Department of Computing
Imperial College London
University of London

An Adaptive Policy Based Framework for Network Management

Leonidas A. Lymberopoulos

A thesis submitted in partial fulfilment of the requirements for the degree of Doctor of Philosophy in the Faculty of Engineering of the University of London, and for the Diploma of the Imperial College London

London, October 2004

Στους γονείς μου,

Αντρέα και Γεωργία

To my parents,

Andreas and Georgia

Abstract

Policy-based management has emerged as a promising solution for the management of large-scale and heterogeneous networks. This approach has been adopted in several network management areas, such as in the areas of Quality of Service (QoS) and security management. However, although policy-based management has been the subject of considerable research, proposed solutions are often restricted to static condition-action rules where conditions determine when actions should be performed on the managed entities. The static policy configurations require manual intervention to cater for configuration changes and to enable policy deployment. However, changes in the system such as QoS violations, network failures or denial of service attacks in a secured network may require adaptation of existing policies to new circumstances. Thus, policies themselves need to be managed and adapted.

Policies define management strategies for network devices, access control systems or internet services. However, little work has been done on validating whether the policies will lead to a feasible implementation for the specific environment to which they apply. Validation requires checking that the policy is consistent with the functional or resource constraints within the target environment. For example, one can check whether the policies assume functionality or specific operations, which do not exist in target devices, or bandwidth in excess of the capacity of data links. Where possible, static checking should be done prior to policy deployment in order to detect invalid policies at design time, but there are some policies, related to resource allocation, that depend on the current state of the system, and require constraints, specified within the policy rules themselves, that must be checked dynamically at execution time.

This thesis introduces a novel framework to address the problems identified above. Our framework supports automated policy deployment and dynamic adaptation of policy in response to changes within the managed environment. It implements a flexible architecture where policy adaptation is specified and enforced by other policies, specified in the same Ponder policy notation. Policy adaptation includes both dynamically changing policy parameters and selecting which policies should be selected from a set of predefined ones to be enforced within the network. Furthermore, using the Common Information Model (CIM) as the modelling framework for managed entities, we provide solutions for validating policies with respect to the capabilities of the target network environment.

Acknowledgements

First of all, I would like to thank my supervisor, Professor Morris Sloman for his constant support during my PhD studies. This thesis would certainly not have been completed without him. His technical guidance, helpful critique and belief in me, have been an endless source of motivation and support throughout my studies. I also feel gratitude to Morris for arranging my financial support and for his honest patience and tolerance.

I also thank Dr. Emil Lupu for his precious guidance and critical observations. Our edifying discussions have helped me develop several research skills. Many thanks to my colleagues and academics at Imperial, Dr. Naranker Dulay, Dr. Dan Chalmers, Krish Krishnakumar and Arosha Bandara for providing a friendly and productive atmosphere. As for all the other members of Imperial College whom I may have forgotten to mention, thank you.

I also thank my friends in London for making me feel at home: my best friend George Stephanopoulos, George Mournos, Yiannis Georgiadis, Nikos Damianou and Tyrone Grandison.

Finally, I thank my parents Andreas and Georgia, to whom this thesis is dedicated, for their endless support during my stay in London. They gave me courage and the strength to never give up. I also thank my sister, Anna, and Alexandra for her encouragement and support.

The work carried out in this thesis was partly funded by:

- The UK Engineering and Physical Sciences Research Council (EPSRC) as part of the PONDS and the PolyNet (GR/R31409/01) research projects.

- Cisco Systems as part of the Language Based Policy Specification, Analysis and Deployment for Large-scale Systems (Polyander) Project (http://www-dse.doc.ic.ac.uk/Projects/polyander/)

Contents

List of Figures

List of Tables

List of Abbreviations

AC	Access Controller	**PDB**	Per Domain Behaviour
AF	Assured Forwarding	**PDP**	Policy Decision Point
CIM	Common Information Model	**PHB**	Per Hop Behaviour
CIMOM	CIM Object Manager	**PIB**	Policy Information Base
COPS	Common Open Policy Service	**PEP**	Policy Enforcement Point
CORBA	Common Object Request Broker Architecture	**QoS**	Quality of Service
DiffServ	Differentiated Services	**RMON**	Remote Monitoring
DMTF	Distributed Management Task Force	**RSVP**	Resource Reservation Protocol
DSCP	Differentiated Services Code Point	**SLA**	Service Level Agreement
EF	Expedited Forwarding	**SLO**	Service Level Objective
GUI	Graphical User Interface	**SLS**	Service Level Specification
IETF	Internet Engineering Task Force	**SNMP**	Simple Network Management Protocol
JNDI	Java Management Application Programming Interface	**TCB**	Traffic Control Block
LDAP	Lightweight Directory Access Protocol	**TCP**	Transport Control Protocol
MIB	Management Information Base	**ToS**	Type of Service
OCL	Object Constraint Language	**UDP**	User Datagram Protocol
PBNM	Policy Based Network Management		
PCIM	Policy Core Information Model		

Publications

Leonidas Lymberopoulos, Emil Lupu and Morris Sloman. An Adaptive Policy Based Management Framework for Differentiated Services Networks. In Proceedings of Policy 2002: IEEE 3rd International Workshop on Policies for Distributed Systems and Networks, Monterey, CA, USA, 5-7 Jun. 2002

Leonidas Lymberopoulos, Emil Lupu and Morris Sloman. An Adaptive Policy Based Management Framework for Network Services Management. In Special Issue on Policy Based Management of Networks and Services, Journal of Networks and Systems Management, Vol. 11, No. 3, Sep. 2003

Leonidas Lymberopoulos, Emil Lupu and Morris Sloman. Using CIM to Realize Policy Validation within the Ponder Framework. Presented at DMTF's 2003 Global Management Conference, San Jose, CA, USA, 16-19 Jun. 2003 (Winner of the Academic Alliance Paper Competition)

Leonidas Lymberopoulos, Emil Lupu and Morris Sloman. Ponder Policy Implementation and Validation in a CIM and Differentiated Services Framework. In Proceedings of the 9th IEEE/IFIP Network Operations and Management Symposium (NOMS 2004), Seoul, Korea, 19-23 Apr. 2004

Chapter 1
Introduction

Policy based network management has recently emerged as a promising solution for the management of networks and distributed systems. In this new management approach, policies are defined as rules governing the choices in behaviour of a system. Policy based management addresses the requirement for providing dynamic and flexible management in order to deal with the increasing size and the complexity of the systems being managed. Support for distribution, automation and dynamic adaptation of the behaviour of the managed system is achieved by using policies. As stated in [Damianou 2002], the main benefits from using policy are improved scalability and flexibility for the management system. *Scalability* is improved by uniformly applying the same policy to large sets of devices and objects, while *flexibility* is achieved by separating the policy from the implementation of the managed system. Policy can be changed dynamically, thus changing the behaviour and strategy of a system, without modifying its implementation or interrupting its operation. Policy-based management is widely adopted by organisations such as the Internet Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF), as well as many network equipment vendors.

However, although the Internet community has shown considerable interest in policy-based techniques, and policy-based management has been the subject of considerable research, proposed solutions are often restricted to condition-action rules where conditions determine when actions should be performed on the managed entities. This results in static policy configurations where manual intervention is required to cater for configuration changes and to enable policy deployment. Whilst current research focuses mostly on rules for low-level device configuration, significant challenges remain to be addressed in order to: a) provide dynamic adaptation of policy in response to changes within the managed environment, and b) ensure, prior to deployment, that policies will lead to a feasible implementation for the specific environment where they apply.

In this thesis, we propose a novel policy based management framework that provides solutions to these research issues. This chapter will discuss the motivation behind the ideas presented in this thesis, will identify the requirements for a policy management framework, and will conclude by highlighting our contribution and presenting an outline of the structure of the thesis.

1.1. Motivation

The benefits of the policy-based management approach – improved scalability and flexibility within the management system – have attracted the Internet community to focus on the implementation of policy-based techniques for Quality of Service (QoS), caching, persistence and security management to support modern multimedia applications, mobility and ubiquitous computing. Service providers deploy policies within their networks and systems to satisfy customers requirements, defined within Service Level Agreements (SLAs).

Many current approaches to specifying Service Level Agreements, particularly for network services, concentrate on specifying QoS parameters such as delay, throughput, error rates and availability. The specification of the service is essentially static in that it often assumes a single type of service is provided at all times but many clients require services which vary according to date or time. In addition, 'fallback' classes of services should be provided under failure conditions when the primary class of service cannot be provided – service adaptation may take place either as a result of failures within the network or to accommodate changes in client application requirements. For example, a collaborative design application may switch from an audio phase to a phase needing video services, so the client application must be able to trigger changes to the underlying communication service.

Consider as an example a typical network of a large enterprise which consists of several local area networks (LANs) interconnected with a wide area network (WAN) through one or more access routers. The IT department of the enterprise is responsible for operating the network so as to satisfy the SLA established in the enterprise. Following the policy based management approach, the administrator will deploy network policy rules and the management system will automatically distribute the rules to the network devices. The enforcement of the policy rules will provide the network service with certain QoS guarantees to the applications using the service. For example, if the established SLA in the enterprise states that *"A video application between clients in Site A and a video server in Site B should receive Gold Service"*, where "Gold Service" is defined as the network service with the lowest delay and the lowest packet loss, and Differentiated Services architecture [Carlson et al. 1998] is deployed in the network, then the administrator should probably deploy a policy rule that instructs the network to forward the packets that belong to the video application according to the Expedited Per Hop Behaviour [Jacobson et al. 1999], which guarantees traffic with the lowest values of delay and packet loss.

Similar configuration policies can be deployed within networks implementing other QoS architectures, such as MPLS [Cucchiara et al. 2001] or Integrated Services [Braden et al. 1994], in order to guarantee the "Gold Service" to the video application.

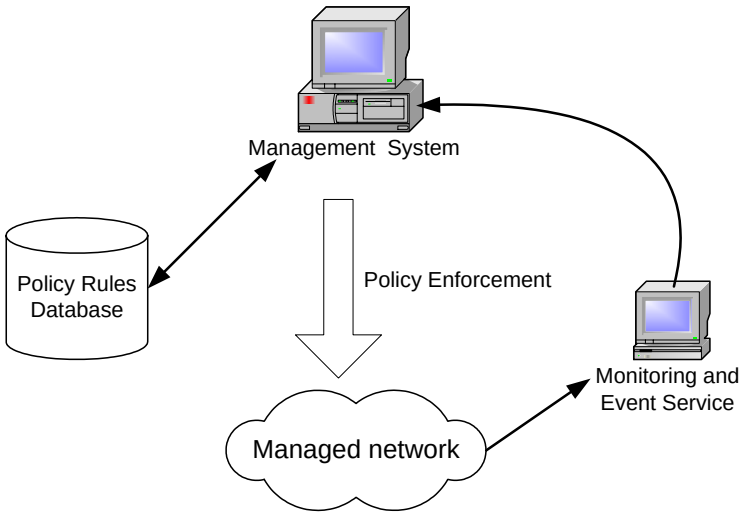


Figure 1.1 SLA management with a policy-based management system

However, in addition to performing the mapping of policy to network configuration, the management system should also be able to react to changes that require modification of the existing network configuration. Figure 1.1 outlines some typical cases where the management system should change the existing network configuration – these include:

- A new user or an application requests changes to the provided QoS. For example, clients in site A may request more network resources for a running session, in order to receive better video quality from the video server located at Site B. Adaptive applications, which tailor their behaviour according to the available network resources, can change their QoS requirements at run-time. This implies that network policy attributes must be changed at run-time to support the new user or application requirements.

- Performance measurements from a monitoring service may indicate performance degradation, thus requiring changes in the service network configuration or even the selection of a new service to cater for the client application. This, in turn, may require attribute changes in the deployed network policy rules or even the selection of a different network policy to cater for the application. For example, if a deployed network policy that handles the video application packets can no longer guarantee low packet loss due to high congestion, then a different network

policy rule which can guarantee low packet loss should be chosen for the video application.

- Events indicating network failures or time events may trigger changes. For example, a network policy deployed only along a specific path of routers in the managed network may not be suitable for the video application when the routing path inside the managed domain changes. In this case, a new network policy, which can be applied to the new path, must be automatically configured and distributed in order to handle the video application packets.

Similar requirements for adaptation of policies exist in different applications of policy based management systems, such as in network security management, where policy should be adapted to change firewall and router packet filtering rules upon certain events, eg. the detection of denial-of-service attacks. Adaptation within a ubiquitous computing environment is also required in several cases in order to adapt policy with respect to changes in the current context of the user, e.g. when a user enters a new location or changes his activity, etc.

Another problem that service providers experience with policy-based systems, is that deployment of policy in complex network environments can become error-prone. In a real networking scenario, multiple policies apply to network elements in order to support the requirements of different applications, different users and cooperating but distinct administrative domains. Since the shared resource that the network represents is itself composed of different elements with varying capabilities and interfaces, it is essential to ensure that the network elements have the capability to implement the policy. This will prevent the management overhead and the potential problems that may arise when trying to enforce policies that are not feasible in the given network environment.

The management system must therefore implement mechanisms to check that the devices to which the policies apply support the required functionality, i.e. the policy invokes an operation actually implemented by the device; the device has the required resources needed to satisfy the policy action or the policy satisfies any application specific constraints or restrictions imposed by the existing managed environment.

1.2. Requirements

The framework proposed in this thesis strives to deliver an integrated solution to the problems identified in the previous section. We have identified the following requirements for providing policy adaptation and validation of policy in large-scale and heterogeneous networking environments:

- The management system must have the flexibility and necessary abstractions to manage a variety of device types, with different capabilities and limitations, from different vendors. The system architecture should be sufficiently flexible to allow adding new device types with minimal updates and recoding of existing components. To cater for large-scale networks, the management system must be able to apply policy rules to sets of devices rather than individual ones. When adding a new device to a set, relevant policies should be automatically deployed and enforced on it.

- The management system must be able to adapt to changes in user requirements or to changes within the managed network environment. In addition to adapting the behaviour of managed devices, the management system should also adapt its own behaviour, if necessary. Consequently, the management system must implement mechanisms to modify network behaviour by dynamically changing policies relating to the configuration of managed devices and dynamically selecting which policy should be enforced within the network in order to modify the management strategy.

- The management system should ensure that the policy is consistent with the functional or resource constraints within the target environment. Static checking should be performed, where possible, prior to deployment, in order to detect invalid policies at design time. Furthermore, policy constraints that must be checked at execution time are required for policies related to resource allocation that depend on the current state of the system.

1.3. Contribution

This thesis introduces a novel policy based management framework to address the requirements identified above. Ponder [Damianou et al. 2001], a declarative, object-

oriented language for specifying security and management policies in networks and distributed systems, is used as the policy specification language. The ideas that this thesis introduces for event-based configuration of networking mechanisms, adaptation of policy in respect of changes within the managed environment and validation of policy in respect of the capabilities of the network elements implementing the policy, form the contribution of this thesis.

In particular, the framework presented in this thesis specifies network management policy as event-based Ponder obligation rules, which explicitly identify their policy targets and allow grouping of targets into domains to cater for large-scale networks. It provides the administrator the flexibility to specify policies at different levels of abstraction and implements a generic enforcement architecture for policy deployment, independent of the lower level management protocols.

This thesis also presents a novel approach for adaptive network management. Adaptive management is realised through dynamic adaptation of policy in response to changes within the managed networking environment. A flexible architecture is implemented where policy adaptation is specified and enforced by other policies, specified in the same Ponder policy notation. Policy adaptation includes both dynamically changing policy parameters and selecting policies from a set of predefined ones to be enforced within the network.

Furthermore, this thesis presents novel ideas that provide solutions for validating policies with respect to the capabilities of the target network environment implementing the policy. This enables our framework to support both static checking, prior to deployment, in order to detect invalid policies at design time, and dynamic checking of policies related to resource allocation that depend on the current state of the system.

Although this thesis focuses on solutions for a policy based framework for management of networking mechanisms, such as Quality of Service and security, the same ideas can also be applied to other areas of management, eg. storage, distributed file systems and pervasive systems management. Our ideas on policy enforcement, adaptation and validation are not restricted to specific management protocols, mechanisms or information models; they can be implemented using available protocols, mechanisms or information models, other than these described in this thesis.

1.4. Structure

This thesis is structured as follows. Chapter 2 will present the background basis of this thesis and an account of related work. The latter will cover the most important work on policy-based management for networks, an account of QoS adaptation techniques and proposed approaches for policy adaptation and policy validation.

Chapter 3 will present our approach for policy-based configuration of network mechanisms, using Differentiated Services as an example of a managed network mechanism. It will present how device-independent Ponder policies can be specified using the DMTF's Common Information Model (CIM) [DMTF 2004] modelling framework for representing network mechanisms, and will describe a generic architecture for mapping Ponder rules into device configuration. An example implementation of policy enforcement for Linux DiffServ routers will also be described.

Chapter 4 will present how policy adaptation can be realised within the Ponder policy framework. A generic policy adaptation enforcement architecture and a prototype implementation of this architecture, targeted for adapting policies that apply to a Differentiated Services network environment, will also be presented.

Chapter 5 will present how Ponder policy can be validated, using CIM as the modelling framework of the capabilities of the networking elements implementing the policy. A generic policy validation architecture, and as a proof of concept, an implementation of the proposed architecture for validation of network policy that applies to a Differentiated Services environment, will also be discussed.

Chapter 6 will present a critical evaluation of the overall work achieved and Chapter 7 will give conclusions and directions for future work.

Chapter 2
Background and Related Work

The framework we propose for the management of network services is built upon the Ponder Policy Based Management Framework, developed at Imperial College over the past 10 years. This chapter will provide an overview of the Ponder Policy Framework and an account of related work, which we have categorised in several areas. Figure 2.1 presents the related work areas that will be covered in this chapter. Note that much of the related work discussed in this chapter was carried out concurrently with the work presented in this thesis.

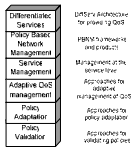


Figure 2.1 Related work coverage

2.1 Ponder Policy framework

Substantial research in the area of Policy Based Management has been carried out by Imperial College during the past years. The new ideas that the IC policy framework introduced – the definition of domains for the purpose of specifying policies for groups of objects, the concepts of authorisation and obligation policies for the management of distributed systems, etc. – have resulted in the development of the Ponder language for policy specification.

Obligation policies defined in Ponder [Damianou et al. 2001] specify the actions that managers must perform when certain events occur, and provide the ability to respond to

changing circumstances. Obligations are event-triggered condition-action rules, which explicitly identify the *subjects* (i.e., managers or configuration agents) that are responsible for performing the management actions on *target* objects. The syntax of obligation policies is shown in Figure 2.2.

```
inst oblig policyName {
on   event-specification ;
subject  [<type>] domain-Scope-Expression ;
[ target  [<type>] domain-Scope-Expression ;]
do   obligation-action-list ;
[ catch   exception-specification ; ]
[ when   constraint-Expression ; ] }
```

Figure 2.2 Obligation policy syntax

Both subject and target objects are specified in terms of *Ponder domains,* which are a means of grouping objects to which policies apply [Sloman et al. 1994]. For example, in order to logically group a number of Cisco routers, the administrator could define the Ponder domain */Devices/Routers/CiscoRouters*, following the syntax shown in Figure 2.2. Scalability, with respect to the number of policy rules that need to be specified and deployed for large sets of devices is achieved this way, as the same policy can be applied to many devices in the Ponder domain, rather than having to define a separate policy rule for each individual device.

Events can be internal, e.g. a timer event, or external events, which are collected and distributed by a monitoring service. Composite events can be specified using the event composition operators that the language supports.

Actions can be operations defined in the management interface of the target object or internal operations of the management agent. In the latter case, the target element of a policy is optional. Concurrency operators specify whether actions should be executed sequentially or in parallel and are used to separate actions in an obligation policy. Constraints follow the *when* clause in the policy specification and are used to limit the execution of the policy only when the constraint expression evaluates to *true*. The optional catch-clause specifies an exception that is executed if the execution of the policy actions fails for some reason. The above syntax is used for declaring a policy instance. The language provides reuse by supporting the definition of policy types, which can be instantiated for each specific environment. Figure 2.3 shows the syntax for declaring obligation policy types and instantiations.

```
type oblig policyType (formalParameters ) { { obligation-policy-parts } }
inst oblig policyName = policyType ( actualParameters ) ;
```

Figure 2.3 Obligation Types and Instantiations

Ponder also includes support for access control, which is implemented by access controllers running on the target objects whose behaviour is controlled by *authorisation policies*. These specify what actions *subjects* can perform on the *target* objects. Actions are operations defined in the management interface of the target object. A positive authorisation policy defines the actions that subjects are permitted to perform on target objects. A negative authorisation policy specifies the actions that subjects are forbidden to perform on target objects. The syntax of an authorisation policy is presented in Figure 2.4.

```
inst ( auth+ | auth– ) policyName {
subject [<type>] domain-Scope-Expression ;
target [<type>] domain-Scope-Expression ;
action action-list ;
[when constraint-Expression ; ] }
```

Figure 2.4 Authorisation Policy Syntax

Ponder also supports grouping of policies into roles related to positions in organisations or the set of policies applying to a particular network component [Sloman et al. 1999]. Management structures can be defined as configurations of roles with policies applying to relationships between roles for organisational units such as departments or buildings. Meta-policies specify constraints over a set of policies, with respect to the permitted types of policies or the elements within the policies. We have given a very brief overview of Ponder. More details on event composition, composite policies, roles, relationships and constraints can be found in [Damianou et al. 2001].

Apart from the Ponder language specification, the Policy Framework at Imperial College has introduced an object-oriented model for the deployment of Ponder policies [Dulay et al. 2001]. The model defines objects for policies, for domains, and for policy enforcement agents. Each policy type is compiled into a policy class by the Ponder compiler and is represented by a policy object at runtime, after the instantiation of the class. Policy objects entrust the enforcement of policies to one or more enforcement agents: for authorisation policies to each target's access controller (AC), and for obligation policies to each subject's policy management agent (PMA).

Access controllers enforce all the authorisation policies for one or more target objects. When an action is "intercepted" by an access controller, it checks whether the access should be permitted. The access controller will check, for example, that the subject of the action is in the subject set of the policy, that the target of the action is in the target set of the policy and that the action is a valid action for the target.

Policy Management Agents interpret and enforce the obligation policies that are relevant to them. Each PMA registers with an event service to receive the relevant events which will trigger the policies it holds. Policy actions are executed on the target domain objects when the triggering event is received by the PMA. Events may pass parameters to the PMA and the policy actions may also have parameters.

A set of management tools [Damianou et al. 2002] have been implemented following the ideas presented above. This set of tools includes a policy editor, a domain browser, and a management console. The policy administrator uses the policy editor to edit and compile policies. The policy editor tool is integrated with the policy compiler. Code generators added to the compiler framework, are accessible and can be enabled from within the editor to select the type of code to be generated. Policies in the current implementation are compiled into Java objects and are stored in the domain service. The latter uses LDAP [Wahl et al. 1997] as its access protocol and stores policy objects, subject/target objects and policy enforcement objects. The domain browser is a graphical tool that displays the objects stored in the domain service and provides a common user interface for all management interactions with objects stored in the domain service. Finally, the management console tool is used to distribute policies to their enforcement agents and manage their lifecycle. A policy object can be loaded into its enforcement components, and once loaded, it can be enabled, disabled or unloaded from its enforcement components. Unloaded (i.e. dormant) policies can be either re-loaded or deleted. More information about the use of the Ponder tools can be found in [Damianou et al. 2002].

Apart from the Ponder language specification and the tools for deploying Ponder policies, work has been carried out at Imperial College on conflict detection and resolution in [Lupu et al. 1999]. Recent work [Bandara et al. 2003] investigates techniques for rule refinement, conflict and consistency analysis of policies adopting a formal approach towards analysis and refinement by translating Ponder policies into event calculus. A number of new research projects at Imperial College [Aedus 2004], [Amuse 2004] are investigating if and how the Ponder framework can be used for delivering management solutions for ubiquitous environments.

2.2 Differentiated Services

Network applications commonly compete for bandwidth, leading to congestion and variable performance in terms of throughput, latency variations (jitter), propagation delay, etc. Traditional IP networks provide a "best-effort" service which treats all applications equally when competing for network resources, irrespective of how critical they are. To address this issue, research and industrial efforts have focused on the development of Quality of Service (QoS) enabled networks which provide mechanisms to allow network applications to request and receive predictable performance levels.

Two approaches have been proposed for providing QoS in IP networks. *Integrated Services* (IntServ) [Braden et al. 1994] uses the Resource ReSerVation Protocol (RSVP) [Braden et al. 1997] to provide per-flow QoS support by dynamically reserving resources on RSVP-enabled routers. Each flow is identified by the destination IP address, the transport protocol and the port number used by the application. As discussed in [A. Mankin et al 1997], this approach has significant scalability problems as routers must maintain a lot of information about the application flows and their reservations as well as processing a large number of messages for each reserved flow.

Differentiated Services (DiffServ) is a much simpler alternative to IntServ/RSVP. The QoS information is encoded within the DiffServ Code Point (DSCP), which is the most significant 6 bits of the Type of Service (ToS) byte contained in the IP header. Only edge routers in the DiffServ architecture need to perform the classification of traffic flows into classes of service. The core routers queue and schedule packets according to the value of the ToS field.

Throughout this thesis, we will use Differentiated Services as the mechanism for providing network services due to the fact that it scales well in large networks and is therefore becoming more popular with network providers. However, our proposed policy-based framework is designed at a fairly high-level of abstraction; it uses the abstractions of the underlying QoS mechanisms. This enables us to use our framework for configuring other types of QoS enabled networks, such as Multiprotocol Label Switching (MPLS) [E. Rosen et al. 2001] networks.

2.2.1 Differentiated Services Architecture

Figure 2.5 shows an example DiffServ network.

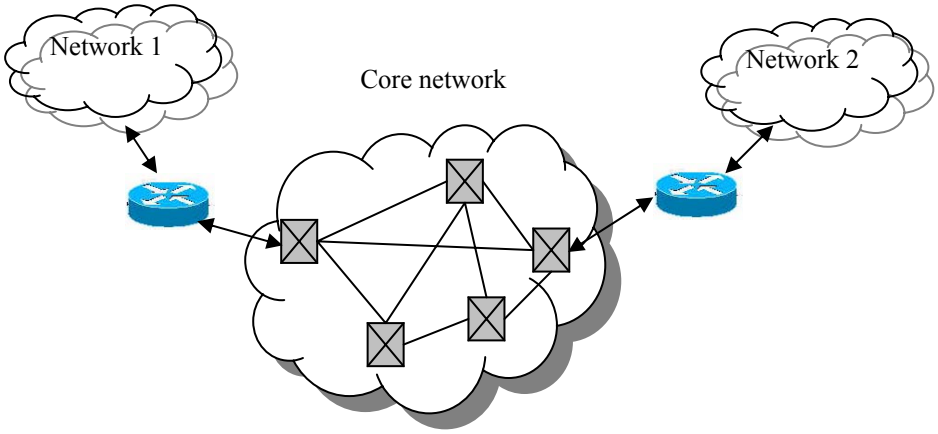


Figure 2.5 Differentiated Services Architecture

According to the DiffServ Architecture [Carlson et al. 1998], at the boundaries of the DiffServ domain, edge (or access) routers classify and possibly condition the incoming traffic. Each incoming flow is assigned to a specific behaviour aggregate, identified by the 6 bit DSCP within the IP header, allowing 64 different classes of service to be defined in a DiffServ network.

Within the core network, routers examine only the DSCP field of the incoming packet and forward it according to the Per-Hop Behaviour (PHB) associated with that DSCP. PHB defines how a packet should be processed within the router, before being transmitted to the next hop. The IETF DiffServ working group has defined a number of standard PHB's – Class-Selector, Assured Forwarding, Expedited Forwarding, and Default [Nichols et al. 1998; Heinanen et al. 1999; Jacobson et al. 1999].

In order to implement a PHB, a DiffServ-enabled node has several components for which a model is presented in [Bernet et al. 2002]. The proposed model (DSMODEL) represents the mechanisms defined in the DiffServ Architecture as components that form building blocks. The model includes abstract definitions for Traffic Classification Elements, Metering Functions, Actions of Marking, Absolute Dropping, Counting and Multiplexing, and Queuing elements. The latter include capabilities for algorithmic dropping and scheduling. Certain combinations of the above functional datapath elements form higher-

level blocks known as Traffic Conditioning Blocks (TCBs). Figure 2.6 shows a TCB comprising of six components of the DiffServ architecture.

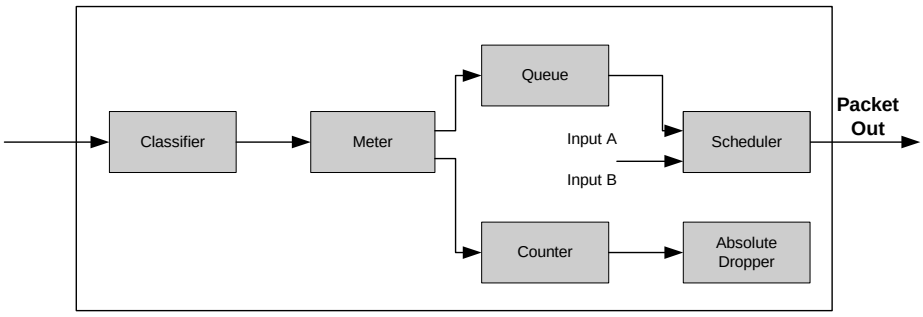


Figure 2.6 Example of a Traffic Conditioning Block

According to DSMODEL, a *Classifier* element takes a single traffic stream as input and generates N logically separate traffic streams as output. Packets from the input stream are sorted into various output streams by filters, which match the contents of the packet or possibly match other attributes associated with the packet. *Meters* are logically 1:N (fan-out) devices. Meters are parameterized by a temporal profile and by conformance levels, each of which is associated with a meter's output. Each output can be connected to another functional element. A *Counter* element updates a packet count and also an octet count for every packet that passes through it. An *Absolute Dropper* simply discards all packets arriving at its input. A *Queue* element is a First In First Out (FIFO) data structure, which at any time may contain zero or more packets. Furthermore, it may have one or more thresholds associated with it. It must support an enqueue operation to add a packet to the tail of the queue and a dequeue operation to remove a packet from the head of the queue. Packets must be dequeued in the order in which they were enqueued. A *Scheduler* is an element which gates the departure of each packet that arrives at one of its inputs, based on a service discipline, i.e. a scheduling algorithm. It has one or more inputs and exactly one output. Each input has an upstream element to which it is connected, and a set of parameters that affects the scheduling of packets received at that input. More information about the functionality of the DiffServ elements can be found in [Bernet et al. 2002].

We mentioned earlier that the functional components of a DiffServ device form higher-level blocks, the TCBs. According to [Bernet et al. 2002], a DiffServ-enabled device consists of a set of TCBs, a routing component, and a Configuration and Management Module. Their interconnection is presented in Figure 2.7.

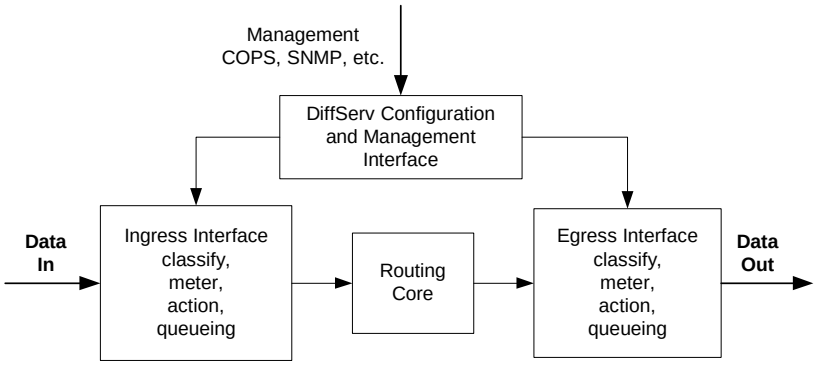


Figure 2.7 Conceptual model of a DiffServ-enabled device

The management of a Diffserv device, i.e. the management of the individual functional elements and TCBs it contains, is realized through the Configuration and Management Interface via one or more management protocols. The following section will present the different protocols that are available within the Differentiated Services framework for managing DiffServ-enabled devices.

