner, A. Westerinen, and W. Weiss. Information Model for Describing Network Device QoS Datapath Mechanisms. Internet Draft <draft-ietf-policy-qos-device-info-model-06.txt>, IBM Corporation, Intel, Longitude Systems, Intelliden Inc, Cisco Systems, Ellacoya, November 2001.

[18] Y. Snir, Y. Ramberg, J. Strassner, and R. Cohen. Policy QoS Information Model. Internet Draft <draft-ietf-policy-qos-info-model-04.txt>, Cisco Systems, Ntear LLC, November 2001.

[19] TU Braunschweig, NEC C&C Research Laboratories, http://www.ibr.cs.tu-bs.de/projects/jasmin/. *Jasmin - A Script MIB Implementation*, 1999.

[20] T. Braun, M. Scheidegger, H. Einsiedler, G. Stattenbergerand, K. Jonas, and H. J. Stüttgen. A Linux Implementation of a Differentiated Services Router. Technical report, Network and Services for Information Society (INTERWORKING2000), October 2000.

[21] N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. Technical report, Departament of Computing, Imperial College, January 2001.