2.2.2 Management Protocols for Differentiated Services

Simple Network Management Protocol (SNMP)

The Differentiated Services Management Information Base (MIB) [Baker et al. 2001] is designed according to the abstractions provided by [Bernet et al. 2002]. This MIB models the individual functional elements that constitute the Traffic Conditioning Blocks (TCBs). These elements are stored as table entries and are interconnected through the "Next" attribute that each entry contains. The description of the tables that are defined in the DiffServ MIB is given below.

Table 2.1 DiffServ MIB tables

Data Path Table	**Defines the starting points of DiffServ data paths**
Classifier and Filter Tables	**Enumerate all the classifier and filter elements**
Meter Tables	**Enclose entries representing DiffServ meter elements**
Action Tables	**Action tables are used to represent elements that perform Marking, Counting and Absolute Dropping actions**
Queue, Scheduler and Algorithmic Dropper Tables	**Contain elements that correspond to queuing, scheduling and algorithmic dropping mechanisms**

The configuration of the DiffServ MIB table entries is realized through SNMP set-request messages sent by the SNMP manager to the agent holding the MIB. For Policy Based configuration, the SNMP manager must be a component of the management system. What is needed is the translation of policy rules to the corresponding SNMP messages.

Common Open Policy Service (COPS) protocol with Provisioning extensions (COPS-PR)

The COPS [D. Durham et al. 2000] protocol has been defined by the IETF Resource Allocation Protocol (RAP) working group as a scalable protocol that allows policy servers to communicate policy decisions to network devices. The standard COPS protocol was for use within the context of admission control with signalling protocols such as RSVP. However, the IETF RAP working group proposed modifications to the protocol (COPS-PR) that allow a policy server to send provisioning information to the policy enforcement device. Provisioning policies are generic and can be used for both QoS and Security management, i.e. IPSec.

The data carried by COPS-PR [Chan et al. 2001] is a set of policy data. The protocol assumes a named data structure, known as Policy Information Base (PIB), to identify the type of policy provisioning information that is communicated from the Policy Decision Point (PDP) to the Policy Enforcement Point (PEP). The PIB can be described as a conceptual tree namespace where the branches of the tree represent structures of data or Provisioning Classes (RPC's), while the leaves represent various instantiations of Provisioning Instances (PRI's).

The IETF Differv working group has defined a PIB for Differentiated Services [Fine et al. 2001]. Like the SNMP DiffServ MIB, the DiffServ PIB models the individual elements that constitute the TCBs. In addition, it includes classes describing the capabilities and limitations of the device. This information is reported by the PEP to the PDP, so that the PDP can apply the correct configuration to the PEP. The provisioning classes for DiffServ are represented as tables. Here is a description of PIB tables defined in [Fine et al. 2001].

Table 2.2 DiffServ PIB tables

Data Path Table	**Defines the starting points of DiffServ data paths within a single device**
Classifier Tables	**Specify a group of filters used for traffic classification**
Meter Tables	**Enclose entries representing DiffServ meter elements**
Action Tables	**Action tables are used to represent elements that perform Marking, Counting and Absolute Dropping actions**
Queue, Scheduler and Algorithmic Dropper Tables	**Contain elements that correspond to queuing, scheduling and algorithmic dropping mechanisms**
Capabilities Tables	**Define the capabilities and limitations of the elements listed above**

The semantic information and the relationship among different PIB tables are very similar to these in the MIB model. The difference is the syntax used to represent the various entries and the protocol used to download the tables to the devices. In the case of COPS, all the PIB tables are downloaded to the device in a single operation over a TCP connection. In the case of SNMP, the MIB tables are written one element at a time over UDP.

2.2.3 Differentiated Services support on Linux

Support for DiffServ on Linux is part of the more general Traffic Control Architecture [Almesberger 1999]. This architecture is comprised of the following major conceptual components: *Queuing disciplines*, *Classes* (within queuing disciplines) and *Filters*.

A *Queuing discipline* is an algorithm that manages the queue (ingress or egress) of a device. A simple queuing discipline may consist of a single queue which operates in a First In First Out mode, i.e. all packets are stored in the order in which they have been enqueued, and the queue is emptied as fast as the respective device can send.

Queuing disciplines are categorized as *Classless* or *Classful*. Classless Queuing disciplines do not contain internal subdivisions. Supported Classless Queuing disciplines are the following: *pfifo_last*, which operates in a FIFO mode, *Token Bucket Filter* (TBF) that transmits only packets arriving at a rate which is not exceeding some administratively set rate, but with the possibility of allowing short bursts in excess of this rate and *Stochastic Fair Queing* (SFQ), which is a simple implementation of the fair queueing algorithms family.

Classful Queuing disciplines may contain one or more *Class* components. Classes represent different kinds of traffic which should receive differing treatment by the algorithm that the queuing discipline implements. Classes may contain further Queuing disciplines. An example is the *Priority* (PRIO) Queuing discipline, which contains Classes that are assigned different priority. Packets enqueued within Classes with higher priority are always dequeued before packets within lower priority Classes. Other supported Classful Queuing disciplines are: *Class Based Queuing* (CBQ) and *Hierarchical Token Bucket* (HTB). For information about the functionality of the Queuing disciplines supported by the Linux kernel refer to [Linux Advanced Routing & Traffic Control HOWTO 2002]. In addition to the above traffic control Queuing disciplines, a special Queuing discipline has been implemented in the Linux kernel: *DSMARK*. This queueing discipline allows changing (marking) the DSCP of the packets it receives.

Filters are consulted to determine which class packets need to be sent when entering a Queuing discipline. For this purpose, filters match a packet's attributes such as the DSCP, source/destination address, source/destination port, ip protocol etc. and then decide which class the packet should be forwarded to. Filters can also be used to police incoming traffic. If the incoming traffic rate exceeds a configured rate, the filter can be configured to drop the traffic in excess, reclassify it or check if another filter will match it.

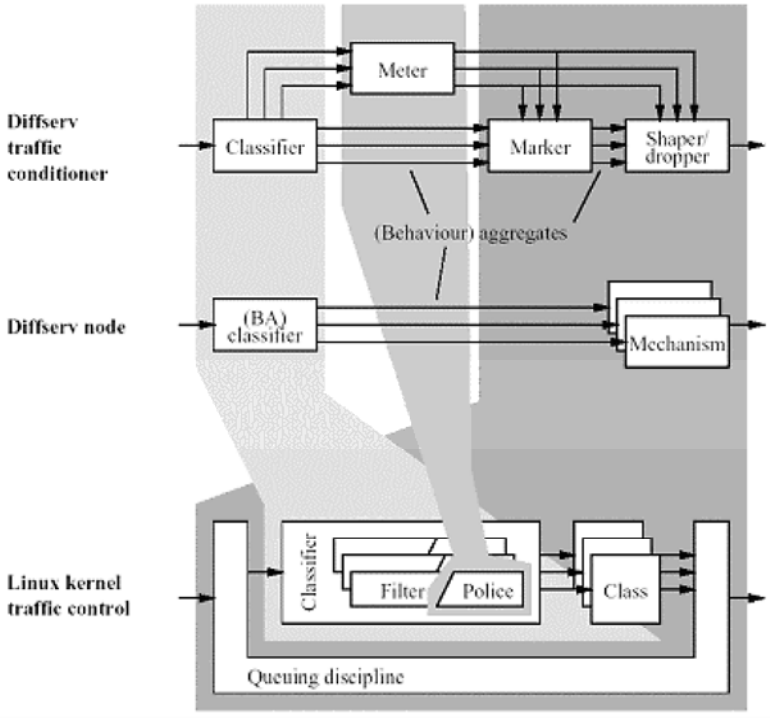


Figure 2.8 Relationship between DiffServ and Linux Traffic Control Architecture (from [Almesberger 1999])

Figure 2.8 from [Almesberger 1999] presents how Linux traffic control components relate to the Differentiated Services [Carlson et al. 1998] architectural components.

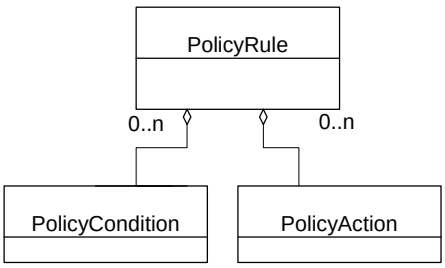
2.3 Policy Based Network Management

This section will provide an account of policy based management frameworks that strive to deliver solutions for management of networks implementing QoS mechanisms.

2.3.1 IETF Policy Workgroup

One of the most important ongoing research activities on policy specification is carried out by the IETF Policy working group [IETF Policy Working Group]. They do not use a language for specifying policies, but they represent policy with an object-oriented information model [Moore et al. 2001]. This model is an extension of the Common Information Model (CIM) that is developed by the Distributed Management Task Force (DMTF). A policy rule is modelled as an aggregation of policy conditions and policy actions. According to that representation, a policy rule expresses the statement: *if (set of conditions) then execute a set of actions.*

Policy rules, conditions and actions are represented as object classes and their associations are modelled with association object classes. For example, a *PolicyCondition* instance is linked to a *PolicyRule* instance with the association *PolicyConditionInPolicyRule* and a *PolicyAction* instance is linked to the same rule with the association *PolicyActionInPolicyRule*.



Network QoS policies within the IETF Policy framework are represented according to the information model defined in [Snir et al. 2001]. This model extends the PCIM model with QoS related policy actions, values and variables in order to add QoS specific semantics to the framework defined by [Moore et al. 2001]. Policy actions defined in QPIM are management actions for Differentiated Services (DiffServ) and Integrated Services (IntServ) networks.

An example of a policy rule defined in QPIM for establishing an EF Per Hop behaviour (EF-PHB) on a DiffServ node is the following:

If (traffic belongs to EF aggregate) then do EF-actions

The condition is used to identify and classify the traffic that enters the node. The action for providing a PHB is an instance of the class QosPolicyBandwidthAction. This class is used to control the bandwidth, the delay, and the forwarding behaviour for the flow where this action applies. For the above rule, this action instance can be described as following:

```
QosPolicyBandwidthAction EF:
p ForwardingPriority: 1
p BandwidthUnits: %
pMaxBandwidth: 50%
```

This action will configure the scheduler of the node to provide to the EF traffic aggregate maximum bandwidth of 50% of the total link bandwidth.

Apart from providing an information model for representing policies, the IETF framework has defined a schema for storing policies in a directory which uses the LDAP as its access protocol. This has been done for the classes defined in PCIM [Strassner et al. 2001]. Storing policies in a central directory is a key component of the Policy Based Management framework. This follows the concepts of Directory Enabled Networking [Strassner] which has been accepted as a powerful technology for the management of large networks.

The architecture the IETF has proposed for the enforcement of policies is presented in Figure 2.9.

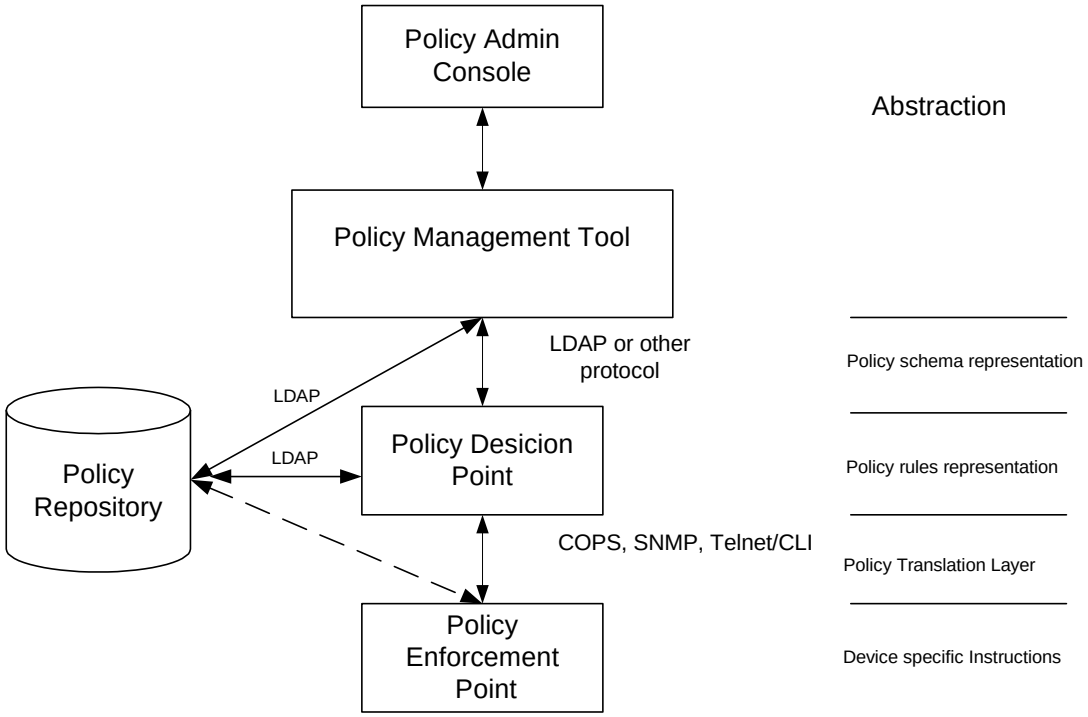


Figure 2.9 IETF policy enforcement architecture

In this architecture, each Policy Decision Point (PDP) is responsible for managing one or more Policy Enforcement Points (PEPs). The PDP is responsible for translating a policy into a form that the device can understand. The PEP tells the PDP what actions is capable of performing and how it wants its policy to be specified (for example, the particular form of conditions and actions that are transmitted to it). This may be communicated by several means, including dedicated policy protocols like COPS [D. Durham et al. 2000].

In the IETF architecture, directories are used for storing policies but not for grouping subjects and targets. They do not have the concepts of subject and target that can be used to determine to which components a policy applies, so the mapping of policies to components has to be done by other means (i.e., interface roles). Furthermore, they do not support policy rules that can be dynamically triggered by events to reconfigure the managed system according to changing circumstances. The policy work in the IETF seems to be focused only in the network layer and they have not considered the interaction between application and network policy.

2.3.2 Commercial Products

A number of vendors are marketing policy toolkits. The majority of these commercial tools are specific to quality of service management, but many also include access control

configuration. Here we will overview a number of major commercial products that are specific to QoS management. It should be noted that the information presented here is based on public-domain documentation, available at the time of writing, from the vendors' websites and industry surveys.

Cisco QoS Policy Manager (v3.0) [Cisco Systems Inc.]

QPM supports a broad range of Cisco devices, including routers and switches. Following the IETF policy representation, a QoS policy rule consists of a set of conditions and a set of actions. Policy actions (actions for classification, limiting, shaping and queuing traffic) are applied on a traffic flow if the flow matches the filters (conditions) defined in the policy. Filters define traffic characteristics.

QPM provides a web-based interface to define QoS policies and translates the policies into device-specific Command Line Interface (CLI) commands. Since policies do not specify their target elements, the administrator, prior to deployment, manually assigns through the management console a set of devices to each implemented policy rule. Policies are stored in the manager's QoS database, which is vendor's specific; policies are not stored according to a directory schema that follows a standard information model.

HP PolicyXpert [HP Policy Expert]

HP PolicyXpert defines policy as a combination of one or more sets of rules. Policy rules consist of a single action and one or more condition lists. These are constructed from one or more conditions, which match against time/date or packet/traffic characteristics. Policy actions are used to manage DiffServ and RSVP mechanisms. The product offers support for management of devices from a number of vendors. It also offers an Agent Software Development Kit (SDK). This SDK enables vendors to develop support for specific Quality of Service mechanisms on their devices.

Allot Communications NetPolicy [Allot Communications]

This product also follows the IETF ideas. A policy rule consists of conditions and actions. Conditions are used for matching IP addresses, protocols, application data, type of service (ToS) settings and time of the day. The administrator can group devices together in domains and manually enforce a set of policy rules to an existing domain. The COPS protocol is used only when NetPolicy uses the NetEnforcer device as enforcement point. The communication with other devices is realized through the Command Line Interface

(CLI) or SNMP. Directories are used not for storing policies, but for retrieving users and applications information.

In addition to the tools considered here, there are products from a number of companies that provide similar features. Lucent's RealNet Rules, Nortel's Optivity, Extreme Networks's ExtremeWare, Gold Wire Technology's Formulator and Dorado Software's Redcell Suite are some examples of these. Based on our investigation of the different tools available, we can summarise their features as follows.

Most of the tools specify the policies in the form of *if* <*condition*> *then* <*action*> rules. Target elements are assigned to policies either manually through the administrator console or using a role-based model. The different products allow the specification of varying degrees of conditions in policy rules including a number of time attributes, source or destination IP addresses, IP ToS, TCP and UDP port numbers, as well as higher-level user-defined data, and allow the user to permit or deny traffic based on those conditions. None of the presented toolkits supports a policy specification language and none of the tools appears to have considered the automation of the policy lifecycle and how to adapt the configuration of the target network elements when conditions change within the managed network. New configurations need to be imposed manually by the administrator through the management console.

2.3.3 Other network management policy approaches

In [Verma et al. 2001], a policy-based management system is proposed for managing Service Level Agreements within DiffServ networks. These SLAs are specified in terms of the application response time that occurs when a specific client accesses an application. The user-defined high-level policies will map each usage scenario, a client accessing an application on a server, to a service class. The service class has specific performance objectives associated with it.

A set of six tables is used in this framework, described in detail in [Verma 2001], to represent these policies, as shown in Figure 2.10. A table of users provides the mapping of users to the different subnets and IP addresses. A table of applications provides information about the port numbers that applications will use. A table of routers and a table of servers provide information about the different policy enforcement points that exist in the network, and whose configuration needs to be generated. The fifth table provides information about the different service levels that are defined within the network. The entries in the tables of users, applications, servers, routers and service

classes are tied together with entries in the table of policies, which maps the different application flows to different classes of service.

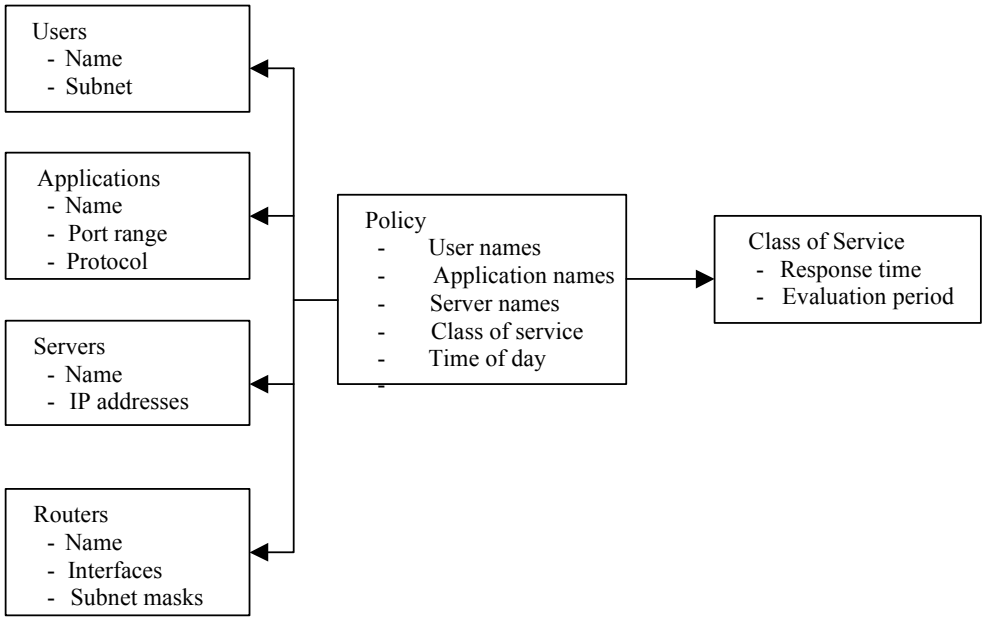


Figure 2.10 Set of tables for representing Enterprise SLAs

A translation process within the proposed management system is used to derive low-level policies from the high-level policies described above. In order to perform this high-level to low-level translation, they propose a representation of the low-level policy that would be required at each policy enforcement point (or policy target) within the network. For DiffServ, they use the low-level policy presented in Figure 2.11. For each device within the network, two tables are defined – one defining a set of classification policies and the other defining the different network levels supported at the device. A classification policy contains the five tuples that describe an IP packet flow (source and destination address, source and destination port and protocol) and a mapping to the network level. The network-level definition contains DiffServ-specific details, such as the appropriate marking within the IP header, and the rates that are appropriate for each of the network levels. The translation process maps the representation specified in terms of Figure 2.10 to the representation in terms of Figure 2.11.

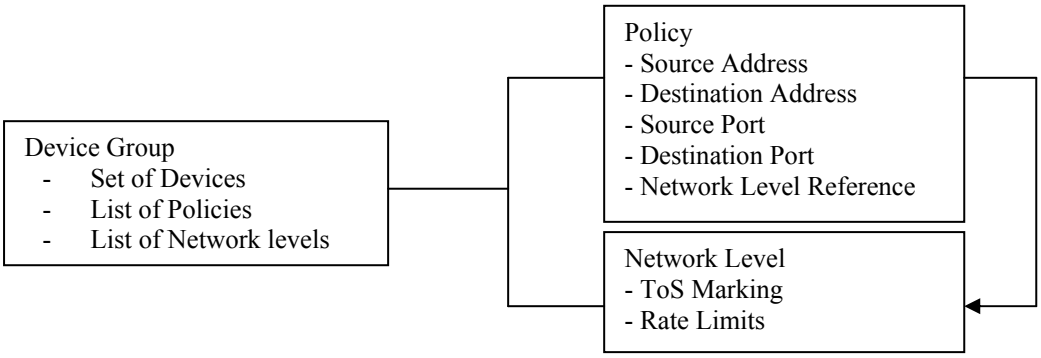


Figure 2.11 Low-level policy representation of DiffServ

The translation consists of four steps: name mapping, in which the high-level constructs like names of applications or users are mapped to header fields like IP addresses and port numbers; Class of Service (CoS) mapping, in which the definition of high-level classes of service (e.g. Precedence) is changed into technology-specific classes of service (e.g. EF-PHB for DiffServ); PHB's relevance determination, where the relevancy of a policy to a device or a set of devices is examined (i.e. to determine the set of devices that are affected by each of the defined policies); and grouping, wherein devices which have identical sets of applicable policies are grouped together in a common structure (this is done to determine automatically devices' roles). Note that in each of these steps, predefined mapping tables are used to derive the low-level representation of elements from the high-level representation. For example, for the CoS mapping process, this work assumes that an expert user has preconfigured the mapping from classes of service to the different network levels by defining a mapping table like Table 2.3.

Table 2.3 Mapping CoS to Network Level classes for DiffServ

Class of Service	Network Level	Rate Limit	Overflow
Precedence	Expedited Forwarding	10 Mbps	Best Effort
Preferred	Expedited Forwarding	12 Mbps	Best Effort
Default	Best Effort	50 Mbps	Drop

Rather than providing a policy-based management system for managing the characteristics of DiffServ devices, the proposed system only maps application flows into predefined and already implemented PHBs. Moreover, this system can only communicate

policies to the enforcement devices during the configuration process, initiated by the administrator. Configuration cannot be changed dynamically at run-time to reflect changes in the managed environment. In addition, the scope of this approach is specifically aimed at a management system for a DiffServ network, whereas our work is applicable to a wide range of management areas.

Another approach to network policy specification is the *path-based policy language* (PPL) described in [Stone et al. 2001]. This language is designed to support both the DiffServ and the IntServ network mechanisms and is based on the idea of providing better control over the network traffic by constraining the path the traffic must take. The rules of the language have the following format:

policyID <userID> @{paths} {target} {conditions} [{action_items}]
action_items =[{condition}:] {actions}

Action_items in a PPL rule correspond to the *if condition-then-action* of the IETF approach. In PPL, "*policyID created by <userID> dictates that a particular target class of traffic may use {paths} only if {conditions} is true after {action_items} are performed*" The following is an example of a PPL rule from [Stone et al. 2001]:

Policy 1 <net_manager> @ {<1,2,5>} {class={faculty}} {*} {priority :=1}

In this example, *Policy1* states that the path starting at node 1, traversing to node 2 and ending at node 5 will provide high priority to *faculty* users. The language has the same limitations with the IETF approach, as rules statically configure the managed elements. Moreover, PPL rules apply only to the network level, while our approach can apply to different levels of abstraction within the managed system.

The framework defined in [Martinez et al. 2002] combines IETF's Script MIB [Levi et al. 1999] and IETF's PDP/PEP architectures into a single architecture. Script MIB provides capabilities to transfer management scripts to distributed agents and to initiate and terminate the execution of the scripts. Since Script MIB can accept any form of management scripts, it provides support for arbitrary programming languages and multiple execution environments.

Script MIB is used to communicate policies to Script MIB agents, which implement PDP functionality. The authors propose two solutions for deployment of policies within the proposed architecture. The *first solution* defines policies as programs. These are

downloaded as scripts to the Script MIB agents and then executed by a common Script MIB runtime engine inside the agent. This runtime engine acts as a PDP and sends the corresponding low-level policy configuration to a local or remote PEP. A prototype implementation is provided for management of DiffServ Linux routers. Policies are implemented as Java programs; their actions involve creating and configuring Java objects that represent the DiffServ mechanisms of the routers according to the DiffServ MIB [Baker et al. 2001] data model. A driver has been implemented for translating the Java DiffServ objects to Linux traffic control [Linux Advanced Routing & Traffic Control HOWTO 2002] commands. The *second solution* represents policies according to IETF's PCIM information model. Policies in this case are not defined as programs, but as groups of PCIM objects. These policy objects are downloaded directly to the managed nodes. Within each managed node, a policy interpreter implements PDP functionality to translate the PCIM policy objects to their corresponding low-level configuration commands.

The system proposed in [Brunner et al. 2001] for the management of QoS in Multi-Protocol Label Switching (MPLS) networks, also follows the IETF Policy working group approach. They have extended the Common Information Model (CIM) policy model with MPLS specific classes. This system has the same limitations as the IETF framework. In [Bearden et al. 2001], IETF's PCIM is extended, to provide support for goal specification. Service-level goals can be specified to enforce QoS on a per-user, per-application basis. Monitored data is used to evaluate whether the specified goals are satisfied. These service-level goals can be expressed in our framework as higher-level obligation policy rules.

2.3.4 Policy based frameworks for other management areas

We have presented an account of the most important policy-driven frameworks for QoS management. As we have discussed, the policy based management approach is adopted in other management areas, such as in security and ubiquitous systems management. This section will provide a brief account of policy-driven frameworks used in the emerging area of ubiquitous systems management. A detailed survey on policy management approaches for security management can be found in [Damianou et al. 2002b].

A policy language (Rei) for a pervasive computing environment has been proposed in [L. Kagal et al. 2003]. Rei defines three types of constructs: policy objects, meta-policies and speech acts. Policy objects specify rights, prohibitions, obligations and dispensations (acting as deferred obligations). Rights are permissions that an entity has to perform an

action. Obligations are actions that an entity must perform and are triggered when a certain set of conditions evaluate to true. Prohibitions are negative authorisations, while dispensations define the actions that an entity is no longer required to perform, acting as waivers for existing obligations.

The language has been implemented in the Prolog language. Figure 2.12 presents how rights and obligations are specified in Rei. Note that the construct *has* is used in the Rei specification to associate policy objects with their corresponding entities.

Rights

An entity, abc, can perform the action actionABC if the following rule evaluates to true:

- has (abc, right(actionABC, Conditions) and abc satisfies Conditions

Obligations

An entity, abc, is obliged to perform the action actionABC if the following rule evaluates to true:

- has (abc, obligation(actionABC, Conditions) and abc satisfies conditions.

Figure 2.12 Rights and Obligations in Rei

Meta-policies defined in Rei are used to resolve policy conflicts. This is realised by setting the modality preference (negative over positive and vice versa) or stating the priority between policy rules. To provide decentralised control of policies, Rei introduces the speech acts constructs. Supported speech acts are: delegations, requests, cancellations and revocations. A delegation speech act allows an entity to give a right to another entity or a group of entities. A revoke speech act is used to remove a right and acts as a prohibition. An entity can use a request speech act to request for an action or a right on one hand, while it can use a cancel speech act to cancel any request it has sent on the other. For more information about the use of speech acts refer to [L. Kagal et al. 2003]

The concepts of rights and prohibitions are similar to those of positive and negative authorisations in Ponder. Obligations are also similar to Ponder obligation rules. However, Rei focuses mainly on security management of ubiquitous systems. Policy defined in Rei is static; the proposed framework does not cater for adaptation of policy in response to changes in the managed pervasive environment.

There are available additional languages for management of ubiquitous systems, eg. the language presented in [Bertino et al. 2003] for specifying message filtering and routing policies for the UCS-Router, a policy engine for message dispatching in a pervasive environment. Like Rei, the main limitation of these approaches is that the policy rules that the administrator defines remain static during their lifecycle. Policy adaptation mechanisms are not provided in order to adapt the management strategy in response to changes in the dynamic pervasive environment

2.4 Service Level Specification and Service Management

Policies are deployed within the network in order to guarantee the Service Level Agreements (SLAs) established within the organisation. Service Level Objectives (SLOs) constitute the technical part of an SLA and are defined formally with Service Level Specifications (SLSs).

Work on Service Level Specification for DiffServ is carried out by the Tequila Project [TEQUILA 2002]. In this work, a Service Level Specification is a protocol independent representation of a set of technical parameters and their associated semantics that describe the service a flow is to receive over the transport domain, between ingress and egress interfaces. A Service Level Objective is a protocol dependent instantiation of a SLS, i.e. it contains the SLS parameters and their values. [D. Goderis et al. 2001] lists and presents the semantics of a set of basic SLS parameters. Table 2.4 presents a brief description of the proposed SLS parameters.

Apart from the SLS specification and semantics, Tequila in [Trimitzios 2001] proposes an architectural model for providing QoS in DiffServ networks. This architecture is composed of three main parts: an SLS management module, a Traffic Engineering (TE) module and a Policy management module.

Table 2.4 Tequila's SLS parameters

Scope	Uniquely identifies the geographical / topological region over which the QoS of the IP service is to be enforced. Scope is expressed by a couple of ingress and egress interfaces, i.e. Scope=(ingress, egress).
Flow Description	Identifies the packet flow for which the QoS guarantees need to be enforced. Packet characteristics such as DSCP, source/destination information, application information, etc. can be used to identify the packet flow.
Traffic envelope and Traffic conformance (TC)	It is a combination of TC parameters and the TC algorithm. TC parameters describe the reference values that the traffic has to comply with. The TC algorithm is the mechanism used to unambiguously identify all the "in" and "out" of profile packets based on the conformance parameters. An example is the token bucket algorithm based on the token bucket parameters (b,r).
Excess treatment	Describes how excess traffic (out-of-profile) will be processed, e.g. drop, shape, remark.
Performance parameters	Describe the service guarantees for the identified flow over the geographical/topological extent given by the scope. Performance parameters are delay, jitter, loss and throughput.
Service schedule	Indicates the start and end time of the service.
Reliability	Indicates the maximum allowed mean downtime and the maximum allowed time to repair.

The SLS management module is responsible for all SLS-related activities and it is comprised of two main sub-blocks: SLS Subscription block and SLS Invocation block. *SLS subscription* is the process of customer registration. This concerns the SLA, containing prices, terms and conditions and the technical parameters of the SLS. The subscription could provide the required authentication information for Authentication, Authorization and Accounting (AAA) purposes, when an SLS is eventually invoked. *SLS Invocation* is the process of dealing with the flow dynamically. It performs admission at run-time as requested by the user and delegated the necessary rules to the traffic conditioning elements in the same architecture.

The TE module is responsible mainly for obtaining the information needed to compute the QoS configuration and for establishing and maintaining the QoS path that has been

selected to process the service request. It is composed of several sub-modules (Network Dimensioning, Routing, Dynamic Route Management and Dynamic Resource Management).

The Policy management module includes functional components such as the Policy Management Tool, the Policy Storage Service and the Policy Consumers. It is possible to define SLS related policies for the SLS management block, for the dynamic resource/route management block, etc. Policy is defined in the Policy Management Tool using a high-level language and is then translated to the appropriate object-oriented policy representation for each of the functional blocks the policy applies to. Policies are stored in the policy repository, i.e. the Policy Storing Service. Policy Consumers may need to have direct communication with a Monitoring Service, in order to obtain information either about traffic-based triggering events or about other types of events, generated by some specific functional block of the architecture.

An SLS to DiffServ configuration mapping framework is proposed in [Prieto et al. 2001]. The Tequila project's specification is used as the SLS. In their architecture, the management system consists of two parts. The first performs both the SLS to PDB mapping process and an admission control process. The mapping module uses an N-dimensional space (e.g. delay, packet loss, and throughput) to classify an input SLS into an available intra-domain service, which is offered by an implemented PDB within the DiffServ network. The second is the policy-based control part. This controls the SLS mapping and the admission control processes. Network policy is used as the device configuration mechanism. However, this work does not have any concrete proposals for the policy part of the framework. Furthermore, the SLS to PDB mapping process is only initiated by the user; no actions are undertaken by the management system to dynamically select a new PDB when network conditions change.

[Keller et al. 2002] proposes a contract-based architecture for application-level service management. Contracts are used for defining, deploying, monitoring and enforcing SLAs in a dynamic e-Business environment. A generic object-oriented model describes the various sections of a contract between a client and a service provider. Contracts are managed by a Contract Management System, whose main functional components are: a measurement component, a violation detection component and a management component. The measurement component is responsible for collecting data relevant to a service's QoS parameters. The violation detection component retrieves data from the measurement component and evaluates if the guarantees defined in the contract are met. In case of a QoS violation, a notification is sent to the management component. The latter, upon

reception of a violation notification, initiates corrective measures to remedy the causes of the violation. The advantage of our proposed framework for network-level service management is the flexibility to implement dynamically new management strategies within the service management system.

A Customer Service Management (CSM) architecture is proposed in [Sprenkels et al. 2000]. This allows delegation of the service management task from the service provider to the customer. A CSM module is the basic block of the proposed management system. Customers can adjust SLS parameters through a parameter setting function block within the CSM module. A SLS mapping function is implemented within the CSM module, to derive device configuration from SLS information. Our framework can provide this functionality, by allowing users to trigger the execution of management actions within the Service Management Agent.

2.5 QoS Adaptation

Although the scope of this thesis is limited to the policy aspects of an adaptive QoS management framework, we also present a brief overview of some existing frameworks and techniques for providing QoS adaptation. The reason is that in many cases, adaptation techniques and mechanisms must be used within a generic policy-based management framework.

To address the problem of QoS adaptation, a number of adaptation-based frameworks have been proposed by researchers in this area. Compared to reservation-based systems [Aurrecoechea et al. 1998] that use resource reservation (with mechanisms such as [Braden et al. 1997]) and admission control, adaptation-based systems seem to provide a more promising solution for providing QoS to network applications. The reason is that reservation-based systems may experience scalability problems and may not be suitable for networks where the available bandwidth to share among applications is variable (eg. in mobile networks). Furthermore, resource reservation must be available end-to-end, something that it is not practically achievable in the Internet.

A lot of work on QoS adaptation has also been carried out in the Distributed Systems area for developing adaptive middleware systems. Most of this work provides adaptation by hard coded adaptive mechanisms in middleware systems for supporting multimedia applications. Adaptive systems of this category, such as [Wang; G. Gordon et al. 1997; Haahr et al. 2000], do not take into account the changes in user preferences and the

application to drive dynamic adaptation. Rather, they work as “black boxes”, implementing a specific adaptation mechanism that can not be dynamically changed to provide new adaptation functionality.

The framework proposed in [Campbell 1996] presents a QoS Architecture transport system for a multicast, multimedia networking environment. In this work, the network provides an adaptability mechanism to cope with QoS fluctuations. The application passes information to the network about the QoS guarantees it should receive and expects that the adaptive mechanisms within the network will provide the requested QoS. It offers a QoS configurable API at the transport layer, which enables applications to have control over QoS. QoS is specified at the API in terms of a flow specification, which includes parameters such as delay, throughput, jitter etc. and a QoS policy. The QoS policy enables users to advise the infrastructure on how to deal with the flow when resource availability changes. A distributed QoS adapter interprets the policy and is responsible for informing applications when resources become available. A QoS adaptation protocol is implemented for the communication between QoS adapters

OpenORB [G. Blair et al. 2000] is a reflective middleware designed at Lancaster University. At load time, appropriate components are selected and composed as a middleware instance. Using reflection [E. Gamma et al. 1994], components can also be changed or loaded at run-time. Every object is associated with a “meta-space” that can be accessed through one of the “meta-model” interfaces: “encapsulation”, “composition” and “environment”. This system was prototyped using the python programming language. As described in [G. Blair et al. 2000], a mechanism for management components can be added dynamically to the component graph to both monitor and strategically adapt the middleware in a procedural, policy controlled manner. These event-based scripted and interpreted adaptation controllers can be dynamically changed to facilitate changing context and requirements. However, a high level view of how the system should adapt is lacking in this framework.

The framework proposed in [Bhatti et al. 1999] implements adaptation mechanisms at the application level. In this work, the adaptation mechanism takes into account information about the QoS achieved in the network, the application capabilities and the user preferences, in order to dynamically decide which flow-state is more suitable for the application. An Application Adaptation Function (AAF) receives information about the user preferences, the application flow-states and the network QoS by means of QoSReports, summarising the compatibility between the network QoS and the application’s flow-states. It then decides which flow-state should be selected for the

application in the given instance. The decision can be performed either automatically (with no external interaction with the user) or semi-automatically, where the user is given the ability to interact with the adaptation module.

As a conclusion, a main disadvantage of QoS adaptation systems is that the techniques they implement are usually static, lacking the flexibility to change when changes occur within the network or build upon previously gained experiences. A policy-based management system can provide the flexibility of dynamically changing, when required, the adaptation strategies implemented within the system without the need to recompile the components implementing the adaptation mechanisms and stop the system while this is being done.

2.6 Policy Adaptation

The framework proposed in [Yoshihara et al. 2001] adapts policy parameters as a result of monitoring the network. A management script includes policies, expressed in the IETF representation, and also specifies how the policy life cycle should be managed. The script notifies the management system about QoS threshold violations. In this work, a prototype implementation is provided for Differentiated Services, where policy parameters, such as the peak rate of a traffic profile, its peak burst size and the associated DSCPs, are changed dynamically to adapt to system behaviour. The framework we propose for the adaptive management of DiffServ can specify, in a uniform way, all the necessary information required for enforcement and adaptation of policies using obligation rules. Furthermore, in addition to providing adaptation by changing policy parameters, we can also select new policies to be enabled upon events other than just QoS violation events.

[Marshall et al. 2001a] presents an architecture for the management of a network offering active services. In their architecture, a bacterial algorithm forms the basis for the adaptation performed by autonomous controllers. These controllers are programmed (like a bacterium) to autonomously replicate policies that improve its performance and de-activate policies that degrade performance. This way, “useful” policies spread and “poor” policies die out. A policy is evaluated through a fitness (revenue-cost) function. In this work, each policy is related to one active service; policies control the deployment of services (proxylets) in their active services environment. [Marshall et al. 2001b] presents an example of this type of adaptation for providing QoS differentiation of active services, where the queue length of network servers (DPSs) is adapted to provide either short delay

or low loss to service(s), depending on the users QoS requirements. Example of these requirements (policies or service genes) can be:

"Accept request for service A if DPS < 80% busy" or

"Accept request for service C if queue length < 20".

In our framework, policies are used in a more generic sense, describing the actions that management agents must undertake when receiving different types of requests. We provide adaptation, in a more systematic way, by adapting the policy based management system itself, either by changing attributes of policies or by removing and adding new policies.

[Uttamchandani et al. 2003] proposes a framework for policy-based self-management of distributed file systems. Their policy rules extend IETF's "if–then" policy semantics by including in the rule definition the implications of the rule within the system and the system's workload characteristics on which the rule is dependent. "Behaviour implications" specify the impact of the rule on the system. This is expressed in terms of the observable characteristics of the system, such as throughput, latency, reliability etc. Workload characteristics capture the properties of the application accessing the distributed file system, such as the read/write ratio, whether the access is sequential or random, etc.

The framework uses the behaviour implication specified in the policy to decide which combination of rule(s) should be invoked when the managed system's desired behaviour has not been met by the current invocation of policies. This decision is taken by an automated decision-making component which uses monitoring information to analyse the current system state and determine which rules possess the necessary behaviour implication to help the system move to its desired state.

Since the decision making-process depends on the non-static behaviour implication of the policy rules, the authors propose a self-learning mechanism in order to approximate the dependency function that relates the behaviour implication of a rule and factors such as the workload characteristics and the system's behaviour value when the rule is about to be invoked. This means that each time a rule is actually invoked, the changes to the system are monitored and feedback information is recorded to refine the function that provides the behaviour implication of the rule. The refined behaviour implication function is then recorded in the rule repository.

[Lutfiyya et al. 2001] presents a policy-driven framework for QoS management of multimedia applications. They specify policy at the application layer using the Ponder

language, although they rely on violation of constraints to trigger policy rules instead of events. Their QoS policy only provides the *QoSHostManager* component with a notification message; the corrective actions which are enforced upon QoS violation are described in other types of rules. No formal specification is discussed for these rules, although they could be specified with Ponder's obligation policies as well. Furthermore, they use the term "adaptation" to refer only to the actions, which are taken when a QoS violation occurs. We support the type of adaptation provided in their framework, but our approach applies to other circumstances other than just QoS violations.

The work presented in [Ahmed et al. 2002] uses IETF's PDP/PEP policy architecture to dynamically control the way "out of profile traffic" is treated in a DiffServ environment. Using information about the bandwidth allocation within the core network, the PDP uses an algorithm to decide which policy rule ("accept", "drop", "remark") should be enforced on edge routers in order to handle the "out of profile" traffic entering the DiffServ domain.

A policy-based management architecture based on intelligent agents has been introduced in [Hamada et al. 2000]. Agents are used to represent active policies. The architecture uses an "hyper-knowledge" space, which is a loosely connected set of different agent groups which function as a pluggable or dynamically expandable part of the hyper-knowledge space. Each agent collects and stores network information is a distributed manner by walking through the nodes in the network. Active policies, which are agents themselves, can communicate with agents in the hyper-knowledge space to implement policies and retrieve information from other agents. The architecture takes advantage of intelligent agents' features such as the runtime negotiation of QoS requirements. However, an active policy itself has to be created by the administrator and once deployed in the network, it remains static through its life-cycle.

2.7 Policy Validation

Very little work has been done on the problem of policy validation, ie. to ensure that policy will lead to a feasible implementation for the target environment where it applies. At the time of the writing of this thesis, we are not aware of any policy based management framework integrating solutions for policy validation, apart from the policy framework for management of Service Level Agreements described in details in [Verma 2001]. In this policy framework, which uses the tabular representation of policy shown in Figure 2.10, two types of policy validation are provided.

The first type of policy validation refers to ensuring that the syntax of the policy specification is valid. An XML representation of the policy (the XML based representation captures the policies defined with the tables shown in Figure 2.10) is checked to see whether it conforms to its data type definition (DTD) specifications. Syntactic validation depends on the policy specification language; as such, this type of validation is provided by the policy compiler [Damianou et al. 2002] within in the Ponder management framework.

In the second type of policy validation in [Verma 2001], simple checks are provided to ensure that the values specified for each of the attributes within a policy rule is valid. For example, an IP address must conform to the requirement of being 4-byte address with a subnet prefix of less than 32 bits, whether a percentile attribute should take values between 0 and 100. In addition, since which values are legal for one attribute may depend on the value of another attribute, the policy validation module checks that all inter-attribute relations that are required in the policy specification are valid. For example, the class of service that is specified in a policy must be a name of a class defined in the table of service classes (see Figure 2.10).

Apart from these simple types of policy validation, the work presented in [Verma 2001] presents an "offline" method to determine if the QoS goals of policies can be achieved within a given network. Since policies deployed in a network reflect the SLAs that are in effect for this network, the objective of this method is to determine whether a given set of SLA objectives can be achieved within the network. To achieve this, the author models the network as a graph and considers the queuing behaviour at each node in the graph. The queuing behaviour at different routers is combined to determine the end-to-end delays and loss rates in the network. More specifically, the method to validate the QoS goals of a policy involves the following steps:

i) Determine a queuing model for each component in the network
ii) Determine the router path that packets will follow in the network
iii) Estimate the amount of traffic that will be used on all network flows
iv) Analyse the queuing network obtained to determine the end-to-end delays that will be experienced in the network

More information on how each step can be implemented is provided in [Verma 2001]. Note that this offline method for validating policies' QoS goals, relies on a number of assumptions. Thus, this method can only be used to provide an estimate as to whether policies' performance levels within the network will be satisfied. However, it is important

to be able to provide the administrator within an offline tool to test the QoS guarantees of his network policies, before deploying them in the real network. In this direction, the framework proposed in this thesis includes tool support for testing policies on simulated networks. This tool will be presented in Chapter 4.

Chapter 3

Policy Based Configuration of Differentiated Services Networks

In the previous chapter, we provided an overview of existing policy based management frameworks and commercial products for network management and discussed their limitations. Based on this discussion, we have identified the following requirements for a policy based network management system:

- A policy-based management system must have the flexibility and necessary abstractions to manage a variety of device types, with different capabilities and limitations, from different vendors. The system architecture should be sufficiently flexible to allow adding new device types with minimal updates and recoding of existing components.
- To cater for large-scale networks, the management system must be able to apply policy rules on sets of devices rather to each device individually. When adding a new device to a set, relevant policies should be automatically deployed and enforced on it.

We propose a flexible, expressive and extensible framework to cover the requirements identified above. This framework uses Ponder as the policy specification language and the Common Information Model (CIM) as the modelling framework for network mechanisms. This chapter will present how management policy is specified and enforced in our framework, using Differentiated Services networks as an example of a managed networking environment.

3.1 Policy Specification for Differentiated Services

A management system must have the flexibility and necessary abstractions to manage a variety of device types, with different capabilities and limitations, from different vendors. To meet this requirement, several information models have been proposed by standards organisations as a means to represent the abstractions of the managed entities.

For Differentiated Services, an informal model has been proposed by the IETF [Bernet et al. 2002] to represent the mechanisms that DiffServ-enabled devices implement to

provide different treatment to traffic aggregates. Based on this model, several information models have been proposed to meet the needs of different management protocols. In particular, work within the IEFT DiffServ framework has resulted in two internet standard information models:

- The Differentiated Services Management Information Base model [Baker et al. 2001] that is used by the Simple Network Management Protocol (SNMP)
- The Differentiated Services Policy Information Base model [Fine et al. 2001] that is used by the Common Open Policy Service (COPS) protocol with Provisioning extensions (COPS-PR)

An additional information model for DiffServ is proposed by the Distributed Management Task Force (DMTF). This model is part of the standard object-oriented Common Information Model (CIM) Schema [DMTF 2004] that DMTF has defined to represent the abstractions of a wide range of network and computing elements of a managed system and the relationships between them. The CIM Schema is the combination of the Core and several Common Models. The Core model captures notions that are applicable to all areas of management, while the Common Models are information models that capture notions that are common to particular management areas, but independent of any particular technology or implementation.

DiffServ functional elements are represented in CIM's Network Common Model, included in [DMTF 2004]. Their representation also follows the abstractions provided by [Bernet et al. 2002]. In particular, the CIM Network Commom Model defines the class *QoSService*, which is a subclass of the generic class *NetworkService*. The *QoSService* is an aggregation of instances of the *ConditioningService* class, whose subclasses define the router's DiffServ mechanisms. Figure 3.1 presents the main DiffServ elements defined in the CIM Network Common Model.

The top-level classes (*ClassifierService, MeterService, MarkerService, DropperService, DropperThresholdService, QueuingService* and *PacketSchedulingService*) represent the abstractions of DiffServ mechanisms of the managed device. Each top-level class may have subclasses that correspond to specific implementations of a particular mechanism. For example, the classes *AverageRateMeterService, TokenBucketMeterService* and *EWMAMeterService* (Exponentially Weighted Moving Average Meter Service) derive from the abstract class *MeterService* to represent specific DiffServ metering mechanisms.

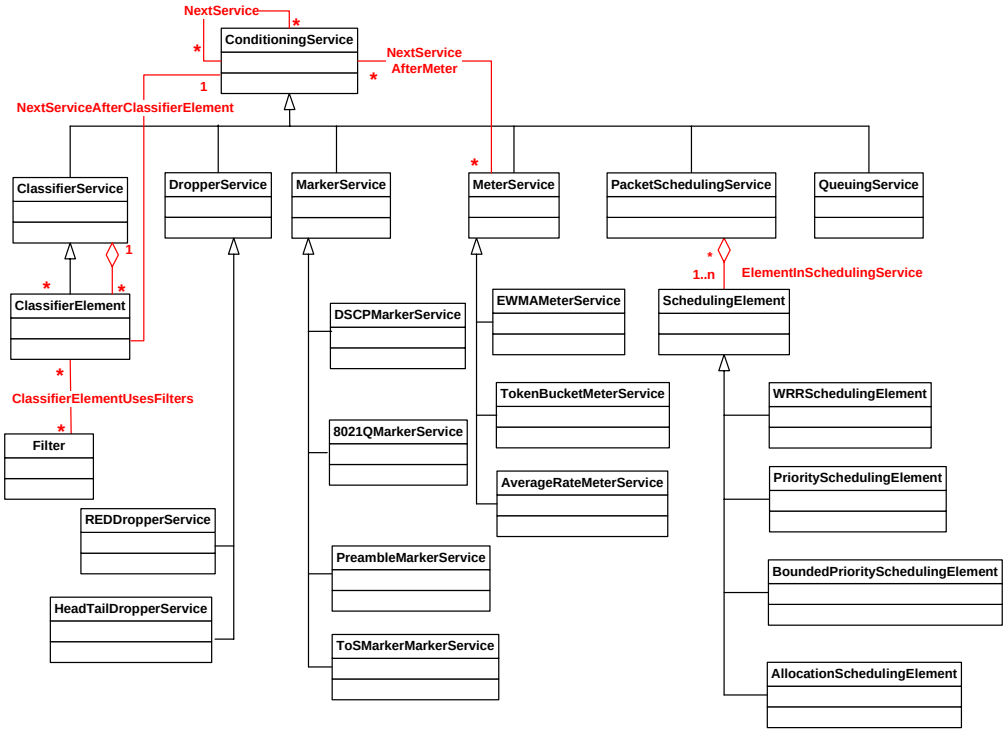


Figure 3.1 ConditioningService Classes in CIM Network Common Model

Our framework uses the CIM representation of the components of the Differentiated Services architecture to specify and enforce policy rules that apply to a DiffServ domain. Using the CIM representation of DiffServ not only provides our framework the necessary abstractions to specify policy independently of the type of the device, but also provides a flexible architecture for mapping the policy rules into device configuration independently of the communication protocol between the policy system and the managed devices. In the following, we will present, with examples, how DiffServ specific policy rules are constructed in our framework.

Example 3.1 presents a policy type, which can apply to any type of DiffServ device, and can be instantiated with device specific parameters. The administrator specifies this policy type to insert any MeterService element (i.e., any subclass of MeterService) on a router.

Example 3.1 Policy type that configures any MeterService mechanism in any type of DiffServ device

```
type oblig insertMeter (target router, MeterService meter) {
subject DiffServManager;
on addMeterRequest();
do router.addDiffServElement (meter);}
```

The parameter meter used in this policy type is an object of the CIM class MeterService. This example assumes that information of the location of this object inside the device's space (this information is specified by the association NextServiceAfterMeter in Figure 3.1) is kept within the object meter. The Policy Management Agent can detect the actual subclass, which may correspond to one of those identified above, and if necessary, take specific actions for this particular subclass. The same insertMeter type can be instantiated multiple times in order to create specific rules, which apply to specific devices within the DiffServ domain. For example, the administrator may want to insert the AverageRateMeterServiceA on the edge routerA and the TokenBucketMeterServiceB on the core routerB. Two policy instances should be created for that purpose, as shown in Example 3.2.

Example 3.2 Instantiation of the insertMeter policy type for different devices

```
inst  Config1 = insertMeter (/Routers/EdgeRouters/RouterA,  AverageRateMeterA);
inst  Config2 = insertMeter (/Routers/CoreRouters/RouterB,TwoParameterTokenBucketMeterB);
```

3.1.1 Policy rules for managing a set of devices

Since subjects and targets are explicitly specified in Ponder policies in terms of domains, the same policy can apply to sets of devices rather than individual ones. For example, an administrator may want to impose the same classification rules on all edge routers in a DiffServ domain, in order to provide a common set of DiffServ codepoints (DSCPs). The solution provided by our approach is a policy rule whose scope is not a particular device, but all devices within the defined domain.

Example 3.3 Policy that adds the same classifier entry to all devices within DomainA of edge routers

```
inst oblig insertClassifierToDomainA {
subject DiffServManager;
target r = /Routers/EdgeRouters/DomainA;
on AddClassifierEntryRequest(ClassifierEntry);
do r.addEntryToClassifierService(ClassifierEntry);}
```

When an AddClassifierEntryRequest event occurs (this event is usually triggered through the management console), this rule will insert the same ClassifierEntry object (this object belongs to the CIM Class ClassifierElement) in the ClassifierService elements of all routers that belong to the domain /Routers/EdgeRouters/DomainA. Moreover, if a new device is later added to that domain, the corresponding Policy Management Agent DiffServManager will be notified and will apply the policy to the newly added device automatically.

3.1.2 Policy rules for implementing DiffServ Per Hop Behaviours

Per-Hop Behaviour specifies the treatment packets should receive on a DiffServ node, as they are traversing a sequence of functional datapath elements. The implementation of both standard and vendor specific PHBs can be provided by using more complex policy rules, whose specification can be easily altered to provide varying PHBs.

In Example 3.4, we present a policy, which configures a set of routers within the DS domain to implement the EF PHB. In this example, traffic above the configured maximum input rate is degraded to receive the Best Effort PHB. The EFConfigurationPolicy is triggered by the administrator's request EFConfigRequest and will configure all routers within the target domain.

Example 3.4 Policy rule for implementing the EF PHB within a given set of core routers

```
inst oblig EFConfigurationPolicy {
subject DiffServManager;
target r = /DomainA/Routers/CoreRouters;
on EFConfigRequest(DSCP,b,r,p);
do r.applyEFPHB(DSCP, b,r,p);

/* The first event parameter, DSCP, specifies the DiffServ codepoint for the EF PHB,
usually 0x2e.The remaining event parameters, b, r and p, will be used to configure the
metering elements within the core routers so that only traffic conforming to the
TokenBucket traffic profile with parameters (b,r,p) will be treated with the EF
behaviour. More specifically, b specifies the size of the bucket, p specifies the peak
rate of the traffic that the bucket accepts and finally, r, specifies the rate for generating
tokens. */
```

An example of the implementation of the complex policy action applyEFPHB is outlined with the pseudocode shown in Figure 3.2, which is technology-specific and assumes that the Weighted Round Robin scheduling algorithm can be used. Alternative implementations could use a Priority or a Class Based Queueing (CBQ) scheduler. Which scheduling algorithm should be used for implementing the EF PHB, is an administrator's decision and must comply with the restrictions imposed by the technology-specific implementation of DiffServ within different devices, by different vendors.

```
applyEFPHB(DSCP, b,r,p) :-
    Filter filter = new Filter ( DSCP);
    ClassifierElement cls = new ClassifierElement();
    cls.addFilter(filter);
    /* b,r and p are the TokenBucket parameters. b is the size of the bucket, p the peak rate of the
     traffic that the bucket will accept and r the rate for generating tokens*/
    TokenBucketMeterService tb = new TokenBucketMeterService (b,r,p);
    cls.setNextService (tb);
    /* The value 1 specifies that this class will receive the highest priority from the WRR scheduler,
     as this will be the EF traffic. The second parameter indicates the rate at which we want the
    traffic to leave the network interface, which must be on average less or equal to the rate at
    which the bucket is filled with new tokens. Therefore, we specify  "r" to be the rate for departing
    the WRR scheduler */
    WRRSchedulingElement sched = new WRRSchedulingElement ( 1,r );
    tb.setNextService(sched);
```

Figure 3.2 Implementation of the applyEFPHB action

Note that failures of execution of a policy action can be detected by our management system, as an optional "catch" clause is supported for this purpose by the policy specification itself, see Figure 2.2. An exception raised by a catch clause can be sent to the administrator's console in order to notify him that some action(s) has/have failed to execute on a subset of devices within the subject or the target domain. Moreover, the exception indicating this type of failure could act as a triggering event for another Ponder obligation rule, which the administrator has a priori defined for resolving the possible reasons that caused the failure of the policy action. More information about how this can be implemented will be discussed in Chapter 5 of this thesis.

Policy actions can be either methods, implemented in a programming language, within the Policy Management Agents' policy engines, or as scripts that are interpreted at run-time by script engines within the Policy Management Agents. It would be possible to perform synchronisation of actions on a set of distributed target objects using two phase locking. Similarly, all-or-nothing semantics could be implemented using a two-phase commit protocol implemented by a script. However, this is not recommended for large numbers of routers as it is likely that not all of them will be available when the action is

executed, so this approach will fail. A better approach is to use a script which performs the action on all target objects which are available and updates a list of objects on which the action failed. The action could be periodically repeated until all target objects are updated i.e. the system converges to the required state over time. This assumes that the set of objects in the target domain can work normally if the action is performed on some of the objects and not others, so the approach to use is application dependent.

Note that in the current implementation of the Ponder toolkit, subject and target policy objects are implemented as Java RMI objects, so we implement complex policy actions as remote methods within subject/target objects' engines.

In the examples we have presented, the implementation of the policy action is based on the creation of individual CIM objects. The interconnection between CIM objects (ie. their place in the CIM agent handling the CIM objects) is provided by the method setNextService, which sets the NextService association (refer to Figure 3.1). In the following section, we will present how CIM objects that the Ponder policies create are downloaded to the target routers.

Also note that in our framework, a policy rule is always active, unless it has been explicitly disabled by the administrator. Policy Management Agents can receive events (other than configuration requests) relating to changes in the system and then enforce the relevant policies. For example, when a monitoring service detects a high drop rate of the EF traffic class, it can raise an EFConfigRequest with a higher maximum input rate value. This dynamic triggering of policies is one of the mechanisms used for adaptive management in our framework, as we will present in Chapter 4 of this thesis.

3.2 Implementation of Policy Enforcement for DiffServ Routers

Ponder policy actions in our framework involve adding, removing or updating DiffServ functional elements represented in the CIM Network Common Model. We have implemented a generic enforcement architecture to communicate the policy actions to DiffServ-enabled devices. This architecture abstracts the lower-level protocols used for communication with the devices' management interfaces. Figure 3.3 presents the main components of our policy enforcement architecture.

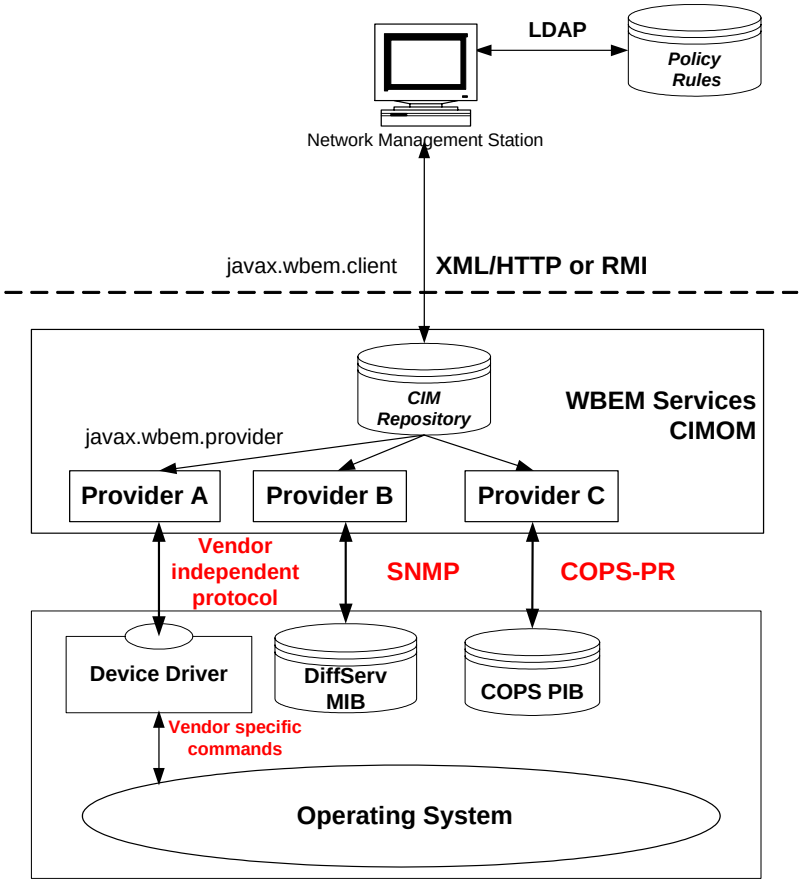


Figure 3.3 Policy Enforcement Architecture

Two main components comprise this enforcement architecture:

- The Network Management System, where Ponder policies are edited, compiled and distributed to their corresponding Policy Management Agents (PMAs). The Ponder deployment model [Dulay et al. 2001] is used for the distribution of policies to the PMAs. We extended the functionality of PMAs and Target Policy Objects with the capability to act as CIM clients to allow these components to add/update/remove CIM objects.

- A CIM Object Manager (CIMOM) that handles requests from the Ponder CIM clients for adding/updating/removing of CIM objects. A set of Provider components are attached to the CIMOM to communicate CIM objects to a specific DiffServ device using an available management protocol, eg. a vendor specific protocol, SNMP or COPS-PR, as we indicate in Figure 3.3.

In the prototype implementation of the policy enforcement architecture, the DiffServ classes of the CIM Network Common Model are implemented within the CIM Object

Manager (CIMOM) from the WBEM Services project [WBEM Services Project 2003]. It provides a Java implementation of a CIMOM with two communication interfaces as shown in Figure 3.3. These are:

- The javax.wbem.client interface, used by Ponder clients for transferring CIM objects to and from the CIMOM. This communication can be realised either through HTTP (where CIM operations are defined in XML) or through Java RMI.
- The javax.wbem.provider interface, which providers attached to the CIMOM use to communicate with the CIMOM. Providers are implemented as Java classes and each Provider is responsible for handling a number of CIM classes.

We have implemented a Provider (Provider A in Figure 3.2) specific for Linux systems. This Provider downloads CIM objects through the Java RMI interface to a Linux Driver, namely CIM2TCDriver, which we have designed and implemented in Java. The CIM2TC Linux Driver is used to translate CIM classes specified in the CIM Network Common Model to traffic control commands [Linux Advanced Routing & Traffic Control HOWTO 2002] for configuring the available DiffServ mechanisms of a Linux router, which we presented in section 2.1.3.

Note that the management system does not communicate directly with the CIM2TCDriver, but via the CIMOM. This provides support for using the management system and the same Provider to enforce policies that apply to other types of DiffServ devices other than Linux routers. This would only require the implementation of alternative device drivers. It is also possible to use SNMP or COPS-PR to configure the Linux device, as indicated by Providers B and C in Figure 3.2, which could issue SNMP set requests to set MIB variables within the DiffServ MIB (as described in [Kim et al. 2000] or COPS-PR messages to transfer information to a DiffServ PIB respectively. However, we were unable to find a complete SNMP or COPS-PR implementation for Linux, so we have not implemented this.

As mentioned earlier, our prototype implementation uses the CIM2TC driver to configure a Linux router using traffic control (“tc”) commands. This approach is similar to that of [Martinez et al. 2002], in which a driver component is used to translate classes that follow the DiffServ MIB object model [Baker et al. 2001] to Linux traffic control commands. However, this prototype implementation used a translation algorithm with limitations in terms of the types of low level mechanisms it can configure, but we have used the “jtc” package from this implementation to represent the traffic control mechanisms of the

Linux DiffServ router which is indicated as the LinuxDiffServ.LinuxDriver.tc package in our implementation, as shown in Figure 3.4.

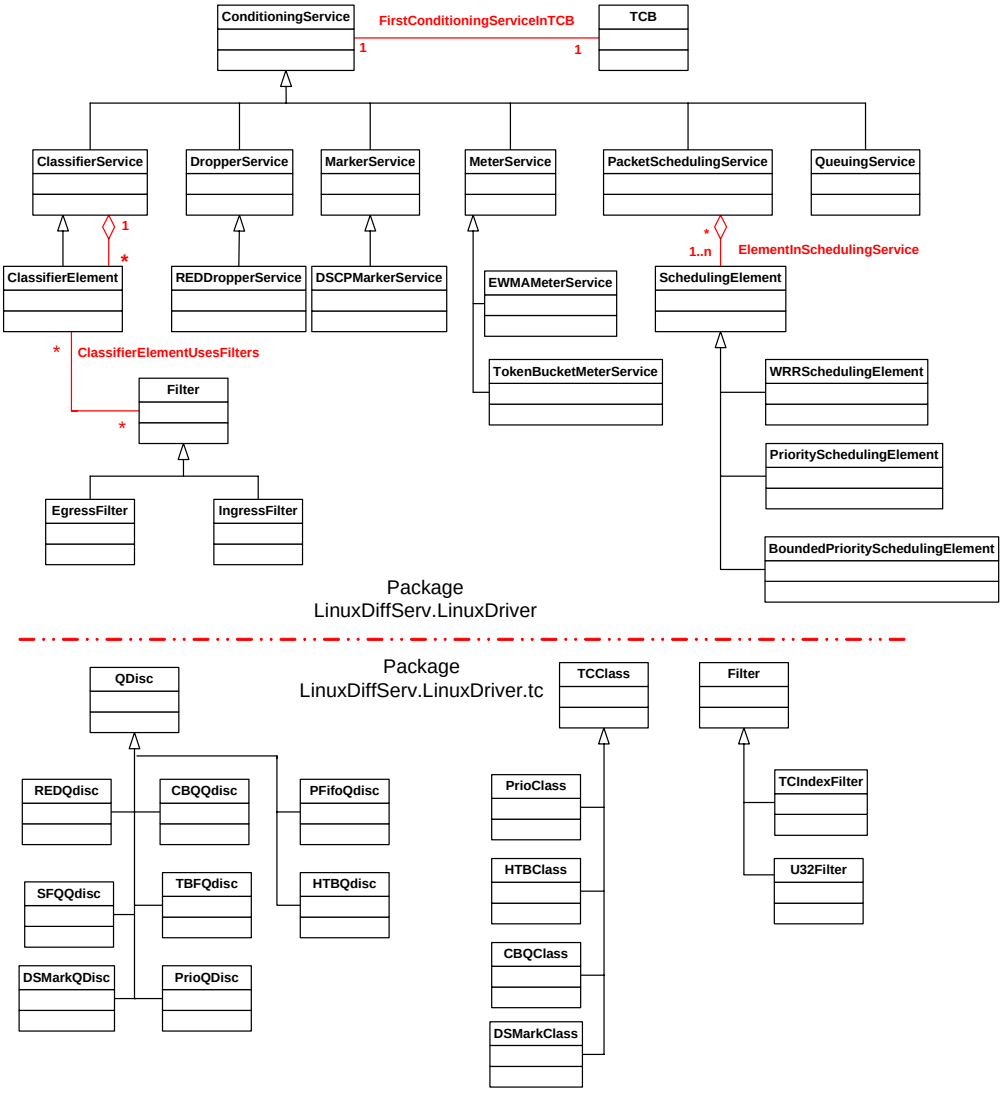


Figure 3.4 Object Model used by the CIM2TC Driver component

Our implementation uses a new algorithm for translating the DiffServ classes of the CIM Network sub-model to the correct low-level “tc” objects, as indicated in the pseudocode outline for the CIM2TCDriver in Figure 3.5.

```
foreach tcb in tcbs[] {
        // create policing filters at the ingress interface
        if (tcb.interfaceDirection == “ingress”) then {
                foreach MeterService in tcb create policing filters;}
        else { // Create tc qdiscs and classes at the egress interface
                if (deviceIsNotConfigured) then outerQdisc = new DSMarkQdisc();
                schedulingElement = tcb.getSchedulingElement();
                filters[] = tcb.getFilters();
                if (schedulingElement is a WRRSchedulingElement) then {
                        innerQdisc = new CBQdisc( … );
                        DiffServClass = new CBQClass( …); }
                else // other types of schedulers
                 if (schedulingElement is a PrioritySchedulingElement) then {
                        innerQDisc = new PrioQdisc( …);
                        DiffServClass = new PrioClass( … );}
                else
                if (schedulingElement is a BoundedPrioritySchedulingElement) then {..}
            } //end of if clause for interface direction

        // Create metering, marking and dropping classes and qdiscs

        foreach element in tcb after ClassifierElement and before SchedulingElement {
        if (element is a DSCPMarkerService) then add new DSMarkQdisc ( … ) ;
        if (element is a TokenBucketMeterService) then add new TBFQdisc (…) ;
        if (element is a REDDropperService) then add new REDQdisc(…); }
        // Create the filters that will direct the packets to the DiffServ class
        createTCFilters (filters[]); }
```

Figure 3.5 Pseudocode for the CIM2TC Driver component

The CIM2TCDriver takes as input a set of instances of the Traffic Control Block (TCB) class. We extended the CIM network sub-model with the TCB class, whose scope is described in [Carlson et al. 1998], in order to represent higher-level blocks constituted by combinations of ConditioningService CIM objects. A TCB object uses the association FirstConditioningServiceInTCB to hold the first ConditioningService; the other ConditioningServices are retrieved using the association NextService already defined in CIM. The TCB objects are downloaded to the CIM2TCDriver from Provider A shown in Figure 3.2, which handles CIM requests for objects whose classes belong to the package Linux.DiffServ.LinuxDriver, presented in Figure 3.4. The CIM2TCDriver parses each TCB chain using the translation algorithm outlined above in order to derive the set of “tc” objects that correspond to the input TCB. Figure 3.6 gives an outline of the mapping of

CIM DiffServ functional elements to Linux traffic control structures (i.e. qdiscs, classes and filters described in [Linux Advanced Routing & Traffic Control HOWTO 2002]).

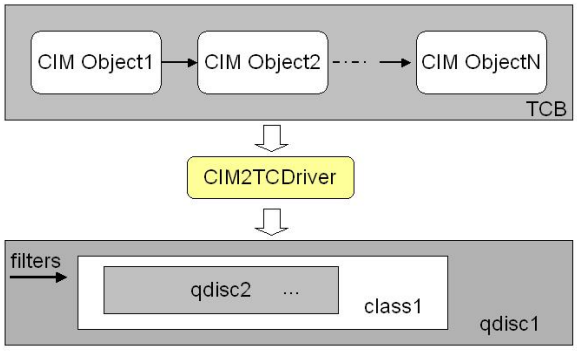


Figure 3.6 Mapping of CIM objects to Linux traffic control structures

A usage example of the CIM2TCDriver is presented in Figure 3.7, which shows a Graphical User Interface [Zelu 2003][1], developed for using the driver to configure Linux routers.

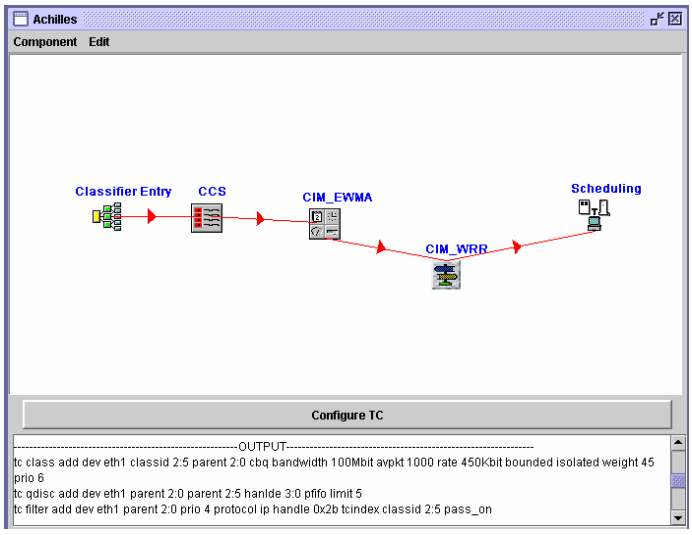


Figure 3.7 DiffServ configuration GUI

Note that one limitation of the mapping of CIM objects to Linux traffic control commands is that it is not possible to use our algorithm, in order to communicate any arbitrary chain of CIM DiffServ objects to a Linux router. This is due to the limitations of

[1] This MSc thesis was carried out under my supervision. The GUI uses the CIM Network Common Model to represent DiffServ functional elements and interfaces with the CIM2TCDriver to send the "tc" configuration to the device.

the Linux traffic control architecture [Almesberger 1999], presented in section 2.1.3. More specifically, it is not possible to communicate a chain consisting of more than one TokenBucketMeterService or more than one REDDropperMeterService CIM objects. The same applies when there exist together one or more TokenBucketMeterService and one or more REDDropperMeterService objects in the chain. This happens due to the fact that the corresponding Queuing disciplines (TBFQDisc and REDQDisc) are classless, i.e. it is not possible to use within them another Queuing discipline or another Class components. Furthermore, one feature that we have not implemented is handling of recursive Traffic Control Block (TCB) structures. Our algorithm takes as input simple TCB structures, i.e. TCBs that are of the form of a linked list of CIM objects. However, since multiple schedulers may be used for the implementation of compound TCBs, the latter can be recursive structures, ie. complex TCBs that consist of simpler TCBs. This could be implemented as future work.

3.3 Usage Example of Policy Enforcement for Linux Routers

In the following, we will provide an example of usage our prototype implementation on a testbed DiffServ network of Linux routers. The network topology we set up for our experiments is presented in Figure 3.8.

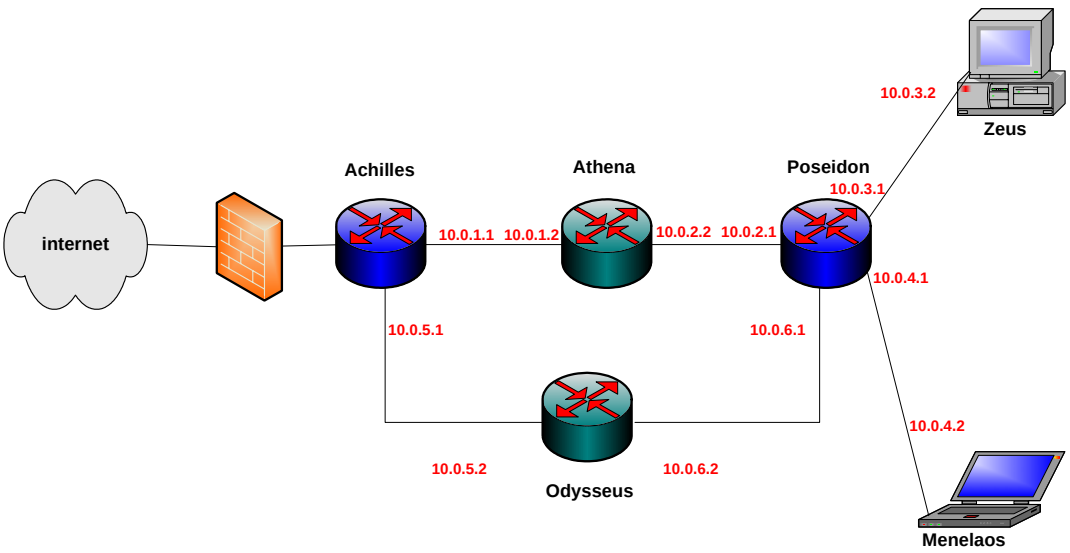


Figure 3.8 Network topology of testbed DiffServ network

The core routers in our testbed DiffServ network are the Linux machines *Athena* and *Odysseus*. The objective of this example is to enforce a policy rule similar to the one presented in Example 3.4 for implementing the EF PHB within all the core routers within the DiffServ domain. The administrator uses the Editor tool of the Ponder Toolkit (shown in Figure 3.8) to edit and compile the policy rule EFConfigurationPolicy. Figure 3.9 displays the policy objects stored in the LDAP directory server, which the Ponder Toolkit uses for storing policies, policy subjects and policy targets.

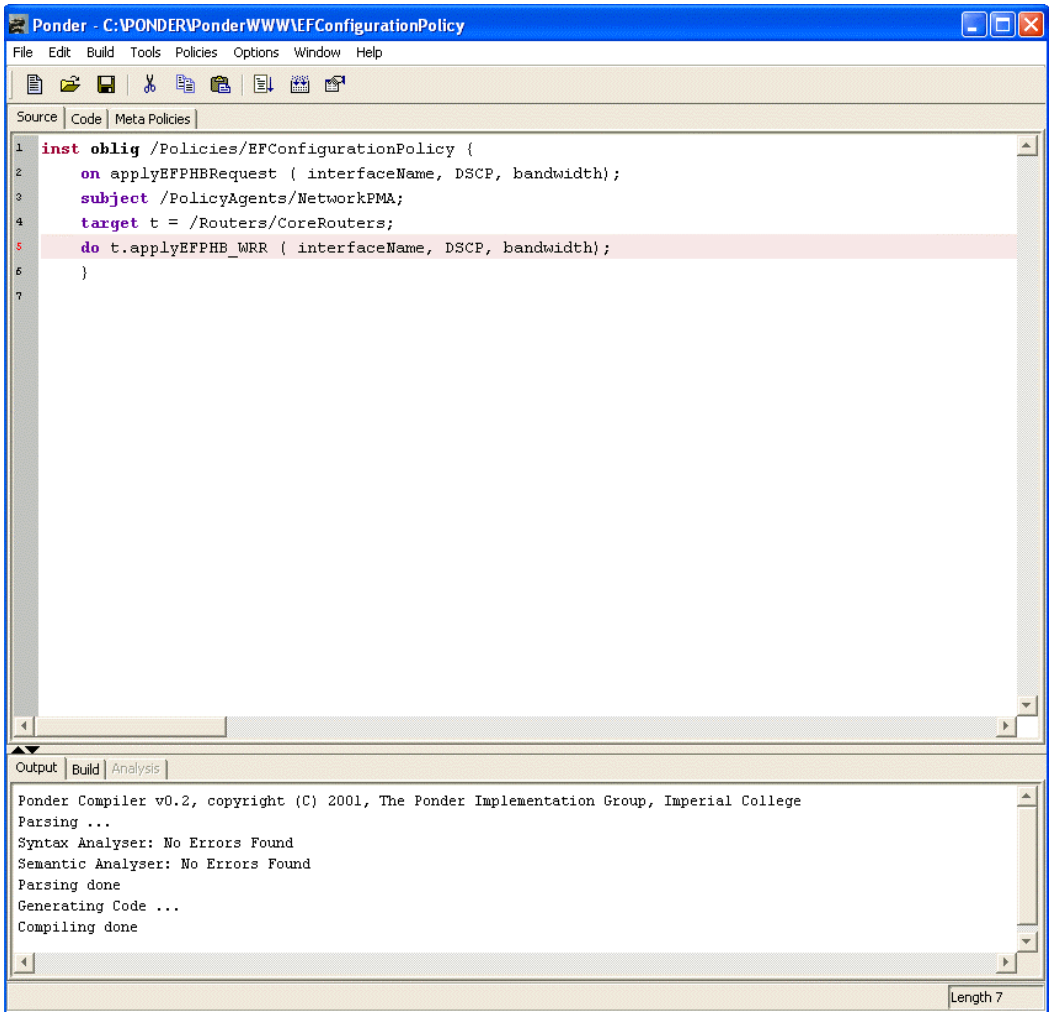


Figure 3.9 Ponder Editor

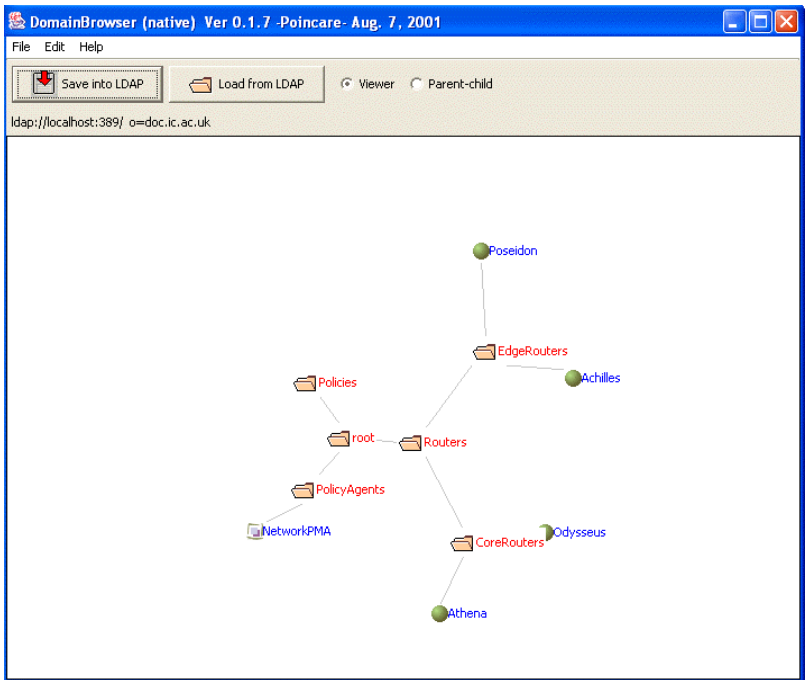


Figure 3.10 Ponder Domain Browser

The policy EFConfigurationPolicy uses the complex action applyEFPHB_WRR, which constructs the TCB shown graphically in Figure 3.7. In our current implementation, this policy action is implemented as a Java method within the Policy's Target Object, which is implemented as a Java RMI object, representing the managed device within the Ponder toolkit. The functionality of the action applyEFPHB_WRR is outlined by the following Java code.

```
public void applyEFPHB_WRR (String interfaceName, int DSCP, double classRate)
{

        TCB tcb = new TCB(interfaceName);
        EgressFilter filter = new EgressFilter ((byte)DSCP);

        // Creates a ClassifierElement object
        ClassifierElement cls = new ClassifierElement();

        cls.addFilter(filter);

        // Creates a EWMAMeterService object
        EWMAMeterService tb = new EWMAMeterService(classRate);

        cls.setNextService (tb);

        // Creates a WRRSchedulingElement object with the highest scheduling priority: 1
        WRRSchedulingElement sched = new WRRSchedulingElement(1,classRate);

        tb.setNextService(sched);

        // Creates the TCB
        tcb.setStart(cls);

        // Communicates the TCB to the CIMOM
        CIMOM.sendTCB(tcb);

}
```

Note that further implementation work could integrate the Ponder Editor with the Graphical User Interface shown in Figure 3.7, so that the administrator is able to graphically specify policy actions that involve CIM classes to add/update/delete TCBs within the managed devices.

The policy rule EFConfigurationPolicy is enforced by the management agent NetworkPMA. The NetworkPMA receives the obligation event EFPHBRequest with parameters indicating the network interface of the target device where the EF PHB is to be implemented, the DSCP associated with this PHB and the bandwidth that will be assigned to the EF traffic class. Figure 3.10 displays the "tc" commands that the CIM2TCDriver, which is running within the target core router Athena, generated for performing the policy action applyEFPHB_WRR, with parameters the list ("10.0.2.2", 46, 10e6) when the action is implemented as the TCB presented in Figure 3.7. It also displays statistics about the DiffServ classes implemented in the core router *Athena*, where the policy action applyEFPHB_WRR is enforced.

```
[root@Athena root]# ./RunDriver.sh
      tc class  add  dev eth1 classid 2:4 parent 2:0 cbq bandwidth 100Mbit avpkt 1000 rate 10Mbit
bounded  isolated  weight 1Mbit prio 1
      tc qdisc add dev eth1 parent 2:4 handle 3:0 pfifo  limit 5
      tc filter  add  dev eth1 parent 2:0 protocol ip prio 4 handle 0x2e tcindex  classid 2:4 pass_on

[root@Athena root]# DiffServMonitor &
ClassID   Queing     Rate     Sent packets      Sent bytes      Dropped Packets  Bandwidth (bps)
   2:1      cbq    1Mbit          9372        14121152              215                1102192
   2:2      cbq    2Mbit             0               0                0                      0
   2:3      cbq    2Mbit         10415        15406832              635                2194376
   2:4      cbq   10Mbit             0               0                0                      0
 ClassID   Queing     Rate     Sent packets      Sent bytes      Dropped Packets  Bandwidth (bps)
   2:1      cbq    1Mbit          9558        14401308              219                1120624
   2:2      cbq    2Mbit             0               0                0                      0
   2:3      cbq    2Mbit         10754        15917214              647                2041528
   2:4      cbq   10Mbit           506          762732                4                3050928
```

Figure 3.11 Result of enforcement of the rule EFConfigurationPolicy

Since there does not exist an implementation of a Linux command line traffic monitoring tool for providing the throughput, round trip delay and packet loss metrics per each implemented DiffServ class, we have implemented for this purpose three "probe" components with the Perl scripting language. The first probe that we implemented (throughput monitoring probe) monitors the traffic within a Linux core router by issuing to the kernel periodically traffic control statistics commands ("tc –s" commands). It then uses the output of the traffic control statistics commands to provide the throughput of each implemented DiffServ class at the egress interfaces of the Linux core router where the probe is running. The bandwidth data for the core router *Athena* shown in Figure 3.10

is provided by the bandwidth monitoring probe. The second probe (round trip delay probe) provides information about the round trip delay between a *source host* and a *destination host* within the DiffServ domain. It runs within the *source host* and monitors actively the round trip delay metric between that host and the *destination host* by issuing periodically "ping" requests to the latter host. The administrator can use this probe in order to obtain the round trip delay experienced by a particular traffic class. A typical usage of this probe is to monitor the round trip delay between an edge router (e.g. Achilles in Figure 3.7) and several destination hosts (e.g. Zeus and Menelaos in Figure 3.7) within the DiffServ network, each one receiving a particular class of service (e.g. if Zeus is receiving "gold" service and Menelaos is receiving "bronze" service). The third probe (packet loss monitoring probe) operates in a similar way to the bandwidth monitoring probe, providing the packet loss experienced by each implemented DiffServ class within the network.

To graphically display the metrics provided by our probes within a monitoring station (this station must not necessarily be inside the DiffServ network), we have implemented with the C++ programming language a graphical tool for generating realtime graphs showing the throughput, round trip delay and packet loss per each implemented DiffServ class. An instance of an execution of this tool is shown in Figure 3.12.

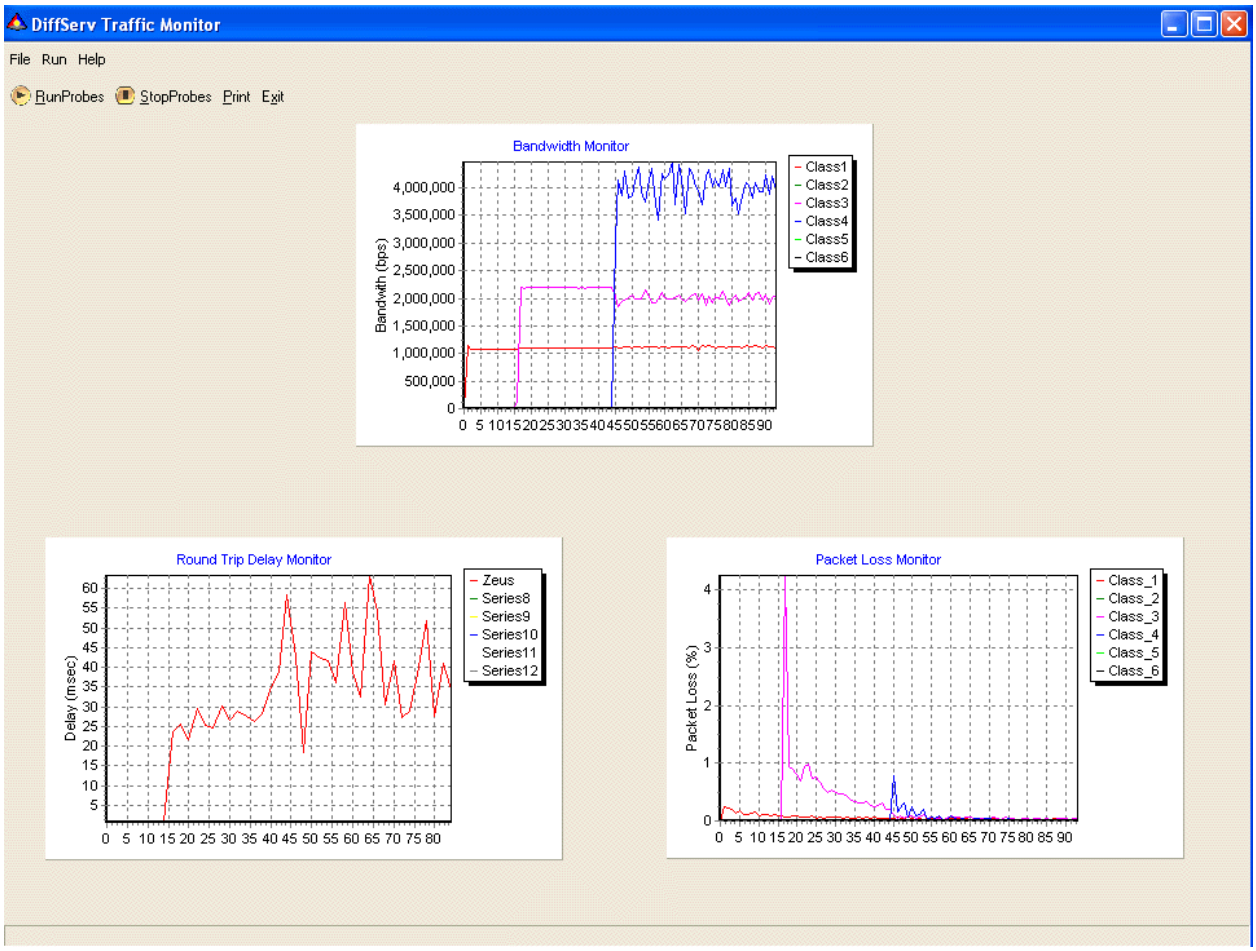


Figure 3.12 DiffServ Traffic Monitor Tool

The administrator can use the DiffServ Traffic Tool to monitor the results of the enforcement of his policy rules within the network. For example, in the scenario shown in Figure 3.12, the administrator used the tool to monitor the core router *Athena*, before and after the execution of the policy EFConfigurationPolicy, shown in Figure 3.8. As the graphs in Figure 3.12 show, traffic marked with the EF DSCP (0x2e) and received by the client host *Zeus* originally was falling within the Best Effort traffic class, indicated with the fuchsia lines in the bandwidth and packet loss graphs. After the enforcement of the policy action applyEFPHB_WRR, the new class of traffic was successfully created within the router *Athena*, so that the EF marked packets are treated with higher priority over the other implemented traffic classes. Finally, note that although the new class receives higher priority, the round trip delay between the edge router *Achilles* and the client *Zeus* increased after the creation time of the EF traffic class. This is due the large amount of ftp traffic that the client *Zeus* started receiving after the creation of the EF class.

3.4 Discussion

This chapter presented a novel policy network management framework which uses the Ponder language for specifying policy and CIM as the modelling framework for network mechanisms. We presented how device-independent Ponder policies can be specified using the CIM representation of network mechanisms, and we described a generic architecture for mapping Ponder rules into device configuration independent of the communication protocol between the policy-based management system and the managed devices.

We presented an example application of our framework for the management of Differentiated Services networks. We described how policy rules for configuring individual DiffServ mechanisms or more complex policy rules, such as rules for establishing either standard (i.e. AF, EF) or used-defined Per-Hop Behaviours in a DiffServ domain can be expressed in our framework. It would also be possible to use our framework to define policies in terms of Service Level Objectives that the managed network must satisfy, eg. a policy rule that guarantees *Gold Service* to certain users or applications, by using. EF PHB. This would require a policy refinement process, such as the one described in [Verma 2001], to derive network-level policies that are expressed in terms of PHBs from the higher-level policies, expressed in terms of Service Level Objectives.

We also presented our prototype implementation for policy enforcement for Linux DiffServ routers, which uses a Linux driver component for communicating the policy actions. This driver implements an algorithm that we designed for translating CIM classes to Linux traffic control classes. Management of different network environments, such as MPLS can also be realised with our framework by specifying policy in terms of CIM represented MPLS mechanisms and using the same enforcement architecture for communicating MPLS-specific policy actions, using available management protocols or device-specific drivers.

Compared to the policy-based systems used for network management (IETF's policy framework, and commercial policy toolkits), which we presented in section 2.3, our policy framework explicitly identifies the policy targets and allows grouping of targets into domains, and such, as we already discussed in section 2.1, our framework provides scalability with respect to the number of policies that need to be specified and deployed within the managed network. We also give the administrator the flexibility to specify

policies at different levels of abstraction and uses a generic enforcement architecture for policy deployment, independent of the lower level management protocols. Furthermore, network management policies defined in our framework are not static. Rather, they can be dynamically triggered by events, with different parameters, in order to automatically change the configuration of the managed objects under changing circumstances. The dynamic configuration of policies forms the basis of the adaptive management our framework can provide.

Chapter 4

Policy Adaptation within the Ponder Framework

When applying policies to network elements, the policy actions are those provided by the management interface of the managed element. Thus, the "level of abstraction" of the policies is determined by the available implementation. However, as we discussed in the introduction of this thesis, network management with a policy-based management system may require adaptation of existing network policies to cater for changes within the managed network. Thus, policies themselves need to be managed and adapted. In this chapter, we identify different adaptation requirements and show how policy adaptation can itself be specified and enforced by other policies, specified in the same Ponder policy notation.

We use the term *"Policy Adaptation"* to describe the ability of the policy-based management system to modify network behaviour in one of the following ways:

- Adaptation by dynamically changing the parameters of a network management policy to specify new attribute values for the run-time configuration of managed objects.
- Adaptation by selecting and enabling/disabling a policy from a set of pre-defined network management policies at run-time. The parameters of the selected network management policy are set at run-time.
- Adaptation by learning which are the most suitable policy configuration strategies from the system behaviour. This can be used to select policies or even generate new ones when needed.

In this chapter, we will focus only on the first two categories of policy adaptation as adaptation by learning still requires considerable further work. The rest of the chapter is organised as follows. Sections 4.1 and 4.2 will present how policy adaptation is realised for each of the first two categories outlined above. Section 4.3 will present the enforcement architecture for an adaptive policy system which implements our new ideas. Sections 4.5, 4.6 and 4.7 will provide usage examples where the proposed adaptive policy framework caters to the adaptive management of a Differentiated Services network, in a network security scenario and in a ubiquitous environment respectively. A description of the implementation of our policy adaptation enforcement architecture in a network

simulator will be given in section 4.7, as well as some results from conducted simulations. Section 4.8 will conclude the chapter with a discussion of the overall work achieved.

4.1 Run-Time Modification of Policy Parameters

In the general case, the specification of a network-level management policy used for dynamically adapting the managed devices' configuration (i.e. configuration of DiffServ, MPLS or other types of network mechanisms) follows the format shown in Figure 4.1.

```
inst oblig NetworkPolicy {
subject NetworkLevelPMA;
target targetSet = TargetDomainofDevices;
on Event(EventParameters[]);
do ActionParameters[] = CalculateActionParameters(EventParameters[]) ->
    targetSet.executeAction (ActionParameters[]); }
```

Figure 4.1 Generic format of a network management policy

In Figure 4.1, an event triggers the execution of the policy in one or more subjects, i.e. Network-level Policy Management Agents (PMAs). Using the EventParameters, the PMA calculates the required policy ActionParameters, by calling the internal method CalculateActionParameters, and then it invokes the relevant policy actions on the target objects in the TargetDomainofDevices with the new policy parameters.

Policies provide a flexible means for providing this type of adaptation rather than components written in a procedural language. New adaptation strategies can be incorporated into the management system by adding new policies which react to different events using the existing policy actions or by replacing existing policies with new versions, which either implement new actions on the managed objects or new actions on the Policy Management Agents. With programmable networks, new actions may be added, via a management interface in network elements, so the policies can be updated to access this new functionality. The code that implements new actions or new calculation methods within the Policy Management Agent's engine can be loaded at run-time either through the administration console or by the new policies themselves. If the functionality of the PMA were implemented using a traditional programming language, it would be necessary to recompile the code and replace the agent which would require stopping the system while this was being done.

4.2 Adaptation by Dynamically Selecting and Enabling Policies from a Set of Policies

In this approach, obligation policies at the service level (we will use the term service management policies for policies defined at the service level) are triggered by events indicating changes within the managed environment and determine which lower-level network policy must be enabled/disabled to adapt the configuration of the managed system. As we discussed in the previous section, the advantage of using policies rather than a procedural language for selecting and enabling the appropriate network-level policies is that modifying or adding new management strategies at this level can be achieved by replacing the service management policy or adding new ones. Furthermore, the same Ponder deployment framework can be used to distribute both service management policies and network policies [Dulay et al. 2001]. Figure 4.2 presents the location and the functionality of service management policies within the network management system.

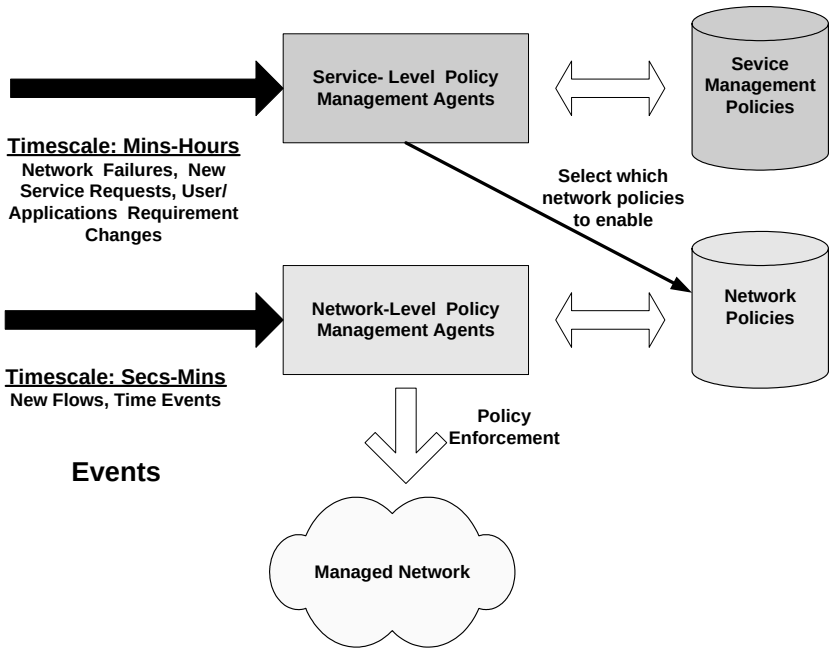


Figure 4.2 Service management policies for policy adaptation

In the general case, a higher-level service management policy is specified with the template obligation rule ServiceManagementPolicy, presented in Figure 4.3.

```
inst oblig ServiceManagementPolicy {
subject ServicePMA;
on AdaptationRequest (params[]);
do NetworkPolicy = selectPolicy (params[])-> NetworkPolicy.enable();
    // The following apply when the network policy is an obligation policy
   ->NetworkPolicy'sParams [] =  calculate (NetworkPolicy, params[]) ;
   ->EventService.GenerateEvent( NetworkPolicy'sObligationEvent,
                                        NetworkPolicy'sParams []);}
```

Figure 4.3 Specification of a generic control management policy

Note that in the specification of the generic service management policy, a network policy can be either an obligation management policy or an authorisation access control policy. This implies that the same mechanism is used to specify how network policy should be adapted independently of its application to management of a specific network mechanism.

In the policy shown in Figure 4.3, an AdaptationRequest event (such as new service request, network failure, changes in application/user requirement as shown in Figure 4.2) triggers the selection of network policies for configuring network elements such as routers or firewalls. The ServicePMA first selects the most appropriate network-level policy to actually implement the configuration, using the selectPolicy method and then the policy is enabled. The network-level obligation policy will be interpreted by either one or more Network-level PMAs, while a selected network-level authorisation policy will be interpreted by one or more Network-level Access Controllers (ACs).

In case of selection of obligation network-level policies, the parameters related to the specific policy are calculated; and, finally an event is sent via the event service to pass parameters and trigger the selected obligation policy. Note that the advantage of triggering the network-level obligation policy via the event service is that there may be multiple agents managing subsets of the network devices. These agents will all receive the event and configure their respective devices.

4.3 Enforcement architecture

In the general case, the management functionality of the generic Policy Management Agent ServicePMA is specified with the obligation rule ServiceManagementPolicy, presented in Figure 4.3.

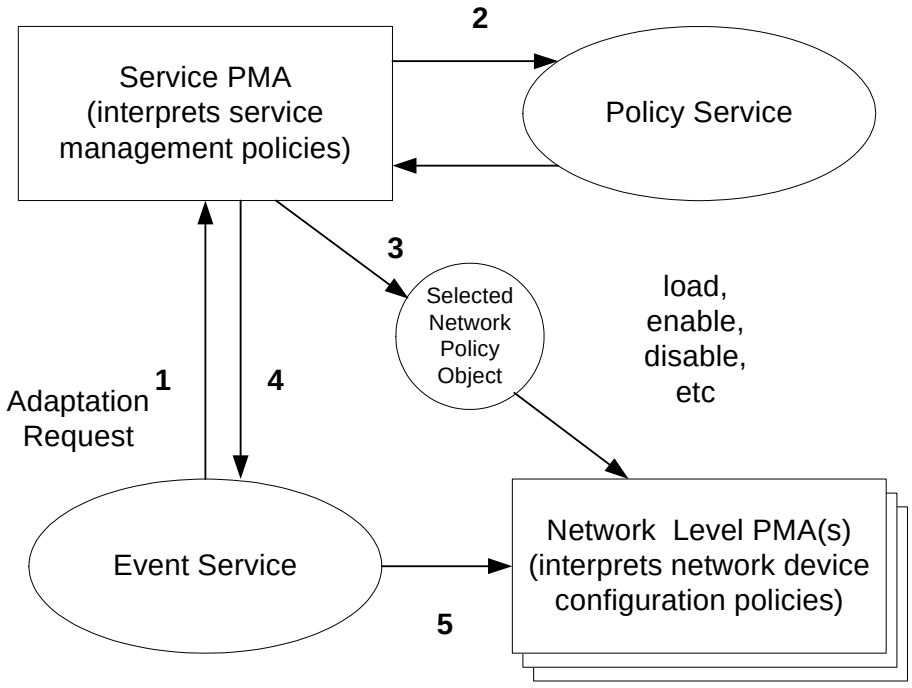


Figure 4.4 Enforcement architecture for policy adaptation

The enforcement architecture is presented in Figure 4.4.

1. The ServicePMA receives the event AdaptationRequest from the event service. As we discussed in section 4.2, the adaptation event could be from a new application requiring changes to QoS, performance measurements coming from a monitoring service that require changes in the existing network configuration, events indicating network failures or time events, security violations, etc.
2. The ServicePMA queries the current policy database from the policy service. The policy database contains references to all implemented network-level policies. The behaviour of network policies is described within the policy database with a common set of attributes, on which their selection is based. The administrator specifies the values of network policy attributes within the policy database. See Table 4.1 for example entries within the policy database. The ServicePMA then

invokes a selection algorithm to choose a suitable network policy from the set of implemented ones.

3. The enable() method is called on the selected network policy object, which in turn calls the enable() method on the relevant Network-level PMAs or ACs. At this point, the selected policy is activated in the relevant Network-Level ACs or in the Network-level PMAs. Furthermore, an "old" policy can be unloaded or disabled from the corresponding PMAs. In case of an obligation network policy, enabling the policy means that policy enforcement objects within the PMAs register the obligation event with the event service, as described in [Dulay et al. 2001]. However, the policy needs to be triggered by the obligation event to perform the policy actions. Steps 4 and 5 apply only to obligation network policies when they need to be triggered immediately after their selection (eg. for policies implementing a new class of traffic to handle a new application request). In other cases, the selected obligation policies will be triggered later (e.g. with a time event), hence steps 4 and 5 are not required.
4. The obligation event is generated with the network policy calculated parameters to trigger the policy.
5. The event service disseminates the obligation event to all Network-level PMAs that are registered to receive the specific event.

4.4 Adaptive Management of Differentiated Services Networks

In our approach, adaptation is enforced by higher-level policies. This section presents a usage scenario, where network policy that provides Per Domain Behaviour (PDB) in a Differentiated Services environment is adapted by service management policies. Service management policies are enforced by Policy Management Agents at the service level. The latter are responsible for the management of services that run within the managed DiffServ network.

Table 4.1 PDB policies and their QoS characteristics

PDB identifier	Enforcement Network Policy	Assured bandwidth (Mbps)	Delay (ms)	Jitter (ms)	Loss (%)	Enforcement Routers Path	Time when valid
PDB1	/Policies/Policy1	10	≤ 20	≤ 3	≤ 1	<r1,…, rN>	Every day
PDB2	/Policies/Policy2	20	≤ 10	< 1	≤ 0.1	<r1,…, rM>	Working hours
…	…	…	…	…	…	…	…

The IETF DiffServ working group has proposed in [Nichols et al. 2001] the term Per Domain Behaviour (PDB) to describe the behaviour experienced by a particular set of packets as they cross a DiffServ domain. A PDB is characterized by specific metrics that quantify the treatment a set of packets with a particular DSCP (or set of DSCPs) will receive as it crosses a DiffServ domain. A PDB specifies a forwarding path treatment for a specific aggregate. A PDB is implemented with one or more Per Hop Behaviours (PHBs). A PHB describes the forwarding behaviour that a DiffServ node applies to a particular DiffServ behaviour aggregate. PHBs are implemented in nodes by means of some buffer management and packet scheduling mechanisms. Each PDB has measurable attributes that can be used to describe what happens to its packets as they enter and cross the DiffServ domain. In our framework, each PDB is implemented as a network-level policy rule. Each rule guarantees the PDB attributes to the corresponding traffic aggregate. Table 4.1 presents examples of QoS guarantees that PDB policies can offer to their associated traffic aggregates.

In our framework, PDB policies are specified as Ponder obligation rules. The actual implementation of the PDB policy, i.e. the implementation of the PHB (or the set of PHBs) that will guarantee the QoS characteristics to the corresponding traffic aggregate, is hidden from the customer. The customer (human or automated agent) is offered the externally observable PDB QoS attributes. An example of a PDB Ponder policy rule is given below.

Example 4.1 Policy rule for providing a specific PDB

```
inst oblig /Policies/PDBPolicy1 {
subject /PMAs/DiffServAgent;
target r = /DiffServDomainA/Routers/CoreRouters;
on PDB1_ConfigRequest(DSCP, TokenBucketProfile(b,r,p));
do r.applyEFPHB(DSCP, TokenBucketProfile(b,r,p));
/* The PDB1 is implemented with the EF PHB. The event paramter DSCP, specifies
the codepoint for the EF PHB, usually 0x2e.The second event parameter is the
structure TokenBucketProfile and it will be used to configure the metering elements
within the routers so that only traffic conforming to the TokenBucket traffic profile with
parameters (b,r,p) will be treated with the EF behaviour*/
```

In this example, we assume that the core routers within the DiffServ domain can implement the EF PHB, so the PDB policy will configure them accordingly. A possible implementation described in [Jacobson et al. 1999] of the applyEFPHB action can use the Weighted Round Robin scheduling algorithm to schedule the packets at the egress

interface of each core router. Note that the policy rule in this example will configure the core routers within the DiffServ domain to implement the EF PHB on the corresponding traffic aggregate. An additional policy is needed to provide the necessary configuration to the edge routers of the DiffServ domain. When the network-level PMA DiffServAgent receives the event PDB1_ConfigRequest, it invokes the applyEFPHB action on all routers in the target domain. This way, all core routers within the target domain will guarantee a minimum output rate (throughput) to the EF-marked packets, when these packets do not exceed the configured maximum input rate at the ingress router interface. An alternative implementation of EF could use a priority scheduler at the egress interface of the target core routers.

Our current implementation extends the Ponder toolkit [Damianou et al. 2002] with the functionality to enforce DiffServ policies. Policies in the Ponder toolkit are Java RMI objects. The DiffServ specific policy actions (e.g. applyEFPHB) are methods within the policy object that the network-level Policy Management Agents invoke when triggered by the configuration request event.

The Per Domain Behaviour policies are enforced by DiffServ enabled Network-level PMAs (see Figure 4.4). These policies configure the QoS mechanisms of the managed devices within the DiffServ network. However, as we have already discussed, network service management requires additional functionality. The required functionality that enables dynamic service management is provided in our framework by Service Level Policy Management Agents. In the following, we will provide examples of service management policies for dynamic service management.

SLS to PDB mapping policy

SLS to PDB mapping can be performed by the ServiceManagementAgent when the administrator triggers the policy rule SLSMappingPolicy in Example 4.2 by means of an SLS request.

Example 4.2 SLS to PDB mapping policy

```
inst oblig SLSMappingPolicy {
subject ServiceManagementAgent;
on SLS_Request  (SLS_parameters[]);
do pdb_policy = select_using_algorithmA(SLS_parameters[])->
    pdb_policy.enable()->
    pdb_policy_parameters[] = calculate ( pbd_policy, SLS_parameters[])->
    EventService.GenerateEvent (pdb_policyObligationEvent,
                                    pdb_policy_parameters[]);}
```

A new SLS request from an application with specific parameters will trigger the SLSMappingPolicy. The ServiceManagementAgent uses the SLS parameters to select a suitable PDB policy from the policy database. The PDB policy is enabled and triggered to configure network devices. Note that we use the Tequila project SLS approach for DiffServ SLS parameter specification [D. Goderis et al. 2001]. A "parser" component within the Ponder management toolkit is used to translate the Tequila external SLS specification to pairs of <parameter, value>. These pairs are stored in the structure SLS_parameters and are conveyed to the agent with the obligation event SLS_Request. Upon the receipt of the SLS_Request, the ServiceManagementAgent will select the appropriate PDB policy for this specific service request. An example of a selection algorithm is outlined in [Prieto et al. 2001], where the PDB is selected according to the triple <delay, loss, throughput>.

Policy to handle service performance degradation

A number of different adaptation strategies could be used for handling service run-time performance degradation, notified by the monitoring service, as indicated in Figure 4.2. There may be a need to dynamically change these strategies by replacing a policy within the ServiceManagementAgent with a new version or by enabling/disabling different versions of the policy. Policies provide a more flexible means of implementing this type of service-level adaptation than scripts or special purpose code. Events indicating high delay or high packet loss could trigger policies in the ServiceManagementAgent. In the following examples, we indicate adaptation strategies, which could be implemented by Ponder policies for the management of a video client application within a DiffServ network, but we do not define the actual policies. In all the examples, we assume that the video application receives the EF network service and that the EF PHB is implemented as presented in Example 4.1.

The monitoring system detects that the EF service end-to-end delay exceeds a threshold so it generates a HighDelay event received by the ServiceManagementAgent. Corrective actions which may be performed include: a) Increase the minimum departure rate of the EF traffic at the egress of every core router to guarantee that the service packets (especially large ones) will remain in the output queue for less time before being transmitted to the next hop; and, b) Notify the client application to choose a different state, which requires less bandwidth and hence decreases the incoming traffic rate at the ingress interface. This way, the EF aggregate will experience less delay.

Measurements provided by [Ferrari et al. 2000] and from simulations in RFC 2598 [Jacobson et al. 1999] show that jitter is not reduced by increasing the EF service rate, when the EF aggregate is constructed from a single microflow. On the contrary, when the EF aggregation degree increases, jitter increases rapidly with the number of microflows and with the EF load. Thus, there are two possible corrective actions for a HighJitter event: a) Decrease the number of microflows, by degrading other EF traffic to receive a lower service. b) Reduce the EF load, by reducing resources assigned to the client application.

The action for a HighPacketLoss event would be to increase the maximum arrival rate of the incoming EF traffic at the ingress interfaces of both the edge and the core routers that the EF traffic traverses. This will reduce the number of packets being dropped by the policer at the ingress interface. Alternatively, as packet loss is proportional to the aggregation degree, the number of EF microflows can be reduced, in order to reduce packet loss in the remaining EF traffic.

Policy to support changes in routing or link failures

A PDB is usually associated with a path of routers within the DiffServ domain (e.g. when using DiffServ over MPLS). When a link fails or routing changes for a specific class of traffic, the corresponding PDB may not be guaranteed by the routers in the new path. A new PDB must be selected for this class of traffic that satisfies the network service QoS requirements and that can be served by the new path. Which PDB is selected and how, is a decision that can be formulated as a service management policy. The following example provides a policy that implements this functionality.

Example 4.3 Policy for configuring DiffServ upon link failures or routing changes

```
inst oblig RoutingChangePolicy      {
subject ServiceManagementAgent;
on routeChanged (newPath);
do pdb = select_using_algorithmA(SLS_params[], newPath) ->
    /* A PDB suitable for the new path must be selected to cater for the service */
    pdb.enable() -> pdb_params[] = calculate(SLS_params[]) ->
    EventService.GenerateEvent (pdbObligationEvent, pdb_params[]); }
```

This policy instructs the ServiceManagementAgent to find a suitable PDB for the service with SLS parameters (SLS_parameters[]) when the path of routers that transports the service packets has changed. Information about the new path is conveyed to the ServiceManagementAgent with the routeChanged event. This event could be triggered by a component which is responsible for detecting and reporting routing changes. The new PDB is selected by the selection algorithm A. As in Example 4.2, the selection of the PDB can be adapted by replacing the existing policy with one using a different selection algorithm.

Policy to reflect changes in application or user requirements

The user/application may request different QoS guarantees at run-time by updating SLS parameters. As a consequence, network policy attributes must be changed to support the new user/application requirements. A policy example, which enables the ServiceManagementAgent to provide this type of service adaptation is given below.

Example 4.4 Policy for re-configuring DiffServ when SLS parameters change at run-time

```
inst oblig SLSRenegotiationPolicy {
subject ServiceManagementAgent;
on SLS_Request (new_SLS_parameters[], service_id);
do pdb_policy = policyService.lookup (service_id) ->
    new_pdb_policy_parameters[]=calculate ( pbd_policy,new_SLS_parameters[]) ->
    EventService.GenerateEvent (pdb_policyObligationEvent,
                                new_pdb_policy_parameters[]);}
```

In this policy example, the event SLS_Request carries both the new SLS parameters that the application/user requires and a unique identifier of the client application that requires

its SLS renegotiation (this identifier could be the Flow Description parameter [D. Goderis et al. 2001] of the Tequila SLS). The PDB policy reference that is responsible for this specific service is obtained via a lookup() operation on the Policy service, assuming that a table containing the service identifiers and their PDBs is updated when the initial request for SLS to PDB mapping has been succesful. Alternatively, the PDB that will guarantee the new service requirements could be selected at run-time among the set of implemented PDBs, as in the SLSMappingPolicy in the Example 4.2.

4.5 Implementation of a Prototype Adaptive Management System

As a proof of concept, we have implemented the enforcement architecture for policy adaptation, which we presented in section 4.3, where network policy is specified for Differentiated Services networks. The higher-level Service Policy Management Agent (Service PMA) and the lower-level Network PMAs are implemented using the Ponder toolkit [Damianou et al. 2002]. Ponder service management and DiffServ network policies are distributed to their relevant policy management agents using the policy deployment model implemented within the Ponder toolkit. The Service PMA implements one or more selection algorithms which query a simple policy database to choose a suitable DiffServ network policy. We have implemented a simple graphical tool, which the administrator uses to select which DiffServ network policies should be included in the policy database and to edit the QoS attributes of each selected policy (see Table 4.1). Figure 4.7 presents a usage example of the graphical tool. New policies can be included in the policy description database at run-time or existing policies can be removed. The administrator can also load new selection algorithms in the Service PMA. This allows new adaptation strategies to be implemented within the management system at run-time by adding new service management policies which use the new selection algorithms.

The applicability of the enforcement architecture for the adaptive management of DiffServ networks has been tested on simulated DiffServ networks, using the J-Sim [DRCL J-Sim] network simulator, which offers DiffServ functionality. We have extended the J-Sim simulator with new components, which allow the communication between the simulator and the Policy Management Agents. These components and their interaction with the Ponder Policy Management Agents are presented in Figure 4.5.

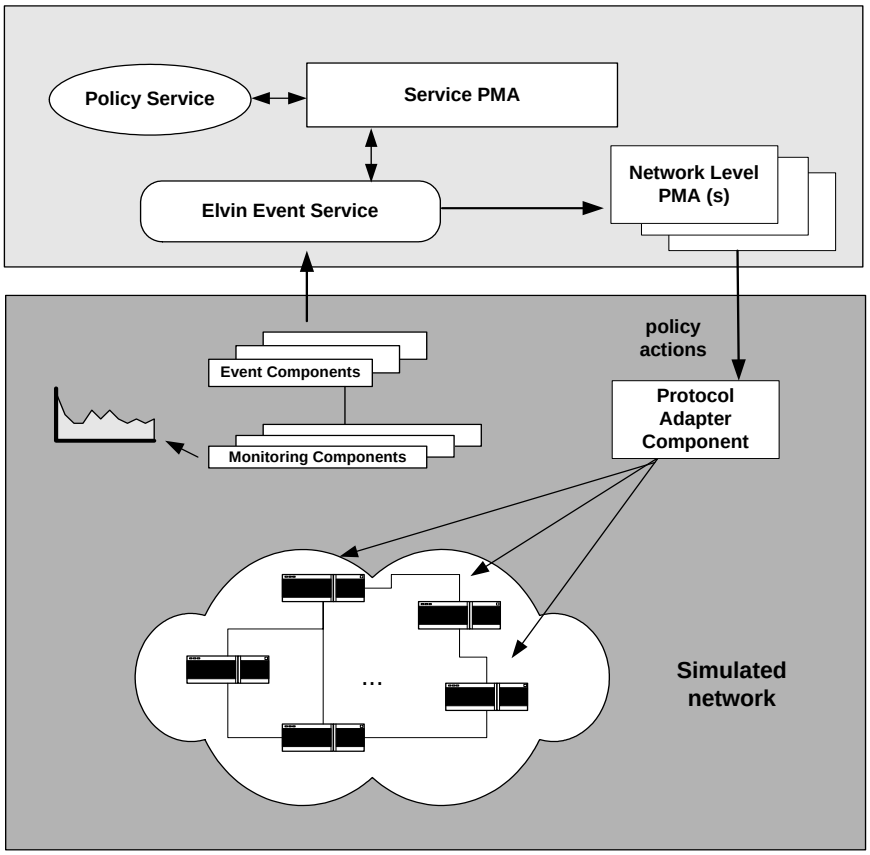


Figure 4.5 Implementation of an adaptive policy based management system for simulated DiffServ networks

For our needs, we have extended the J-Sim simulator with Monitoring, Event and Proxy PMA components, as shown in Figure 4.5. Monitoring components measure the performance characteristics such as throughput, packet loss and end-to-end delay for DiffServ traffic classes during a simulation. These measurements are used to display performance graphs and by Event components to generate performance degradation events when traffic measurements exceed specific thresholds. For example, an event indicating packet loss will be created when the packet loss of a specific class of traffic exceeds a configured threshold. Simple rules are used to configure the thresholds upon which Event components should generate events.

In addition, Event components generate events indicating network failures or new user/application QoS requirements. For example, when a host inside the simulated network requests certain QoS for a new traffic flow, a QoS request event will be generated. All events are dispatched to the management system, where they trigger policies, through the Elvin Event Service [Segall et al. 1997], a publish/subscribe

messaging system, which we are currently using within the Ponder Toolkit. The policies select lower-level network policies to enable/disable or trigger corrective actions on the managed devices.

Ponder Policy Management Agents and the J-Sim simulator run on different hosts. We have extended the simulator with Proxy PMA components, which implement an interface to the simulator. Each Proxy PMA represents a Network-Level Ponder PMA. Policy actions are communicated from the Ponder PMAs to the simulated nodes through the corresponding Proxy PMA components, sending TCP messages. This communication uses our proprietary text-based protocol. Proxy PMAs then translate the messages into commands that belong to the management interface of the target simulation nodes. Note that our policy based management system abstracts the protocols used for communication between a management agent and the managed device. Policy actions could be enforced by the Ponder management agents using standard policy protocols such as COPS or SNMP which would be supported by real network elements.

A graphical tool [Husein 2003][2] has been implemented using the policy-based extensions to the J-Sim simulator. This tool allows the administrator to create a DiffServ network, configure the DiffServ mechanisms of each node and visualise and configure the policy-based components inside the simulator. Figure 4.6 presents a usage example of the graphical tool. The execution of policy actions during the simulation is animated with growing arrows from the Proxy PMA component to the policy's targets. The Proxy PMA component is the graphical representation of the Protocol Adapter Component, shown in Figure 4.6, and communicates with the Ponder Policy Management Agents to receive the policy actions to be executed upon a particular event. Events during the simulation are also animated with growing arrows from the Event Service component to the Proxy PMA component.

We present results of an experiment based on a simulated DiffServ network, with the topology indicated in Figure 4.6. Traffic enters the DiffServ network through the edge router "Edge" and exits the network from the edge router "n4". Nodes h3 to h7 are constant bit rate traffic sources in this expreriment (node h[i] sending i–2 Mbps of traffic) while nodes h8 to h12 receive the transmitted traffic according to the following formula: host [i+5] receives traffic from source node host[i].

[2] This Msc thesis was supervised by me. The policy based extensions to the simulator and the protocol for communicating with the Ponder toolkit have been implemented by me, as well as the design of the GUI. M. Husein has implemented the GUI and the animation modules.

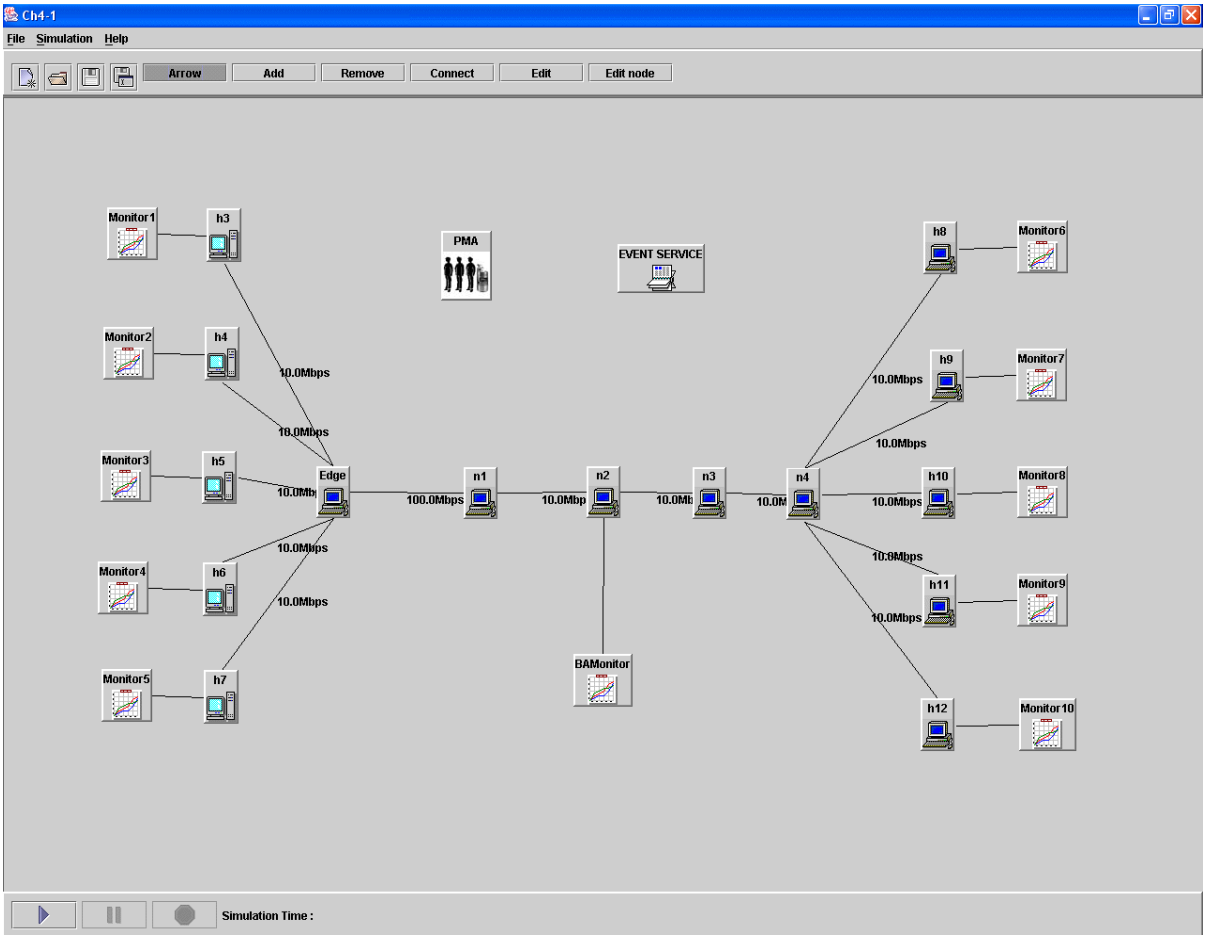


Figure 4.6 Simulated DiffServ network

We implemented three PDB policies within this network, the *GreenPolicy PDB* with the EF PHB, the *YellowPolicy PDB* with the AF11 PHB and the *RedPolicy PDB* with the BE PHB. We used priority scheduling at the egress interface of each core router (core routers are the nodes n1, n2 and n3 in Figure 4.5). The graphical tool shown in Figure 4.7, was used to select which network policies should be included in the policy database, and to edit the QoS attributes for each PDB. The tool interfaces with a domain browser to select policies from the LDAP directory server, where policy objects are stored. The values assigned to QoS attributes represent the attribute upper thresholds, where -1 indicates there is no upper threshold.

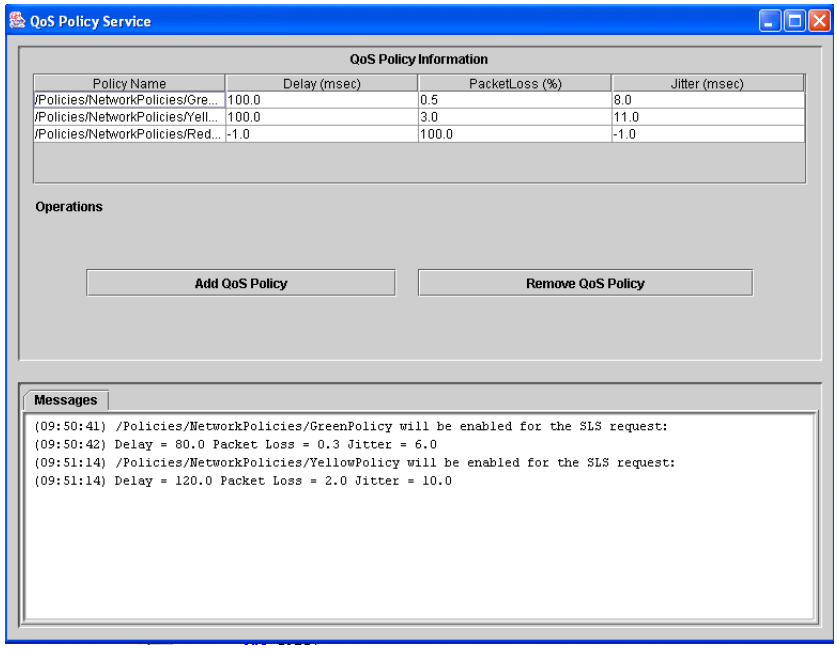


Figure 4.7 Graphical User Interface to the Policy Service

In this experiment, we show how our adaptive management system automatically performs the SLS to PDB mapping for new SLS requests, implemented with the policy *SLSMappingPolicy* from Example 4.2. The event's *SLSRequest* parameters are the address of the node issuing the SLS request and the requested values of delay, packet loss and jitter with the selection algorithm choosing the closest PDB according to the triple <delay, packetLoss, jitter>. Figure 4.8 indicates the throughput of each individual flow that traverses the DiffServ network, whilst Figure 4.9 displays the throughput of the DiffServ aggregates, as measuring at the egress interface of the core router n2 inside the DiffServ network.

At first, node h3 sends 1Mbps of EF traffic, node h4 sends 2Mbps of AF11 traffic and mode h5 sends 3Mbps of BE traffic. At point A in Figure 4.8 and Figure 4.9, node h6 issues a SLS request for a service of 4Mbps with delay < 80ms, packet loss < 0.3% and jitter < 6ms. This SLS request is handled by our management system to adapt configuration of the DiffServ network. The ServicePMA chooses the *GreenService* and h6 starts receiving the EF PHB. At point B, node h7 issues a SLS request for a 5Mbps service which guarantees delay < 120.0ms, packet loss < 2% and jitter < 10ms. This time the ServicePMA chooses the *YellowService* for this SLS request and triggers the *YellowPolicy*. As a result of the enforcement of the *YellowPolicy*, the DiffServ network adapts configuration to guarantee the AF11 PHB to the traffic originating from node h7. We can observe that after this point the BE class suffers from starvation, since the total admitted EF traffic (5Mbps) and AF11 (7Mbps) traffic is greater than 10Mbps which is the maximum capacity of the core network.

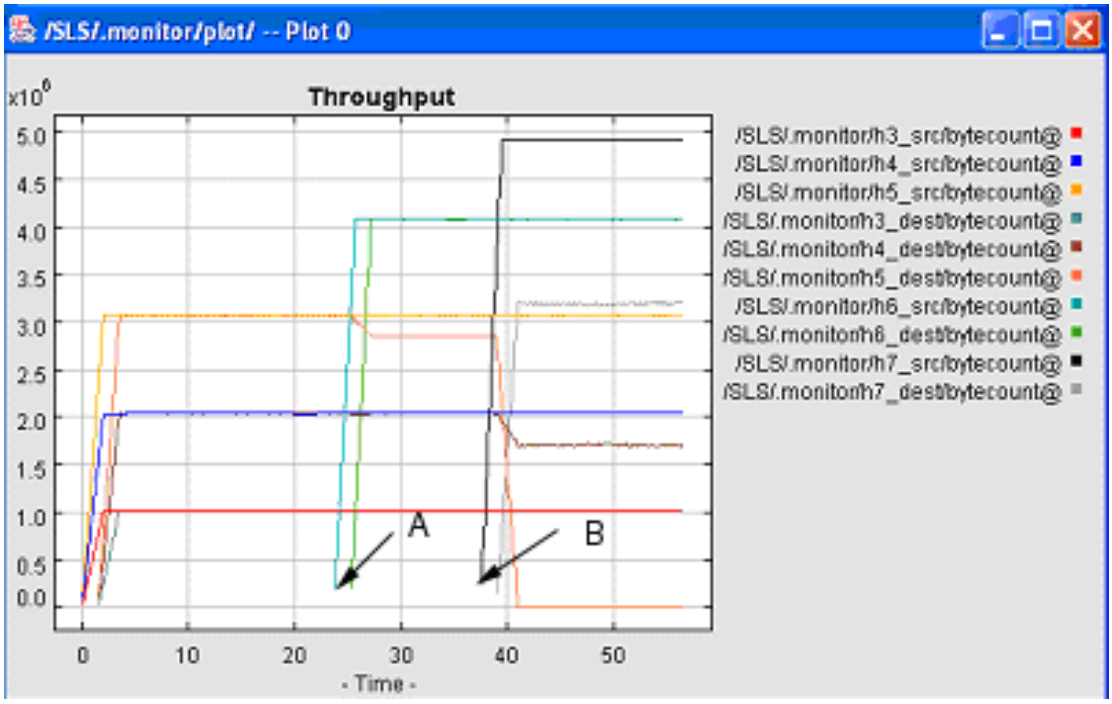


Figure 4.8 Throughput of individual traffic flows (the Y axis shows the bandwidth in bps per flow)

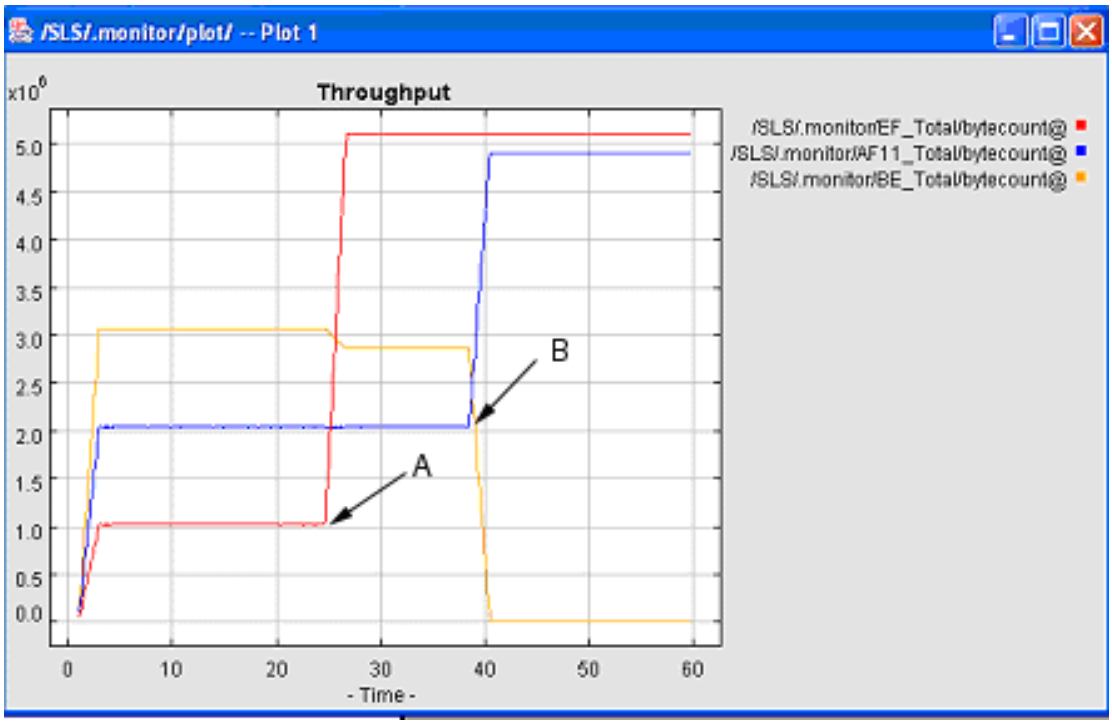


Figure 4.9 Throughput of DiffServ aggregates (the Y axis shows the bandwidth in bps per traffic aggregate)

In the following, we will present results from another experiment we conducted using the topology presented in Figure 4.6. For a second time, we implemented three PDB policies within the simulated network. The PDB policies along with their QoS guarantees are depicted in Figure 4.7. In this experiment, the *ServicePMA* implements an adaptation strategy for handling PDBs' performance degradation. The latter is expressed in terms of the increased loss rate of the packets that receive a particular PDB. This type of adaptation is implemented with the higher-level management policy *LossPolicy*. The specification of this policy is given in Example 4.5.

Example 4.5 Policy for handling increased packet loss rate of network services

```
inst oblig /Policies/ServiceManagementPolicies/LossPolicy {
subject /PolicyAgents/HighLevel/ServicePMA;
on HighLossEvent( DSCP, thresholdValue, inputRate);
do adjustLoss( DSCP, thresholdValue,maxInputRate);
target /Routers/EdgeRouters; }
```

A *HighPacketLoss* event is generated by the Event Components within the simulator when the packet loss rate of the network service identified by the DiffServ codepoint DSCP is above the maximum permitted value *thresholdValue*. The parameter *maxInputRate* carries the maximum arrival rate of the service traffic at the ingress interface of the edge routers.

The action *adjustLoss* reduces the number of packets being dropped by the policer at the ingress interfaces of the edge routers. Note that in this implementation of the action *adjustLoss*, we do not handle packet loss due to packets dropped at the queues of the network interfaces. More specifically, this action only reconfigures the metering elements of the edge routers so that more traffic that belongs to the service can be admitted to the DiffServ network. The new value of the traffic that can be admitted to the network is calculated according to the following formula:

$$\text{AdmittedTraffic} \geq (100.0 - \text{thresholdValue}) * \text{maxInputRate} / 100.0$$

For example, if the maximum arrival rate of the traffic at the edge of the network is 1Mbps and the maximum permitted value for packet loss rate is 10%, then the metering elements will be reconfigured such that a minimum traffic of 0.9Mbps can be admitted to the network. Figure 4.10 presents the adjustment of packet loss rate for the EF and the AF11 traffic classes in our simulated DiffServ network.

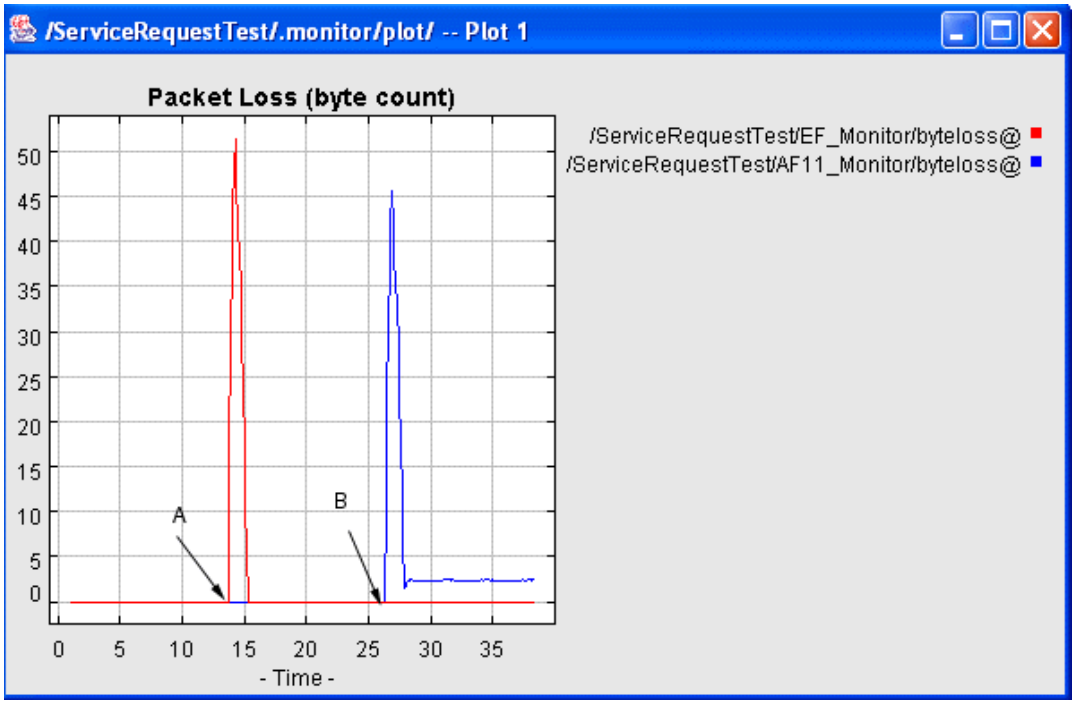


Figure 4.10 Adjustment of network services' packet loss rate

Initially, the edge router (node Edge in Figure 4.6) is configured to admit 2Mbps of EF and 3Mbps of AF11 traffic. In this configuration, host 3 sends 1Mbps of EF traffic and host 4 2Mbps of AF11 traffic. In consequence, the packet loss rate of the EF and AF11 traffic classes is zero. However, at point A, the user at host 6 starts sending 4Mbps of EF traffic. This causes an increase of the EF packet loss rate, since the network was a priori configured to admit a maximum of 2Mbps of EF traffic. This increase in packet loss for the EF PHB is automatically adjusted by the *ServicePMA*, configuring the edge router to accept more EF traffic (the metering element at the edge router is configured to accept EF traffic at a minimum rate of (100-0.5)%*5Mbps, where 5Mbps is the new sum of EF traffic entering the network) so that the packet loss rate of the EF traffic drops below its maximum permitted value, which is 0.5% (see Figure 4.7). Later, at point B, host 7 starts sending 5Mbps of AF11 traffic. This time, the *ServicePMA* adjusts the packet loss rate of the AF11 traffic to drop below its maximum permitted value of 3% by instructing the edge router to accept AF11 traffic at a minimum rate of (100-3)%*6Mbps, where 6Mbps is the rate of the AF11 traffic entering the network.

We presented two experiments showing how our prototype adaptive management system can be used to deploy service management policies within various topologies of simulated DiffServ networks. A number of different adaptation scenarios can be easily implemented and tested within the simulator, eg. adaptation strategies according to changes in routing

or in link failures. Adaptation strategies could also be deployed and tested for mobile networks that can be simulated within J-Sim, supporting simulation of mobile networks. Our prototype provides the policy administrator the ability to easily experiment with his adaptation strategies before deploying them into real network environments.

4.6 Discussion

In this chapter, we presented how policy adaptation can be realised within the Ponder policy framework. Adaptation is provided in one of the following ways: a) by dynamically changing the parameters of a network policy to specify new attribute values for the run-time configuration of managed objects; and, b) by selecting and enabling/disabling a policy from a set of pre-defined network policies at run-time. The parameters of the selected network policy are calculated and set at run-time.

We presented examples that demonstrate how higher-level management policies cater for the dynamic management of services in a Differentiated Services network. We also presented a prototype implementation of an adaptive management system for Differentiated Services networks, where network devices are simulated with the J-Sim network simulator.

It is important to note that our ideas for policy adaptation are not restricted to some specific management area. Rather, they can be used to implement adaptive management systems for management of different networking mechanisms (e.g. QoS, security) and for several aspects of distributed systems (e.g. distributed ubiquitous systems management). Appendix A will present how our ideas could be used for dynamically adapting a network's access control policies and for dynamically adapting policy in a ubiquitous environment based on changes in user's location. Note that we have not actually implemented and tested the policies shown in the appendix. However, the fundamental idea of our framework is that higher-level policies can be deployed to dynamically select which lower-level management policy should be enforced when changes occur within the managed environment. The selection is based on a user-defined selection algorithm, which can also be dynamically changed as a means to implement new adaptation strategies. What is required is to model the behaviour of each low-level policy in a common set of attributes. For example, in our prototype implementation, each DiffServ policy is associated with the triple < delay, packet loss, jitter>, which indicates the behaviour of the policy in terms of the QoS it guarantees. In a general case, a low-level management policy will be associated with the tuple <a1, …, aN>, expressing the

behaviour of the policy on the managed system. Then, different selection algorithms can be used by higher-level policies to select which of the underlying policies will cater for a specific request.

Since knowledge of the behaviour of low-level policies is required in order to deploy an adaptive management system, as a future direction for our work, learning techniques (like the one presented in [Uttamchandani et al. 2003]) can be investigated to see how the behaviour of the low-level polices can be determined based on monitoring and feedback information, when the behaviour of the rules is not static. For example, the behaviour of a policy in terms of its QoS guarantees, expressed with the triple < delay, packet loss, jitter >, may vary according to the traffic entering the network. In this context, a learning technique could be used to determine the function that returns the QoS guarantees of a policy having as parameter the current network traffic.

One issue that needs to be addressed here is the problems that may arise using an adaptive management system rather than manual intervention to adapt network configuration. Possible problems are that the management system may oscillate or fail to deliver a feasible network configuration. For the first problem, we give the network administrator the ability to configure the *Event Components* of the adaptive management system, to filter out frequent events of the same type and to use thresholds to ensure that small changes in performance do not generate events to trigger configuration changes. Badly programmed policies could possibly introduce instability, as is the case with any incorrect program. We give the administrator the ability to use our prototype implementation of an adaptive policy based management system in order to test his adaptation strategies within the simulator and thus determine whether his policies will cause any problems in a real network. Furthermore, in order to deliver a feasible network configuration, a validation process should be implemented within an adaptive policy based management framework. This process will ensure that the policy is consistent with the functional or resource constraints within the target environment. Chapter 5 will give details on how policy validation can be realised within the Ponder policy framework.

Another important issue that needs to be addressed is to enhance the functionality of the adaptive management system to initiate corrective actions when the causes of the violations are not known a priori. Currently, its task is to adapt the set of underlying network policies upon pre-defined conditions. This implies that the administrator knows a priori the causes of violations and consequently can enforce the corresponding correcting actions when a particular violation is detected. However, corrective measures should be undertaken to remedy any causes of violations in the managed environment. This will

require the management system to perform root cause analysis, ie. to discover what is really causing the violation, in order to initiate the corresponding corrective actions when the violation is detected.

Chapter 5

Policy Validation within the Ponder Framework

Network management policies are deployed within the network to configure the mechanisms of the devices that implement the policies. In the policy-based management approach, the specification and deployment models of policies must be device-independent, i.e. independent of the type of the device and the specific implementation of the network mechanisms that vendors have chosen for their products. For this purpose, as we have previously discussed in this thesis, several modelling frameworks have been proposed by standards organisations to represent the abstractions of the networking mechanisms that devices implement. The two most important standard organisations focusing on network management, IETF and DMTF, both use object-oriented models to represent the different available network mechanisms.

Figure 5.1 presents how CIM Network Common Model [DMTF 2004] represents the mechanisms of the DiffServ architecture. Other models to represent the individual functional elements which a DiffServ-enabled device supports are described in IETF's DiffServ router model [Bernet et al. 2002] and IETF's Quality of Service Policy Information Model (QPIM) [Fine et al. 2001].

As we can see in Figure 5.1, the DiffServ mechanisms are represented as objects whose classes are defined in the model and have specific attributes, as defined in each class (the CIM attributes are not shown in diagram). For example, the mechanism to mark packets with a particular DiffServ Code Point (DSCP) is represented as an object of the class DSCPMarkerService and contains the attribute DSCPValue, which holds the value of the DSCP to assign to the associated packets. An example IETF policy rule using the QPIM model for configuring the marking mechanism of a DiffServ node could be the following:

If (sourceIP = 10.0.1.2 AND destinationIP = 10.0.2.3) then DSCP=0x43

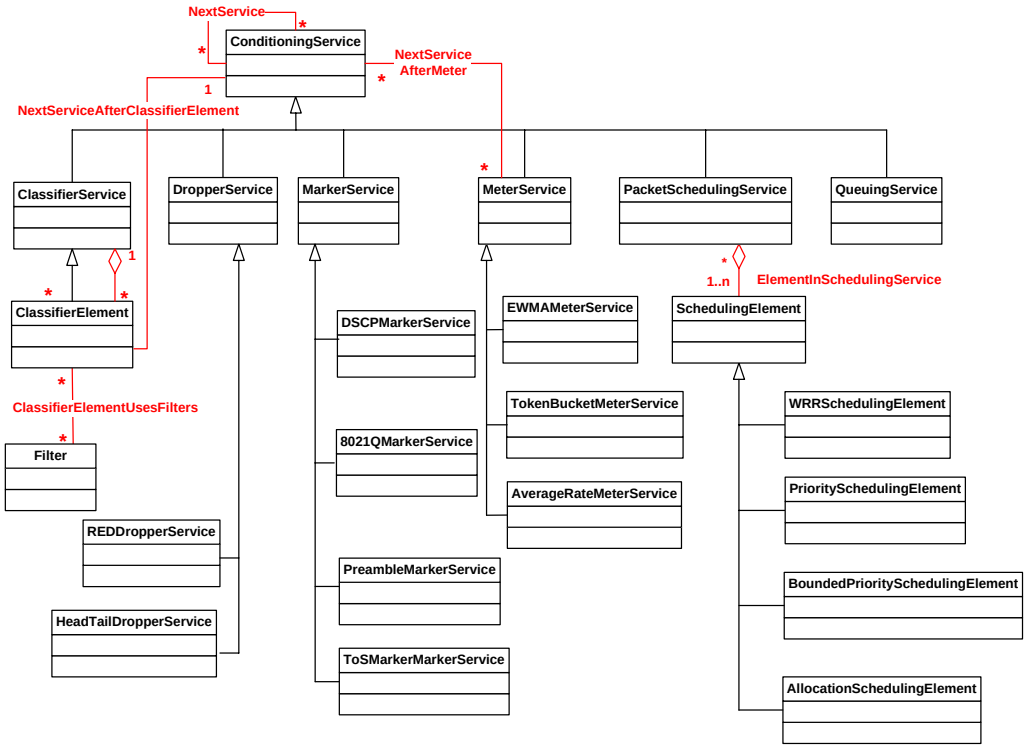


Figure 5.1 DiffServ classes in CIM Network Common Model

The variables sourceIP and destinationIP are defined in QPIM. The semantics of the DSCP assignment is encapsulated in the pairing of a DSCP variable and a DSCP value within a single SimplePolicyAction instance defined in PCIM [Moore et al. 2001] via the appropriate associations. The same semantics are rendered with the following Ponder rule, which uses the CIM representation of DiffServ mechanisms. In the Ponder rule, two CIM objects are created: a ClassifierElement object which holds the information (sourceIP and destinationIP) of the flow to be marked and a DSCPMarkerService object that will actually mark the specified flow to the DSCP with value 0x43, carried by the event MarkingRequest.

Example 5.1 Policy to configure the marker element of a DiffServ meter

```
inst oblig /Policies/MarkingPolicy {
subject  /PolicyAgents/NetworkPMA;
target t = /Routers/EdgeRouters;
on MarkingRequest (sourceIP, destinationIP, dscp);
do Filter filter = new Filter ( DSCP);
    ClassifierElement cls = new ClassifierElement();
    cls.addFilter(filter)->
    DSCPMarker dscp_marker = new DCSPMarker (dscp) ->
    cls.setNextService (dscp_marker)->
    t.addDiffServElement (cls);
    t.addDiffServElement (dscp_marker); }
```

The enforcement of both policy rules will result in the execution of the policy action(s), which involve configuring the corresponding DiffServ mechanisms of the target devices. However, as we have discussed in the introduction of this thesis, it is necessary to ensure that the network elements have the capability to implement the policy. Policy validation is the process that will check that the device to which the policies apply support the required functionality i.e. the policy invokes an operation that is actually implemented by the device or the device has the required resources needed to satisfy the policy action. Validating policies at compile or configuration time will prevent the management overhead and the potential problems that arise when actually trying to enforce policies that are not feasible in the given network environment.

As a first step for analysing the process of validation of network policies, we can observe that the enforcement of a network-level management policy results in the creation and/or configuration of the functional elements which represent the various mechanisms within the devices to which the policy applies. Consider a network element that implements various functional elements to support DiffServ e.g. classifiers, meters, schedulers, markers, etc.

In this context, a policy is valid in respect to the target device's capabilities when the target device is able to:

a) Create the requested new functional element and/or
b) Configure the new or an existing functional element with the requested values.

In the following sections, we will provide details of each case of validation and we will outline how policy validation can be implemented within the Ponder policy framework using CIM as the model for representing the policy target's mechanisms and capabilities, using Differentiated Services as an example of a managed networking environment. Similar to the reasons for using the CIM model for policy specification and enforcement in our framework, the rationale for using the CIM model for validating policies is that it provides a standard device independent information model for the network elements which are potential targets for policies. Policies can be specified with respect to generic CIM target objects and so can be used for many different network elements. Furthermore, mapping of CIM target objects to the mechanisms which are actually implemented by the specific device is independent of the communication protocol between a CIM-based implementation and the managed devices.

5.1 Policy Validation with Respect to the Ability of the Target Devices to Create Functional Elements

This validation requires checking whether the target device is capable of performing an "add" operation for a DiffServ functional element, i.e.

i) Does the device support the requested DiffServ functional element type?
ii) Does the device have enough resources to create (add) the new functional element?

For the first case, consider the following policy that the administrator specifies for providing traffic conditioning on a set of edge routers.

Example 5.2 Network policy rule for creating DiffServ functional elements within the target routers

```
inst oblig /Policies/TrafficConditioningOnEdge {
subject /PolicyAgents/NetworkPMA;
target t = /Routers/EdgeRouters;
on TrafficConditioningRequest ( meter_rate, dscp);
do AverageRateMeter avg_meter = new AverageRateMeter (meter_rate) ->
    t.addDiffServElement (avg_meter) ->
    DSCPMarker dscp_marker = new DCSPMarker (dscp) ->
    t.addDiffServElement (dscp_marker); }
```

The policy TrafficConditioningOnEdge instructs the target edge routers to create two functional elements: a meter of type AverageRateMeter and a marker of type

DSCPMarker. This policy is valid only if the target devices support the DiffServ architecture and in particular the DiffServ metering and marking mechanisms represented by the meter type AverageRateMeter and the marker type DSCPMarker respectively. In order to perform this type of validation, we must know whether the device supports DiffServ and if so, the different types of DiffServ functional elements supported. A solution to this would be to query the CIM representation of the device which can indicate whether the device supports DiffServ and which types of functional elements a DiffServ-enabled device implements.

In the general case this type of policy validation can be implemented within the Ponder framework by using meta-policies as a means to specify the constraints that network policies must satisfy with respect to the DiffServ mechanisms that the targets support. The use of meta-policies allows the policies to be checked at specification time, before deployment, which is clearly an advantage over checking at policy execution time. Meta-policies defined in Ponder specify constraints over a set of policies, with respect to the permitted types of policies or the elements within the policies. These constraints apply to policies within a specific scope and effectively limit the permitted policies in the system. The syntax of a meta-policy is based on the OMG Object Constraint Language (OCL) [OMG 1999]. The body of the meta-policy specifies the constraint as a series of OCL semicolon separated expressions which can be boolean or navigation expressions. If any of the boolean expressions evaluates to true, execution stops and the action following the raises-clause is executed. Example 5.3 shows a meta-policy specifying constraints over a set of network policies with respect to whether DiffServ is supported by the policy targets.

Example 5.3 Meta-policy for specifying constraints for DiffServ policies

```
inst meta DiffServNotSupported raises
  DiffServNotSupportedException(invalid_policies) {
 [invalid_policies] = this.policies ->
  select  (p |  p.action ->
     exists (a | a.name = “addDiffServElement”   and p.target ->
        exists(t | t.notSupports(“DiffServ”))));
invalid_policies -> notEmpty; }
```

The body of the meta-policy contains two OCL expressions. The first one selects all policies (p) with the following characteristics: the action set of p contains at least one action named “addDiffServElement”, and DiffServ is not supported by at least one of the

policy's target devices. The action notSupports on the target device is a look-up operation on the CIM representation of the device which returns true if the device does not support the DiffServ architecture. The second OCL expression returns true if the variable invalid_policies, which is returned from the first OCL exception is not empty. If the result of the last expression is true, the DiffServNotSupportedException specified in the raises-clause is executed with the invalid_policies set as a parameter.

Example 5.4 shows another meta-policy which refines the constraints over a set of network policies. Constraints are specified with respect to the specific DiffServ mechanisms supported by policy targets. Note that similar meta-policies can be used for specifying validation constraints for any underlying technology.

Example 5.4 Meta-policy for specifying constraints for DiffServ mechanisms

```
inst meta notSupportedElement raises
  notSupportedElementException(invalid_policies) {
 [invalid_policies] = this.policies ->
  select  (p |  p.action ->
     exists (a | a.name = "addDiffServElement"   and p.target ->
        exists(t | t.notSupports(a.parameter.oclType))));
invalid_policies -> notEmpty; }
```

The first OCL expression selects all policies (p) with the following characteristics: the action set of p contains at least one action named "addDiffServElement", whose parameter type (i.e. the type of the DiffServ element that p wants to add to each of the target devices) is not supported by at least one of the policy's target devices. Note that we use the OCL method oclType to obtain the type of the "addDiffServElement" action parameter. An obligation policy could be triggered by the exception notSupportedElementException to perform corrective actions to resolve invalid policies. An example would be to install missing DiffServ mechanisms in a programmable router, as shown in Example 5.5. The exception notSupportedElementException from Example 5.4 triggers the obligation policy PolicyForInvalidDiffServPolicies which installs the missing DiffServ mechanisms in the relevant programmable router.

Example 5.5 Policy rule for installing the non supported DiffServ mechanisms in the appropriate programmable routers

```
inst oblig PolicyForInvalidDiffServPolicies {
subject /PolicyAgents/NetworkPMA;
on notSupportedElementException (Policy p[])
/* p is the set of policies that the meta policy in Example 5.4 has returned as not valid*/
do
 foreach pol in [p] {
      foreach action in [pol.actions]
        foreach t in [pol.target] {
         if (action.name = “addDiffServElement” and
            t.notSupports( (DiffServ_Class) action.parameter))
            then  t.installMechanism  ((DiffServ_Class) action.parameter)); }}}
```

The policy rule PolicyForInvalidDiffServPolicies receives the set of invalid network policies with the event notSupportedElementException, which the meta-policy in Example 5.4 raises. The pseudocode following the do statement could be implemented as a script policy action which finds the pairs < target_device, non-supported mechanism > and installs the non-supported mechanisms in the appropriate target devices using the method installMechanism.

As mentioned earlier, a policy is not valid when the device does not have the resources to create the requested functional element. As an example, consider a DiffServ device that can only support a limited number of traffic classes. A policy that tries to create a new traffic class within the device will fail if the maximum limit is exceeded. This case of policy validation can be implemented using CIM as a means to specify the necessary information about the capabilities and the current state of the device – the maximum number of traffic classes and the current number of classes respectively. Since the current CIM version does not include DiffServ-specific capabilities and state information, we are proposing a “DiffServ-metrics” CIM extension to provide this information, which we will present in detail in section 5.3. However, unlike the previous case, we cannot decide offline whether the policy is valid by checking the CIM representation of the device, since the decision depends on device’s current state. This means that the decision must be made at the time the policy is to be enforced, which implies that the conditions under which the policy is valid must be specified as constraints within the policy specification, as indicated in the following example. The policy rule PolicyToAddTrafficClass, in Example 5.6, will only add a new traffic class when the current number of classes is less than the maximum.

Example 5.6 Policy rule for creating a new traffic class when the target device supports a limited number of traffic classes

```
inst oblig /Policies/PolicyToAddTrafficClass {
subject  /PolicyAgents/NetworkPMA;
target t = /Routers/CoreRouters;
on addTrafficClassRequest (classOfTrafficParams[]);
do addTrafficClass (( classOfTrafficParams [] );
/* The attributes CurrentClassesOfTraffic and MaximumClassesOfTraffic are defined
in our proposed extension to CIM, see Figure 5.2 */
when t.CIM_Get (CurrentClassesOfTraffic) <t.CIM_Get (MaximumClassesOfTraffic); }
```

5.2 Policy Validation with Respect to the Permitted Values of Variables within the Device

In the second case of policy validation, checking is necessary to ensure that policies can set variable values within the device only if the requested values fall within permitted ranges with respect to the device capabilities. We can distinguish between variables with static bounds and variables with dynamic bounds.

- Variables with static bounds. As an example, the number of classes of traffic a device supports is a variable that has a specific upper bound. A policy rule that attempts to set this variable greater than its upper bound is invalid. We can detect offline if the policy is valid, if we check the CIM representation of the device, which includes the capability information of the device (i.e. the maximum classes of traffic the device supports).

- Variables with dynamic bounds. In this case, the bounds of the variable to be set are not static, but they depend on the state of the device. As an example, the maximum bandwidth that can be allocated to a particular class of traffic cannot exceed the available free bandwidth that the device can offer, which is calculated as: available_free_bw = total_output_bw – current_allocated_bw (for all traffic classes)

Assume that CIM can provide the necessary information about the capabilities and the current state of the device: the total output bandwidth and current allocated bandwidth to all traffic classes respectively. In this case, we cannot decide offline whether the policy is

valid so the validity must be specified as constraints in the policy specification, which is similar to the policy in Example 5.6 so is not shown here.

To summarise, a CIM based model for representing network devices, can provide two types of management information:

i. Information about device's capabilities (e.g. type of mechanisms it supports, bounds on resources, bounds on specific objects' attributes)
ii. Information about the current state of the device (e.g. current resource allocation, current values of specific objects' attributes)

We can use this information to support static policy validation with respect to device capabilities, which can be performed offline using meta-policies as a means to specify the constraints on the permitted DiffServ mechanisms, resources and bounds on object attributes. The required information about the capabilities of the devices can be obtained from CIM models. Dynamic policy validation with respect to current state of the device can only be performed at the time the policy is to be enforced by means of constraints which are specified as part of the policy rules to define the conditions under which the policy is valid. These constraints can reference information extracted from a corresponding CIM model.

5.3 Implementation of a Policy Validation Architecture

As we have discussed, for the purpose of validating policy against device and network capabilities, it is necessary to extract several parameters such as the number of QoS classes, the bandwidth allocated to each class, the QoS mechanisms that the managed routers support, etc. CIM provides a generic model for representing DiffServ-enabled managed devices. In particular, as we have presented in Chapter 3 of this thesis, the CIM Network Common Model defines the class QoSService, which is a subclass of the generic class NetworkService. The QoSService is an aggregation of instances of the ConditioningService class (see Figure 5.1), whose subclasses define the DiffServ mechanisms in a DiffServ router. These mechanisms are represented by the following CIM classes: ClassifierService, MeterService, MarkerService, DropperService, DropThresholdService and PacketSchedulingService.

Our implementation uses the CIM Network Common Model for representing the DiffServ elements that a router supports. In addition, since we need statistics related to DiffServ,

e.g. number of implemented traffic classes, bandwidth allocated to each class of traffic, etc. we have designed a "DiffServ metrics" extension to the current CIM model which can provide the management system with this information. The proposed "DiffServ metrics" CIM sub-model is presented in Figure 5.2.

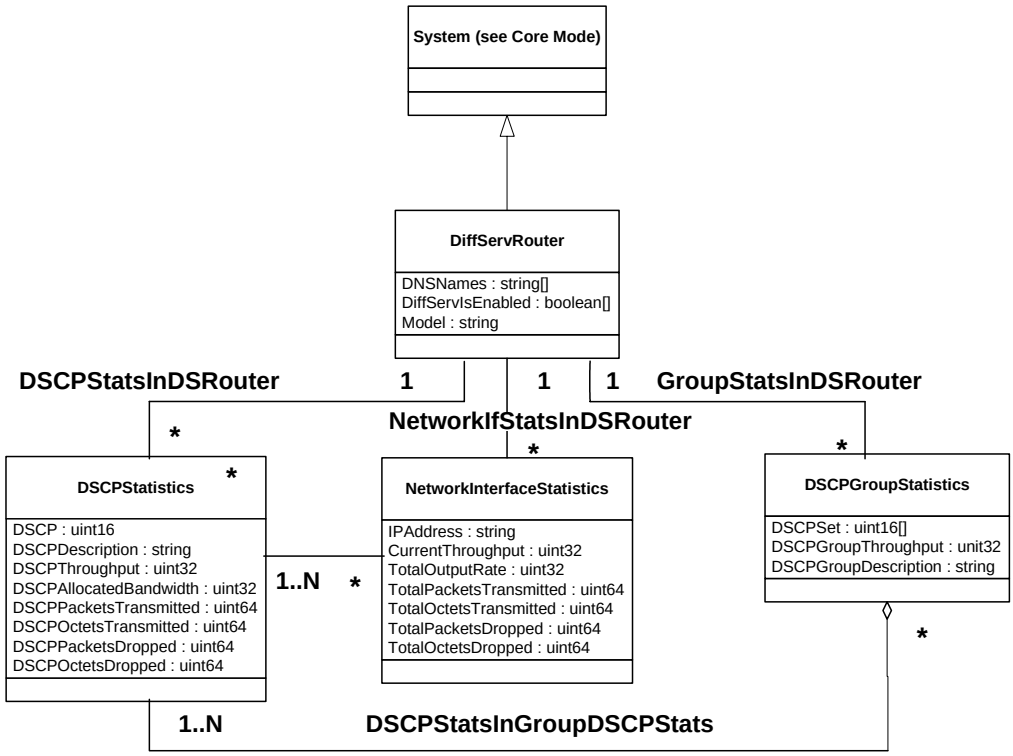


Figure 5.2 UML Diagram of the DiffServ-metrics CIM extension

The CIM class DiffServRouter is the abstraction of a DiffServ-enabled router and derives from the System class in the Core Model and focuses on representing a system that is DiffServ-enabled. The classes DSCPStatistics, NetworkInterfaceStatistics and GroupDSCPStatistics derive from the CIM class StatisticalData in the Core Model (this inheritance is not shown in Figure 5.2?? incorrect number), which is defined as the root class for any arbitrary collection of statistical data and/or metrics applicable to one or more ManagedElements.

In the spirit of CIM, the association classes DSCPStatsInNetworkIfStats and DSCPStatsInGroupDSCPStats derive from the association class RelatedStatisticalData in the CIM Core Model. The extension classes DSCPStatistics, NetworkInterfaceStatistics and GroupDSCPStatistics derive from StatisticalData. Therefore, their associations (e.g. DSCPGroupStatsInDSRouter) with the DiffServRouter class, which derives from the System class (which in turn derives from the ManagedElement class) derive from CIM's

ElementStatisticalData association, which links StatisticalData classes to ManagedElement classes.

The NetworkInterfaceStatistics class provides traffic statistics for every network interface card that belongs to the router. The DSCPStatistics class caters for statistics per implemented DSCP. A DSCPStatistics instance is associated with one or more NetworkInterfaceStatistics instances, as a particular DSCP may be implemented in more than one network interface on a single router. Finally, the GroupDSCPStatistics class is an aggregation of DSCPStatistics classes and provides statistics for a group of DSCPs which together define a class of service offered to the corresponding traffic aggregates. For example, the "Gold Group" in a router may be constructed from the DSCP offering the Expedited Forwarding per hop behaviour (EF PHB) [Jacobson et al. 1999], while the "Silver Group" may be constructed from the DSCPs offering the Assured Forwarding per hop behaviour (AF PHB) [Heinanen et al. 1999]. A DSCPStatistics instance may belong to more than one GroupDSCPStatistics instances.

The architecture for validating Ponder policies with respect to the CIM-represented DiffServ mechanisms and capabilities is conceptually the same as the architecture for policy enforcement of DiffServ-specific policies, presented in section 3.2 of this thesis. Figure 5.3 presents the architecture for policy validation. The functionality of PMAs and Target Policy Objects, defined in [Dulay et al. 2001], is extended with the capability to act as CIM clients to allow these components to query CIM objects. This allows the Ponder components to retrieve variables that belong to CIM classes as a means to evaluate policy constraints or metapolicies that contain such variables. The communication is realised through the CIM Object Manager (CIMOM) from the WBEM Services project [WBEM Services Project 2003].

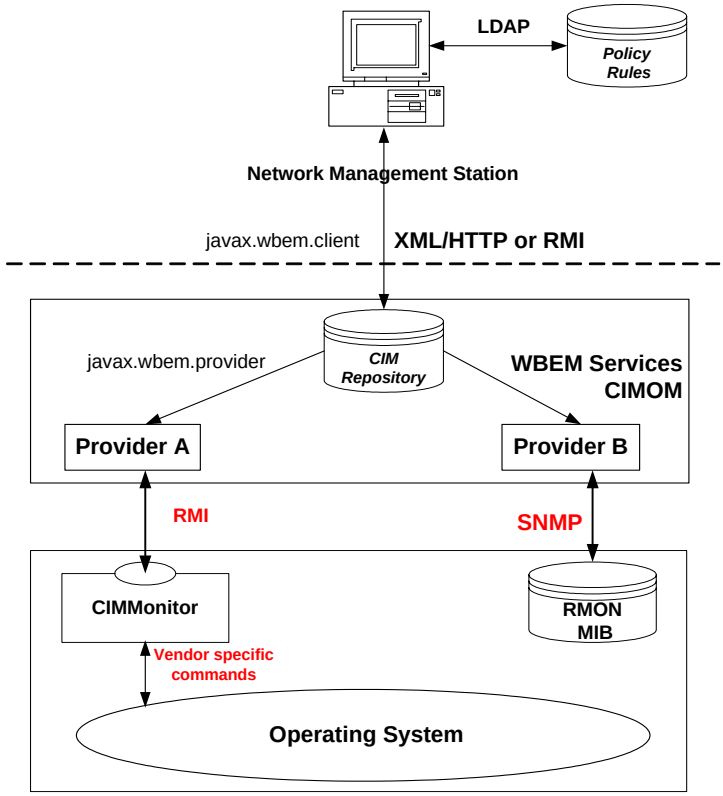


Figure 5.3 Architecture of Policy Validation System

We have implemented a Provider specific for the Linux Operating System to handle the DiffServ metrics sub-model classes presented in Figure 5.2 (shown as provider A in Figure 5.3). This Provider communicates through the Java RMI interface with the CIMMonitor component to get DiffServ variables from the Linux operating system by issuing traffic control statistics commands ("tc –s commands"). For example, in order to obtain the CIM variable CurrentClassesOfTraffic (from CIM class DiffServRouter), Provider A issues a request to the CIMMonitor, which in turn queries the current "tc" configuration of the device to get the current number of implemented DiffServ classes. An alternative for retrieving DiffServ traffic statistics could be to use SNMP to get variables from a DiffServ RMON probe, indicated by Provider B in Figure 5.3, which could translate CIM attributes to SNMP get requests for an RMON MIB. However, there is not yet an implementation of an RMON probe for DiffServ on Linux, so this has not been implemented.

We will now provide an example demonstrating the usage of our implementation. The purpose of this example is to enforce the policy validation rule presented in Example 5.6.

Example 5.6 Policy rule for creating a new traffic class when the target device can provide the requested bandwidth for the new class

```
inst oblig /Policies/PolicyToAddTrafficClass {
subject /PolicyAgents/NetworkPMA;
target t = /Routers/CoreRouters/Athena;
on addTrafficClassRequest (interface, dscp,  bandwidth);
do t.addTrafficClass (interface, dscp, bandwidth);
when t.CIM_GetThroughput(interface) + bandwidth
      <  t.CIM_GetTotalOutputBandwidth(interface); }
```

The policy rule PolicyToAddTrafficClass is evaluated at run-time by the management agent NetworkPMA. When the constraint following the when clause evaluates to true, the target router is instructed to perform locally the addTrafficClassRequest operation.

Figure 5.4 presents the result of the enforcement of the rule PolicyToAddTrafficClass on the target router /Routers/CoreRouters/Athena. The Network PMA receives the obligation event addTrafficClassRequest with parameters the network interface on the target where the new class is to be added, the DSCP associated to the traffic class and the bandwidth that will be assigned to this class. As we can see in Figure 5.4, the policy rule retrieves the relevant CIM variables from the target router and the policy constraint evaluates to true upon the request addTrafficClassRequest ("10.0.2.2", 63, 4096). This in turn means that the policy rule is valid at the current network state and therefore is enforced on the network.

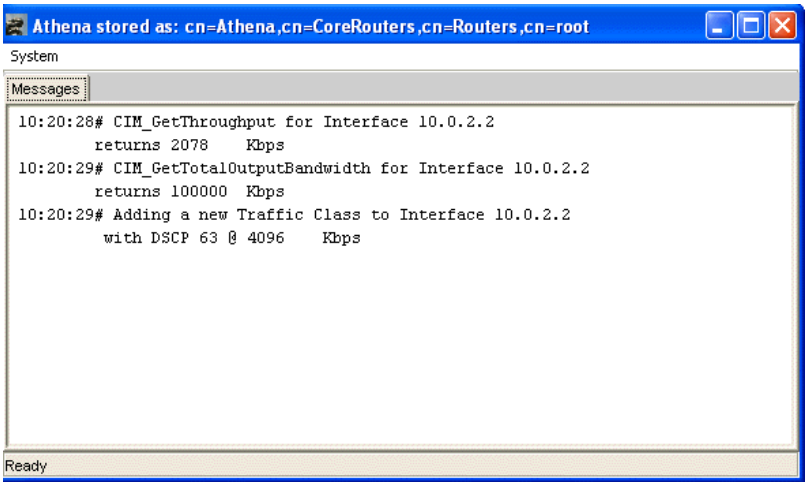


Figure 5.4 Result of enforcement of the policy validation rule PolicyToAddTrafficClass

Further implementation work is needed to support the enforcement of metapolicies within the Ponder toolkit. Future work will integrate our policy validation framework with the CISCO Information Model (CIM-CX). This will enable our framework to apply to both CISCO and Linux DiffServ routers.

5.4 Discussion

We have presented an approach for validation of policies with respect to the devices to which they apply. This includes validation that target devices support required functionality by means of metapolicies (ie. constraints on the allowed policies) or by means of constraints within the policy rules that check at run time that the policy does not violate resource state constraints within the network.

Further implementation is needed to support meta-policies within the Ponder compiler. The Ponder toolkit also needs to be extended to support deployment and enforcement of meta-policies.

Further to the implementation of policy validation for Linux DiffServ routers which we presented in this chapter, our framework provides tool support for validating that the policies actually achieve their objectives in the network. This can only be realised by testing, but testing on a live network may not be practical so it is essential to be able to test the results of policy enforcement on large-scale simulated DiffServ networks using the policy-based extensions to the J-Sim network simulator, as we presented in Chapter 4. This enables policy administrators to check offline the behaviour of their policy rules, before applying the rules onto the real network.

Although we presented examples of DiffServ routers, the approach could be applied to validation of security management policies or building management policies etc. A representation of the device capabilities using an information model such as CIM is essential. Our approach applies to individual network devices within a domain in which the DiffServ based policies apply. The situation becomes much more complex when interactions between the policies related to end-to-end flows or different service level agreements (SLAs) are considered. This requires determining whether the introduction of a new SLA could potentially violate the policies relating to existing SLAs. In the simplest situation this could require determining whether the current network topology has the resources to support the new SLA which can be done by Traffic Engineering systems.

However, when the SLAs cater for dynamic allocation of resources based on time or application requests, this becomes more complex to do.

Another issue is that the end-to-end path may not be within the administrative domain of a single service provider. This requires interaction between service providers to exchange policy information, and current state of the network topology.

Chapter 6
Critical Analysis

The work presented in this thesis has been motivated by the need to provide dynamic adaptation of policy in response to changes within the managed environment and validation of policy in respect to the capabilities of the network elements implementing the policy. In this chapter, we will evaluate our achievements by comparing our work against related work and by listing the limitations and deficiencies of our framework. Finally, we will provide suggestions for improvement.

6.1 Comparison with Related Work

Policy-based management has been introduced as a promising solution to the problem of management of complex and heterogeneous networks consisting of elements with different capabilities. Most of the work carried out in this research area has been focusing mostly on solutions for policy specification and policy enforcement architectures for QoS and security management. However, considering the dynamic nature of the networks, existing policy frameworks, such as the IETF Policy framework [IETF Policy Working Group], do not provide mechanisms to react to changes within the managed environment. Rather, they can only provide static policy configurations and therefore lack the flexibility and are not sufficient to handle changes in the underlying environment where policy applies. To address this problem, research efforts have been made towards the deployment of adaptive policy based management frameworks.

The frameworks for policy adaptation that we presented in section 2.6 of this thesis, eg. [Lutfiyya et al. 2001; Yoshihara et al. 2001; Marshall et al. 2001b; Ahmed et al. 2002] either implement mechanisms to adapt policy parameters based on monitoring information, or they are tailored for a specific network environment, implementing a single adaptation strategy to adapt policies in their systems. Our framework introduces higher-level policies as a means to decide which lower-level policy to enable upon a particular event indicating the need to adapt the policy configuration in the network. We provide the flexibility to implement and use within our management system not a single, but multiple adaptation strategies. New service management policies can be deployed at runtime within our system, in order to change, when required, the adaptation strategy and configure the parameters of the new strategy.

In comparison to many approaches for QoS adaptation in the Distributed Systems area, a brief account of which we presented in section 2.5 of this thesis, the advantage of our approach is that rather than using a single adaptation technique, hard-coded in the system, adaptation is implemented within the policy-driven management system itself, providing the flexibility to dynamically modify the adaptation strategies by enabling different policies which use different adaptation techniques. Futhermore, code implementing a new adaptation strategy can be loaded at run-time within the Policy Management Agents and policies can be updated to access this new functionality.

A lot of work has been carried out by standards organisations to provide information models and management protocols. Using the CIM representation of managed entities, not only provides our framework with the necessary abstractions to specify policy independently of the type of the device, but also provides a flexible architecture for mapping the policy rules into device configuration independent of the communication protocol between the policy system and the managed devices.

Our policy-based management framework also complements approaches that concentrate on low-level policy specification. Examples include: using compiled or interpreted programs to specify policies (e.g. using Java programs as described in [Martinez et al. 2002]), or using an information modelling approach for QoS policy specification as described by the IETF [Strassner et al. 2001]. Policies can be more conveniently specified in Ponder and then mapped to any lower-level specification using automated software tools (i.e. backends to the Ponder compiler).

In policy-based networking most of the tool support comes from the industry and is based on the IETF policy framework. Unlike the Ponder framework, most of these tools are specific to quality of service and bandwidth management with few providing access control configuration, while most of this work focuses on managing individual network elements. Scalability to enterprise wide management is not obvious as the dissemination of policies to specific elements is often performed manually whereas in Ponder this is automated based on domain membership of subjects and targets, which are explicitly specified in policies. Most of these products are optimised to work on single vendor networks and are implemented to manage vendor-specific hardware with only a few working towards hardware-independence. The framework we have described in this thesis enables the implementation of multi-vendor policy-based management products.

A fundamental problem which remains unsolved in all work on policy-based management is the issue of policy validation, i.e. given a policy, can it be instrumented in an existing

hardware/software configuration? We have presented a novel approach for validation of policies with respect to the devices to which they apply. This includes validation that target devices support required functionality by means of metapolicies or constraints on the allowed policies; using constraints within the policy rules to check at run time that the policy does not violate resource state constraints within the network.

6.2 Evaluation of the Framework

In this section we will point out the limitations and deficiencies of the overall work achieved. We will start with the work accomplished on the deployment of a policy adaptation framework.

The new idea for adaptation of network-level policies is that service management policies can be dynamically introduced to select, given an adaptation request, the most suitable implemented network policies. New adaptation strategies can be implemented by introducing service management policies that use new selection algorithms. We suggested a number of adaptation strategies for a DiffServ environment, using selection algorithms based on the QoS guarantees of DiffServ network-policies, the policies' associated path of routers and the time period when policies are valid. Additional selection algorithms can be investigated, taking into account additional criteria, such as determining which QoS class of service should be chosen for an application based on the pricing schemes that apply to the available service classes. Another aspect is that in our framework, the adaptation strategies that the administrator uses by means of service management policies, such as algorithms to select among 'fallback' classes of service when the main class of service cannot be provided, are hidden from the customers. New work on SLA specification could formally define the adaptation strategies that need to be undertaken upon network failures, SLA violations, etc. In this case, additional mechanisms would need to be implemented to derive service management policies from the "dynamic" part of an SLA. Work has been done in this direction in [Zhao 2002], where a new language is proposed for specifying XML-based contracts for a SLA management system, based on the contract model proposed in [Keller et al. 2002]. The language allows specification of the dynamic aspects of a contract, e.g. switching between different service levels according to specific conditions such as timer events, system failures, etc. An SLA parser component is implemented to derive policies from the contracts specified within the system. Derived policies can be deployed within the adaptive management framework presented in this thesis.

Our policy adaptation framework is distributed in that all objects including network-level and service management policies and network and service-level policy management agents are distributed across the network. Policies are distributed to their enforcement agents (PMAs and ACs) and the policy decision process is performed locally at each enforcement agent. However, problems may arise when different service-level management agents are trying to enforce adaptation strategies that result in conflicting actions on the managed devices. This would require analysis of service management policies for conflicts, as discussed in [Bandara et al. 2003]. In our adaptive framework, conflict analysis is more complex to perform since the enforcement of service management policies results in a set of low-level actions that is not static. Rather, this set of policies is determined at run-time by the selection algorithms within the service management policies.

We have found that Ponder is well suited for specifying policies for CIM described managed objects. Use of the CIM model to provide the underlying semantics for policy rules facilitates an improved and consistent environment for policy definition and enables mapping of Ponder policies into device configuration independently of the communication protocol between the policy-based management system and the managed devices.

We acknowledge the existence of available technologies that can be used to implement policy management solutions and refrain from specifying new technologies or protocols in our management framework. The architecture for policy enforcement and policy validation is based on simple ideas and can be implemented using existing protocols. We do not define any new protocols for the communication of policy information. Instead, depending on the implementation of the architecture, we assume that policy information is communicated using existing mechanisms. These can be object-middleware such as CORBA, protocols such as the Common Open Policy Service Protocol (COPS) or the Internet management architecture's Simple Network Management Protocol (SNMP).

We found that use of the CIM framework for representing network devices can provide two types of management information: a) Information about device's capabilities, and b) Information about the current state of the device. We use this information to support static policy validation with respect to device capabilities, which can be performed offline using meta-policies as a means to specify the constraints on the permitted DiffServ mechanisms, resources and bounds on object attributes. The required information about the capabilities of the devices can be obtained from CIM models. Dynamic policy validation with respect to current state of the device can only be performed at the time the

policy is to be enforced by means of constraints which are specified as part of the policy rules to define the conditions under which the policy is valid. These constraints can reference information extracted from a corresponding CIM model.

The applicability of the proposed policy validation framework is limited to performing checking of policy rules against the capabilities of individual network devices within a domain in which the policies apply. The situation becomes much more complex when interactions between the policies related to end-to-end flows or different service level agreements (SLAs) are considered. This requires determining whether the introduction of a new SLA could potentially violate the policies relating to existing SLAs. In the simplest situation this could require determining whether the current network topology has the resources to support the new SLA which can be done by Traffic Engineering systems. However, when the SLAs cater for dynamic allocation of resources based on time or application requests, this becomes more complex to do.

Another issue is that the end-to-end path may not be within the administrative domain of a single service provider. This requires interaction between service providers to exchange policy information, and current state of the network topology. The Ponder compiler can generate policies in an XML format to enable easy exchange of policies, but policies usually refer to specific domains so may not be transferable. For more information refer to the Ponder Policy Group documentation [Ponder Policy Group]. A number of problems may arise when using policies for configuration of a service provider's network. For example the forward and return path between two end points may not be symmetrical within a network. This implies that there may be needed to define two sets of policies – one to cater for the inbound direction and the other for the outbound direction.

6.3 Evaluation of the Implementation

6.3.1 Evaluation of the policy enforcement implementation

A limitation of our enforcement implementation is that enforcement of DiffServ policies is supported only for policies specified in terms of TCBs, but not for policies specified in terms of single CIM objects (eg, a policy that will add/update/remove a single MeteringService CIM object). This is due to the fact that one needs to know the location of a single CIM object inside the CIMOM's space. This would require integration of the Ponder Editor tool with a Graphical Tool, such as CimNavigator [Cim Navigator], to graphically browse CIM objects and their associations within the CIMOM. Use of this tool will enable the administrator to select which CIM object to update/remove in/from

the CIMOM or alternatively to select the position where a new CIM object should be added in the CIMOM.

Ponder Management Agents interpreting DiffServ policies are written in Java and their communication with the Java implementation of the WBEM services CIMOM [WBEM Services Project 2003] is realised using Java RMI. This leads to a poor performance when enforcing obligation policies that involve CIM operations within the CIMOM. In addition, the CIM2TCDriver component is also written in Java and communicates with the CIMOM with Java RMI leading to an overall poor performance to communicate policy actions to Linux routers. The rationale for using the Java platform for deployment of the CIM2TC Driver component and its RMI communication with the CIMOM was due to compatibility issues. The WBEM services CIMOM, which our implementation uses, is implemented in Java. Other available CIMOMs, eg. the CIMOM from the Open Group [The Open Group], are also mostly implemented in Java. Other available technologies could be used to implement the communication protocol from the CIMOM with the Linux device, such as CORBA, COPS or SNMP. In the latter two cases, an additional software layer would be required to convert the CIM Network Common Model classes to the respective object models used by the management protocols (i.e. DIFFSERV-PIB and DIFFSERV-MIB), while COPS or SNMP agents should be running in the Linux device. We have not found a COPS or a SNMP agent implementation for Linux, so we have not implemented use of COPS and SNMP to perform policy enforcement and evaluate the performance of these approaches. However, since the enforcement of DiffServ provisioning policies is not a frequent operation, we do not expect that the performance overhead of our implementation will increase significantly the management overhead for a DiffServ network. The main purpose of the prototype was to evaluate the feasibility of implementing the proposed framework, and this was satisfactorily achieved.

Finally, use of domain-based Ponder specification for network policy provides scalability with respect to the number of configuration policies that need to be deployed within the network, as the same policy is applied to sets of devices and objects within the managed network, rather than being limited to apply only to individual devices.

6.3.2 Evaluation of the policy adaptation implementation

The experiments conducted with our prototype implementation of the policy adaptation architecture for simulated DiffServ networks have proven the feasibility of our approach. Use of simulated networks enabled us to experiment in various network scenarios and use results as feedback to resolve a range of problems that may arise in an adaptive network

management system, such as oscillations that occur when small changes in performance generate events to trigger configuration changes. A next step could be to deploy the proposed architecture in real test-beds, using available third-party monitoring and fault reporting tools to provide measurements and generate events indicating network failures, etc.

In the current version of the implementation, the administrator manually edits the QoS guarantees of DiffServ Per Domain Behaviour (PDB) policies in the QoS Policy database (refer to Figure 4.8 for an example of a policy database instance). This can lead to accurate selection of PDB policies only under the assumption that the expert user has the tools to analyse the network in order to predict the QoS guarantees of PDB policies. This is due to the fact that the characteristics of a PDB depend on network-specific features, eg. maximum number of hops, maximum transit delay of network elements, minimum buffer size available for the PDB at a network node, etc. Alternative approaches could be adopted in order to predict the QoS guarantees of the PDB policies based on analysis of monitored information.

The introduction of service management policies, which implement the selection algorithms, increases the management overhead within our adaptive framework, as they need to dynamically choose which lower-level network policy must be enabled, given an adaptation request. However, the selection algorithm does not require sorting of the behaviour attributes of lower-level policies, ie. the policies' characteristics upon which the selection is based. In most cases, as in the examples we presented in Chapter 4, it is adequate to perform a search operation to find, given a request, only the closest policy attribute value to the request, the minimum, or the maximum value of a policy attribute in order to decide which policy to enable. As a consequence, use of service management policies scales linearly with the number of deployed low-level policies.

6.3.3 Evaluation of the policy validation implementation

The implementation of the policy validation architecture uses the same technologies as the policy enforcement architecture of CIM-based policies. Queries on attributes of CIM objects, modelled in our DiffServ metrics CIM extension, are issued using Java RMI from Ponder PMAs to the WBEM services CIMOM, which in turn communicates with a Java component CIMMonitor (see Figure 5.3) to get the corresponding variables from the Linux operating system. The performance of this implementation is relatively poor, compared to other implementations that could use more efficient technologies in terms of performance, other than Java and Java RMI.

As far as the scalability of the implementation is concerned, runtime checking of policy by means of constraints specified in the policy, scales linearly with the number of devices within the target domain within which the policy constraint needs to be evaluated. To analyse the scalability issues of providing validation by means of meta-policies, consider a meta-policy rule for validating N policies. If the maximum number of target elements in an arbitrary policy is T, then evaluating the OCL constraint in the meta-policy rule (eg. the constraints in examples 5.3 and 5.4) requires checking the static capability of all targets within all deployed policies. Hence, the complexity for evaluating the constraint is O(N*T), and the implementation may not perform adequately when the number of policies and target elements are large. One way to resolve this scalability issue is to group devices with the same capabilities under the same target domain. Note that in the Ponder policy framework, a target object may belong to one or more domains at the same time. Thus, multiple dimensions of domains can be defined for grouping policies' target elements. For example, one domain can be defined for grouping the core routers in a DiffServ network, while another domain can be used for grouping the Cisco routers within the network. When grouping devices with the same capabilities under the same domain, there is no need to evaluate the constraint of the meta-policy against all target elements within a policy, but against only one among the same-type targets in the target domain (eg. if the target domain contains routers of the same type, connected with identical link capabilities). This implies that the implementation will scale with the number of target elements in the network.

Finally, the current version of the Ponder compiler does not implement some of the features of OCL, so additional work is needed to support syntax checking and enforcement of meta-policies within the Ponder toolkit.

6.4 Conclusions

This chapter provided a critical analysis of the overall work achieved. We compared our framework with other approaches for providing adaptation and validation within a policy-based management framework and discussed how our work complements approaches that concentrate on the implementation of adaptive mechanisms and platforms. We also discussed how the abstractions provided by our framework cater for use of available lower-level management protocols and lower-level policy specification.

We discussed the limitations in terms of functionality and the shortcomings in terms of performance and scalability of the proposed policy enforcement, adaptation and

validation architectures and their implementation. We identified the implementation issues that have been hard to provide, and those that need to be resolved in the future.

The implementation of our novel ideas has proven the feasibility of our approach to providing an adaptive policy-based management framework with mechanisms for validation of policy. Note however, that no major efforts were undertaken to optimise the performance of the "proof of concept" implementation provided in this thesis. Implementation still needs to be provided for some of architectural components of the framework.

Chapter 7
Conclusions

This chapter concludes the thesis by providing a summary of the results achieved and examines the extent to which our work meets the requirements for a policy based management framework that provides solutions to the problems of adaptation and validation of network management policy. Based on the critical analysis of our proposed framework, which we presented in the previous chapter, we will conclude with a summary of directions for future work.

7.1 Review and Discussion of Achievements

New technologies are being developed to build networks to support the growing demands of network applications and the needs of increasing numbers of users. However, although the cost of hardware to build faster, more reliable networks has decreased over time, the management cost of modern complex network infrastructures does not decrease at a similar rate. This is due to the fact that network management still remains a difficult task, requiring considerable effort by many competent network administrators. Automation, simplification and interoperability of network management solutions are key factors that will help administrators perform their tasks more easily and effectively and hence reduce the management cost of contemporary networks.

Policy-based management is increasingly gaining attention from industry and from academia as a promising solution to automate and simplify the management task. However, no standard policy languages exist, the standard policy information models from the Internet Engineering Task Force (IETF) and the Distributed Management Task Force (DMTF) have limitations and are difficult to use, whilst the commercial policy toolkits do not yet support a high degree of automation and full interoperability. The work presented in this thesis is motivated by the need to provide solutions to the problems of policy adaptation and policy validation that remain unsolved for policy-based management of distributed systems. We believe that the proposed ideas will provide a solid background for future research, while their implementation will yield more automated and effective policy-based network management products.

We presented a survey of policy-based management architectures and platforms, and management frameworks implementing policy adaptation techniques. This survey led us to the conclusion that significant challenges still need to be addressed in order to provide

adaptation of policy in a more systematic and generic way than existing approaches. Moreover, we did not find any implementation of a policy based framework that caters for validation of policy as a means to ensure that policies can lead to feasible configurations in a given existing network environment. The ideas we propose in this thesis strive to deliver solutions to the open problems of policy adaptation and policy validation. The basis of our work has been Imperial College's Ponder policy framework, which includes the Ponder policy language for specifying security and management policies, a generic deployment model for Ponder policies, and an integrated policy toolkit [Damianou et al. 2001; Dulay et al. 2001; Damianou et al. 2002].

First of all, we addressed the requirement for a policy-based management framework to cater for complex and large-scale distributed systems. This requirement is the common denominator of the set of requirements for deploying any policy-based management framework. It is essential that the framework includes the necessary abstractions of the managed elements on one hand, and that mechanisms are supported to automate distribution of policies to sets of devices rather than individual ones on the other. The solution to this requirement was given by extending the Ponder policy framework with the CIM modelling framework for representing the managed entities where Ponder policies apply. The rationale for choosing Ponder as the policy specification language was that Ponder's object-oriented features allow user-defined types of policies to be specified and then instantiated multiple times with different parameters. This provides extensibility of the management system. In addition, support for domains as a means of grouping objects to which policies apply, facilitates policy specification for large-scale systems. On the other hand, use of the CIM modelling framework that represents the abstractions of policy target objects, such as logical and physical network elements, services, etc., not only provides the Ponder framework with the necessary abstractions and the underlying semantics to specify policy independently of the type of the device, but also provides the flexibility to allow mapping of the policy rules into device configuration independent of the communication protocol between the policy system and the managed devices. Interoperability is achieved this way, as standard (eg. SNMP and COPS) or vendor specific management protocols can be used to communicate management information to different types of devices.

As a proof of concept, we implemented a novel algorithm to map CIM-based DiffServ policies to low-level configuration commands for Linux routers. We used this implementation to enforce several configuration scenarios on a set of routers within our DiffServ testbed network. The results have proven the feasibility of our approach and the

practical benefits of using a policy-based management framework: policies simplify management of complex network mechanisms.

For the problem of policy adaptation, we first analysed in a systematic way the possible types of policy adaptation that are required in a policy-driven management system. In the first type, adaptation is carried out by dynamically changing the attributes of network policies. In the second type, mechanisms should be implemented to dynamically select which of the implemented network policies should cater for a particular event, indicating changes within the managed environment. The third and most general type of adaptation is to learn from the system behaviour which management strategies are the most suitable. This can be used to select among implemented policies or even generate new ones when needed. We have presented ideas on how policy adaptation can be realised within the Ponder policy framework for the first two types of adaptation. For the first type, the parameters of the policy actions are calculated at run-time, based on the parameters of the event triggering the policy. The administrator is given the flexibility to accommodate, within the system, a variety of techniques for deriving policy action parameters from event parameters which measure the changes within the managed network. The solution to the second type of adaptation was given by introducing higher-level policies within the policy based management system. These higher-level policies are used to manage the underlying network-level policies, by selecting at run-time which network-policy should be enforced for a particular situation. A variety of selection algorithms can be implemented within the management system, providing the administrator with the flexibility to change the adaptation strategy when required.

We have designed a generic enforcement architecture for policy adaptation within the Ponder framework. Policy Management Agents (PMAs) at the service-level interpret service-level obligation policies to select which network-level obligation or authorisation policies should be enforced upon requests indicating changes in the managed system. The applicability of our approach has been tested with a prototype implementation for simulated DiffServ networks. However, the applicability of our ideas on policy adaptation is not restricted to a DiffServ framework, but the same ideas could be used to implement adaptive policy-based management systems for management of different networking mechanisms (eg. security) and for several aspects of distributed systems (eg. storage and ubiquitous systems management). The results of our experiments have shown the feasibility of our approach and have proven that policy adaptation can provide a flexible means of dynamically changing the adaptation strategies within policy-enabled networks.

One of the reasons that many network administrators do not yet trust policy-based management products is the fact that they do not know whether the enforcement of policy will ensure feasible configurations of their systems; it is thus important to provide tools to validate policies prior to their enforcement. We addressed the requirement of policy validation by using the device-independent CIM modelling framework to provide our policy-based management system with two types of information: a) information about device capabilities (e.g. type of mechanisms it supports, bounds on resources); and, b) information about the current state of the device (e.g. current resource allocation, current values of specific devices' variables). We use this information to support two types of policy validation: a) static policy validation with respect to device capabilities. We perform this type of validation offline, using meta-policies as a means to specify the constraints on the permitted mechanisms and resources that policies can use within the device; and, b) dynamic policy validation with respect to current state of the device. This type of validation is performed at the time the policy is to be enforced by means of constraints which are specified as part of the policy rules to define the conditions under which the policy is valid. Failure to satisfy the policy constraints prevents the execution of the policy. These constraints can reference information extracted from corresponding CIM models.

We have implemented an architecture for validating policies that apply to DiffServ networks. Target routers' capabilities are extracted from the CIM Network Common Model, while information about the current device configuration (e.g. number of implemented classes, bandwidth allocated to each DiffServ class, etc.) is provided by a "DiffServ-metrics" extension to CIM that we have designed and implemented within the WBEM Services CIMOM for providing DiffServ run-time statistics.

Throughout this thesis we have used examples, mostly related to management of Differentiated Services networks, to describe the concepts of the proposed framework. The framework is however applicable to a wider range of management areas, such as network security, storage and ubiquitous systems management. Our ideas on policy enforcement, adaptation and validation are not restricted to specific management protocols, mechanisms or information models; the same ideas can be used in different management areas using corresponding protocols, mechanisms or information models. For example, the Diadem EU project [Diadem Project] will be applying our ideas to adaptive management of distributed firewalls.

7.2 Future Work

We have identified several issues for further work through the critical evaluation of the work in Chapter 6. We list the most important of them in this section and extend them with a few additional issues.

7.2.1 Policy enforcement architecture

We have implemented enforcement of CIM-based DiffServ policies for Linux routers. It would be easy to extend the implementation to provide management of commercial routers. Future work could also use this approach for management of other QoS mechanisms, such as MPLS.

Further work is needed to extend the implementation to use other available management protocols for policy enforcement, such as SNMP and COPS and evaluate the limitations and the performance implications of each approach for executing policies on target routers.

7.2.2 Policy adaptation framework

The area of policy adaptation still needs further investigation. The ideas presented in this thesis provide solutions for adaptation of network-level policies according to changes in application requirements and events indicating network failures or problems delivering services at the desired levels. The same concepts could be used to cater for policy adaptation when policy is defined at the application level. In this case, policy-enabled applications interpret policies to tailor their behaviour according to events they receive from the network, such as the available bandwidth, or according to requirements of specific users, such as what information to filter when bandwidth or device capabilities are limited. Policy adaptation techniques such as those described in this thesis can be used to adapt policies within applications as a means to dynamically change the adaptation strategies at the application-level.

Note that some of the application-specific policies may have to be enforced within the network. Thus policy-enabled applications need to be able to transfer policies to the network. Conversely, the network may need to pass policies to be interpreted by the application for more efficient adaptation, for instance related to caching or monitoring of application specific components. It is necessary to develop techniques and interfaces for

interaction between policy-based applications and policy-enabled networks in order to exchange policy information.

Considerable research is required to investigate use of self-learning mechanisms to find out which are the most suitable policy configuration strategies from the system behaviour. This can be used to enhance the functionality of the policy adaptation system to select policies or generate new ones when needed.

7.2.3 Policy validation framework

Further implementation is needed to support meta-policies within the Ponder compiler. The Ponder toolkit also needs to be extended to support deployment and enforcement of meta-policies.

This thesis has only addressed the problem of policy validation in respect to the capabilities of the network elements implementing the policy. Additional work is needed to validate policies related to end-to-end flows or different service level agreements (SLAs). In the latter case, it should be determined whether the introduction of a new SLA could potentially violate the policies relating to existing SLAs.

7.3 Closing Remarks

The work presented in this thesis has been motivated by the need to provide solutions to the problems of policy adaptation and validation. The results of our work will benefit the network management community at large. Applying Ponder policies to CIM represented network elements and services will benefit all those in industry and academia adopting the CIM standard. Our novel ideas on policy adaptation and policy validation can be successfully applied to real-life configuration management scenarios and constitute a background basis that will help researchers in this area.

Bibliography

A. Mankin et al (1997), *Resource ReSerVation Protocol (RSVP) -- Version 1 Applicability Statement Some Guidelines on Deployment, RFC 2208,* September 1997.

Aedus (2004), *Adaptable Environments for Distributed Ubiquitous Systems,* http://www-dse.doc.ic.ac.uk/Projects/aedus/

Ahmed, T., A. Mehaoua and R. Boutaba (2002). *Dynamic QoS Adaptation using COPS and Network Monitoring Feedback*. In Proceedings of the IFIP/IEEE International Conference on Management of Multimedia Networks and Services, Santa Barbara, CA, October 6-9, 2002.

Allot Communications *NetPolicy Policy Based Management System product documentation, available from http://www.allot.com*

Almesberger, W. (1999). *Linux Network Traffic Control - Implementation Overview*. In Proceedings of the 5th Annual Linux Expo, Raleigh, NC, May 1999, pp. 153-164.

Amuse (2004), *Autonomic Management of Ubiquitous Systems for e-Health,* http://www.doc.ic.ac.uk/~ecl1/projects/AMUSE/index.html

Aurrecoechea, C., A. Cambell and L. Hauw (1998). "A survey of QoS architectures." ACM/Springer Verlag Multimedia Systems Journal, Special Issue on QoS Architecture **6**(3): 138-151.

Baker, F., A. Smith and K. Chan (2001), *Differentiated Services MIB, Internet Draft, draft-ietf-diffserv-mib-09.txt,* Mar. 2001.

Bandara, A., E. Lupu and A. Russo (2003). *Using Event Calculus to Formalise Policy Specification and Analysis*. In Proceedings of the 4th IEEE Workshop on Policies for Distributed Systems and Networks (Policy2003), Como, Italy, June 2003.

Bearden, M., S. Garg and W. Lee (2001). *Integrating Goal Specification in Policy-Based Management*. In Proceedings of the Policy 2001: International Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag LNCS 1995, 29-31 Jan. 2001, pp. 153-170.

Bernet, Y., A. Smith, S. Blake and D. Grossman (2002), *An Informal Management Model for DiffServ Routers, RFC 3290,* May 2002.

Bertino, E., M. Cochinwala and M. Mesiti (2003). *UCS-Router: A Policy Engine for Enforcing Message Routing Rules in a Universal Communication System*. In Proceedings of the Third International Conference on Mobile Data Management, MDM 2003, Singapore, Jan. 8 - 11, 2002.

Bhatti, S. and G. Knight (1999). "Enabling QoS adaptation decisions for Internet applications." Computer Networks **36**(7): pp. 669-692.

Braden, R., D. Clark and D. Shenker (1994), *Integrated Services in the Internet Architecture: an Overview, RFC 1633,* June 1994.

Braden, R., L. Zhang, S. Berson, S.Herzog and S. Jamin (1997), *ReSerVation Protocol (RSVP) Version 1 Functional Specification, RFC 2205,* Sep. 1997.

Brunner, M. and J. Quittek (2001). *MPLS Management using Policies*. In Proceedings of the IM 2001: 2001 IEEE/IFIP International Symposium on Intergrated Network Management, Seattle, WA, USA, 14-18 May 2001, pp. 515-528.

Campbell, A. (1996), *A Quality of Service Architecture,* PhD Thesis, Lancaster University , UK, Jan. 1996.

Carlson, M., W. Weiss, S. Blake, Z. Wang, D. Black and E. Davies (1998), *An Architecture for Differentiated Services, RFC 2475,* Dec. 1998.

Chan, K., D. Durham, S. Gai, S. Herzog, K. McCloghrie, F. Reichmeyer, J. Seligson, A. Smith and R. Yavatkar (2001), *COPS Usage for Policy Provisioning, RFC 3084,* March 2001.

Cim Navigator. http://cimnavigator.com

Cisco Systems Inc. *COPS QoS Policy Manager, available from http://www.cisco.com*

Cucchiara, J., H. Sjostrand and J. Luciani (2001), *Definitions of Managed Objects for the Multiprotocol Label Switching, Label Distribution Protocol (LDP), Internet Draft, draft-ietf-mpls-ldp-mib-08.txt,* Aug. 2001.

D. Durham et al. (2000), *The COPS (Common Open Policy Service) Protocol, RFC 2748,* January 2000.

D. Goderis et al. (2001), *Service Level Specification Semantics, Parameters and negotiation requirements, draft-tequila-sls-01.txt,* June 2001.

Damianou, N. (2002), *A Policy Framework for Management of Distributed Systems,* PhD Thesis, Dept. of Computing, Imperial College, March 2002.

Damianou, N., A. Bandara, M. Sloman and E. Lupu (2002b). "A Survey of Policy Specification Approaches." IEEE Network, **16**(2): pp. 10-19.

Damianou, N., N. Dulay, E. Lupu, M. Sloman and T. Tonouchi (2002). *Tools for Domain-based Policy Management of Distributed Systems*. In Proceedings of the IEEE/IFIP NOMS 2002: Network Operations and Management Symposium, Florence, Italy, Apr. 2002, pp203-217.

Damianou, N., E. Lupu and M. Sloman (2001). *The Ponder Policy Specification Language*. In Proceedings of the Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag LNCS 1995, Jan. 2001, pp. 18-39.

Diadem Project *Distributed Adaptive Security by Programmable Firewall,* http://www.diadem-firewall.org/index.php

DMTF (2004), *Common Information Model (CIM) Version 2.7 Specification*.

DRCL J-Sim. *J-Sim Network Simulator, available from http://www.j-sim.org*

Dulay, N., E. Lupu, M. Sloman and N. Damianou (2001). *A Policy Deployment Model for the Ponder Language*. In Proceedings of the IM 2001: 7th IEEE/IFIP International Symposium on Intergrated Network Management, Seattle, USA, 14-18 May 2001, pp. 529-544.

E. Gamma et al. (1994), *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison Wesley.

E. Rosen et al. (2001), *Multiprotocol Label Switching Architecture, RFC 3031,* January 2001.

EasyLiving Project. *http://research.microsoft.com/easyliving/*

Ferrari, T. and F. Chimento (2000). *A Measurement-based Analysis of Expedited Forwarding PHB Mechanisms*. In Proceedings of the IWQoS 2000, Pittsburgh, PA, USA, June 2000.

Fine, M., K. McCloghrie, J. Seligson, K. Chan, S. Hahn, C. Bell, A. Smith and F. Reichmeyer (2001), *Differentiated Services Quality of Service Policy Information Base, Internet Draft, draft-ietf-diffserv-pib-03.txt,* March 2001.

G. Blair et al. (2000). "Supporting Dynamic QoS Management Functions in a Reflective Middleware Platform." IEE Proceedings - Software. **147**(01): pp. 13-21.

G. Gordon et al. (1997). *Adaptive Middleware for Mobile Multimedia Applications*. In Proceedings of the NOSSDAV '97: Network and Operating System Support for Digital Audio and Video, St Louis, USA, 1997.

Haahr, M., R. Cunningham and V. Cahill (2000). *Towards a Generic Architecture for Mobile Object-Oriented Applications.* In Proceedings of the SerP 2000: Workshop on Service Portability, San Francisco, CA, USA.

Hamada, T., P. Czezowski and T. Chujo (2000), *Policy-based Management for Enterprise and Carrier IP Networking,* Fujitsu Scientific and Technical Journal, Special Issue on Information Technologies in the Internet Era, Vol. 12, 2000.

Heinanen, J., F. Baker, W. Weiss and J. Wroclawski (1999), *Assured Forwarding PHB Group, RFC 2597,* Sep. 1999.

HP Policy Expert. http://www.openview.hp.com/products/pxpert/index.html.

Husein, M. (2003), *Graphical user interface to a policy based network management simulator,* MSc. Thesis, Department of Computing, Imperial College London, Sep. 2003.

IETF Policy Working Group. *http://www.ietf.org/html.charters/policy-charter.html*.

Jacobson, V., K. Nichols and K. Poduri (1999), *An Expedited Forwarding PHB, RFC2598,* Sep. 1999.

Keller, A., G. Kar, H. Ludwig, A. Dan and J. Hellerstein (2002). *Managing Dynamic Services: A Contract-based Approach to a Conceptual Architecture*. In Proceedings of the NOMS 2002: 8th Network Operations and Management Symposium, Florence, Italy, 15-19 Apr. 2002.

Kim, J.-Y., J. W.-K. Hong, S.-H. Ryu and T.-S. Choi (2000). *Constructing End-to-End Traffic Flows for Managing Differentiated Services Networks*. In Proceedings of the 11th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Houston, Texas, USA, Springer LNCS 1960, Dec. 2000, pp. 83-94.

L. Kagal et al. (2003). *A Policy Language for A Pervasive Computing Environment*. In Proceedings of the Policy 2004: IEEE 4th International Workshop on Policies for Distributed Systems and Networks, Como, Italy, June 2003.

Levi, D. and J. Schoenwaelder (1999), *Definitions of Managed Objects for the Delegation of Management Scripts, RFC 2592,* May 1999.

Linux Advanced Routing & Traffic Control HOWTO (2002), *available from http://lartc.org*

Lupu, E. and M. Sloman (1999). "Conflicts in Policy-based Distributed Systems Management." IEEE Transactions on Software Engineering - Special Issue on Inconsistency Management: **25**(6): pp. 852-869.

Lutfiyya, H., G. Molenkamp, M. Katchabaw and M. Bauer (2001). *Issues in Managing Soft QoS Requirements in Distributed Systems Using a Policy-Based Framework.* In Proceedings of the Policy 2001: International Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag LNCS 1995, 29-31 Jan. 2001, pp. 185-201.

Marshall, I. and C. Roadknight (2001b). "Provision of quality of service for active services." Computer Networks, **36**(1).

Marshall, I., H. Gharib, H.Hardwicke and C. Roadknight (2001a). *A novel architecture for active service management*. In Proceedings of the 7th IEEE/IFIP International Symposium on Intergrated Network Management, Seattle, WA, USA, May 2001, pp. 795-810.

Martinez, M., M. Brunner, J. Quittek, F. Strauß, J. Schönwälder, S. Mertens and T. Klie (2002). *Using the Script MIB for Policy-based Configuration Management*. In Proceedings of the IEEE/IFIP NOMS 2002: Network Operations and Management Symposium, Florence, Italy, Apr. 2002, pp 187-202.

Moore, B., E. Ellesson, J. Strassner and A. Westerinen (2001), *Policy Core Information Model -- Version 1 Specification, RFC 3060,* February 2001.

Nichols, K., S. Blake, F. Baker and D. Black (1998), *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers, RFC 2474,* December 1998.

Nichols, K. and B. Carpenter (2001), *Definition of Differentiated Services Per Domain Behaviours and Rules for their Specification, RFC 3086,* April 2001.

OMG (1999), *Object Constraint Language Specification, version 1.3*.

Ponder Policy Group, http://www-dse.doc.ic.ac.uk/Research/policies/ponder.shtml

Prieto, A. and M. Brunner (2001). *SLS to DiffServ configuration mappings*. In Proceedings of the DSOM 2001: 12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Nancy, France, 15-17 Oct. 2001.

Segall, B. and D. Arnold (1997), *Elvin has left the building: A publish/subscribe notification service with quenching,* In Proceedings of the Queensland AUUG Summer Technical Conference, Brisbane, Australia, *available from http://www.dstc.edu.au/Elvin/doc/papers/auug97/AUUG97.html*

Sloman, M. and E. Lupu (1999). *Policy Specification for Programmable Networks*. In Proceedings of the First International Working Conference on Active Networks (IWAN'99), Berlin, Germany, ed. S. Covaci, LNCS, Springer Verlag, June 1999, pp. 73-84.

Sloman, M. and K. Twidle (1994). "Domains: A framework for Structuring Management Policy." Chapter 16 in Networks and Distributed Systems Management (Sloman, 1994ed), 1994a: pp. 433-453.

Snir, Y., Y. Ramberg, J. Strassner and R. Cohen (2001), *Policy Framework QoS Information Model, Internet Draft, draft-ietf-policy-qos-info-model-03.txt,* Apr. 2001.

Sprenkels, R. and A. Pras (2000). *A Customer Service Management Architecture for the Internet*. In Proceedings of the DSOM 2000: 11th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Texas, USA, 4-6 Dec. 2000.

Stone, G., B. Lundy and G. Xie (2001). "Network Policy Languages: A Survey and A New Approach." IEEE Network: Vol. 15, No 1, pp. 10-21, Jan 2001.

Strassner, J. "Directory Enabled Networking". Macmillan Technical Press.

Strassner, J., A. Westerinen, E. Ellesson, B. Moore and R. Moats (2001), *Policy Core LDAP Schema, Internet Draft, draft-ietf-policy-core-schema-11.txt,* May 2001.

Strassner, J., A. Westerinen and B. Moore (2001), *Information Model for Describing Network Device QoS Datapath Mechanisms, Internet Draft, draft-ietf-policy-qos-device-info-model-03.txt,* May 2001.

TEQUILA (2002), *Traffic Engineering for Quality of Service in the Internet, at Large Scale, http://www.ist-tequila.org*

The Open Group *SNIA CIMOM,* http://www.opengroup.org/snia-cimom/

Trimitzios, P. e. a. (2001). *An Architectural Framework for Providing QoS in IP Differentiated Services Networks*. In Proceedings of the IM 2001: 2001 IEEE/IFIP International Symposium on Intergrated Network Management, Seattle, USA, May 2001, pp. 17-34.

Uttamchandani, S., C. Talcott and D. Pease (2003). *Eos: An Approach of Using Behaviour Implications for Policy-based Self-management*. In Proceedings of the DSOM 2003: 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Heidelberg, Germany, 20-22 Oct. 2003.

Verma, D. (2001), *Policy-Based Networking, Architecture and Algorithms. New Riders Publishing*.

Verma, D., M. Beigi and R. Jennings (2001). *Policy Based SLA Management in Enterprise Networks*. In Proceedings of the Policy 2001: International Workshop on Policies for Distributed Systems and Networks, Bristol, UK, Springer-Verlag LNCS 1995, 29-31 Jan. 2001, pp. 137-152.

Wahl, M., T. Howes and S. Kille (1997), *Lightweight Directory Access Protocol (v3), RFC 2251, available from http://www.ietf.org,* December 1997.

Wang, N. e. a. *Adaptive and Reflective Middleware for QoS-Enabled CCM Applications, available from www.computer.org/dsonline*

WBEM Services Project (2003), *available from http://wbemservices.sourceforge.net*

Yoshihara, K., M. Isomura and H. Horiuchi (2001). *Distributed Policy-based Management Enabling Policy Adaptation on Monitoring using Active Network*

Technology. In Proceedings of the DSOM 2001: 12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, Nancy, France, 15-17 Oct. 2001.

Zelu, C. (2003), *Graphical User Interface for DiffServ configuration of CIM enabled Linux Routers,* MSc. Thesis, Department of Computing, Imperial College London, September 2003.

Zhao, L. (2002), *Service Level Agreement Implementation,* MSc. Thesis, Department of Computing, Imperial College London, September 2002.

Appendix A

A1. Adaptive Management of Network Security Mechanisms

Authorisation is often part of a policy-based management system to specify which users are permitted to access particular services or functions within the services. Consider a scenario where a business organisation provides a certain number of services to users who work within the organisation, to clients who have signed a Service Level Agreement with the organisation and to internet users. Access control agents should be implemented within the policy-based management system of the organisation to interpret authorisation policies and control requests related to each offered service. For example, the security administrator will define the policy type WebServiceAccessControlPolicy in Example A.1 to specify that users that belong to the subject domain domainS are not permitted to use the web service offered by the organisation's web servers.

Example A.1 Policy for controlling users' access to an organisation's web service

```
type auth- WebServiceAccessControlPolicy (subject domainS) {
target    /Servers/WebServers;
action    openHTTPconnection (); }
```

The policy type WebServiceAccessControlPolicy can be instantiated multiple times with different subjects to define access control policies for different sets of users. Example A.2 presents three policy instances of the type WebServiceAccessControlPolicy. Policy policy1 denies access of the web service to internet users, while policies policy2 and policy3 deny access to users within the Client A and Client B networks respectively. These policy instances, when enabled, will be interpreted by Access Control agents to configure the firewall of the organisation to reject web requests from the corresponding set of users.

Example A.2 Policy instances of the type WebServiceAccessControlPolicy

```
inst auth- policy1 = WebServiceAccessControlPolicy (/Users/InternetUsers);
inst auth- policy2 = WebServiceAccessControlPolicy (/Users/ClientA_Users);
inst auth- policy3 = WebServiceAccessControlPolicy (/Users/ClientB_Users);
```

Consider now a scenario where a Denial of Service (DoS) attack is detected against the web servers of the organisation. This would result in a situation where it is impossible for legitimate users to use the web service. In turn, this situation would indicate that the existing access control policy deployed within the network is not efficient to prevent the DoS attack, but needs to be changed. This adaptation of the access control policy can be implemented using the policy shown in Example A.3 which denies access of the web service to internet users, assuming that the attack originates from a set of internet users.

Example A.3 Policy for adapting the access control policy for the web service

```
inst oblig DoSAttackPolicy  {
subject ServiceManagementAgent;
on DoS();
do policy1.enable();
/*The authorisation policy denying access of internet users must be enabled */  }
```

Another adaptation strategy that could be implemented in a DoS attack scenario would require detecting where the attack originates. In this case, the ServiceManagementAgent could only enable an authorisation policy denying access to users of the network where the attack comes from. This adaptation strategy could be implemented with the management policy presented in Example A.4.

Example A.4 Policy for adapting the access control policy for the web service based on information of the DoS attack

```
inst oblig DoSAttackPolicy  {
subject ServiceManagementAgent;
on DoS ( networkID );
do
/* the constructor domain takes as input the IP address of the network where the attack
originates and constructs a subject domain representing the attacker's domain*/
domain attackerDomain = new domain ( networkID );
/* Instantiate an authorisation policy of the type WebServiceAccessControlPolicy with subject
the domain attackerDomain*/
inst authorisationPolicyForAttacker = WebServiceAccessControlPolicy (attackerDomain ) ->
/* Enable the new authorisation policy */
authorisationPolicyForAttacker.enable (); }
```

A2. Adaptive Management of Ubiquitous Systems

In the future, mobile communicators will be an important means of interacting with an intelligent ubiquitous environment in the home, office, hotels, in shopping malls or while travelling in order to support commerce, entertainment or navigation. Policies can be introduced in such environments to facilitate management of large number of devices with different capabilities. In particular, security policies can be deployed within the system to specify which users or devices are permitted to access services or data within other devices, while obligation policies can be used to specify the actions that need to be undertaken within the system when certain events are generated. For example, an obligation policy can be deployed to specify that "all voice messages arriving at the mobile device should be converted to text messages" when the user is in a meeting room, while another policy can be deployed to specify that "all text messages should be converted to voice messages", when the user is driving his car. Both policies will be triggered by an event indicating that a new message has arrived at the mobile device.

Considering the dynamic nature of a ubiquitous environment, we can perceive that adaptive management is required in several circumstances, in order to automatically change the management strategies when changes occur within the ubiquitous system. In our approach, this can be realised with an adaptive policy based management system, which dynamically adapts policy according to changes within the managed system, such as changes of user's current context in terms of his location, activities, device capabilities, etc.

Consider as an example a user using a mobile device for receiving voice, email and text messages. According to the current context in terms of the activity of the user, certain policies can be deployed to specify how incoming messages should be handled by the user's mobile device. Table A.1 presents examples of policies that can be used for handling the user's incoming messages.

Table A.1 Policies in a ubiquitous environment

User's Activity	Policy
In a meeting	"Convert voice message to text"
Driving	"Convert email to voice message"
At Home	"Convert text message to voice message"

Policies should be dynamically enabled with respect to changes of the user's activity. Example A.5 presents this type of adaptation can be implemented in our framework.

Example A.5 Policy for adapting policy for a mobile device when the user's activity changes

```
inst oblig PolicyForMobileDevice {
subject ServiceManagementAgent;
on ActivityIsChanged (newActivity)
do
      policies[] = select (newActivity) ->
      for each policy in policies[]
           policy.enable(); }
```

A more complicated scenario uses a ubiquitous system similar to the "Easyliving Lab" from Microsoft research [EasyLiving Project], a prototype of an architecture and technologies for ubiquitous computing. In this scenario, a person is using his mobile device within the intelligent environment, where three possibilities are offered: a) When the user is near his computer workstation, the display of the mobile device can be moved to the monitor of his computer and the user can use the keyboard and mouse of the computer, b) When the user is in his office (where the computer station is), the display of the mobile device can be moved to the projector; and, c) When the user is anywhere in the house, the display of the mobile device can be moved to the wall screen. Policies can be introduced within the system to define how the display of the mobile device is handled in each location. Table A.2 presents examples of policies for the scenario we described above.

Table A.2 Policies in an "smart" home

Location	Policy
Computer workstation	"Display on the computer monitor"
Office	"Display on the projector"
Anywhere in the house	"Display on the wall screen"

As we can observe, these three policies should not all be enabled for every location as they contain conflicting actions; rather policies should be dynamically enabled and triggered according to changes of the user's location. Assume that the user is entering in the intelligent environment using his mobile device. How the display of his device should be directed should be calculated at runtime, based on the current location of the user in the house. For example, if the user enters somewhere near his computer station, all three

choices are available. Either policy 1 from Table A.2 could be triggered, or policy 2, since the user is in the office, or policy 3, since the user is in the house. The method to calculate which policy should be enabled and triggered for the new location, can be implemented within a service management policy, as in the policy in Example A.6.

Example A.6 Policy for adapting policy for a mobile device when its location changes

```
inst oblig  PolicyForMobileDevice {
subject ServiceManagementAgent;
on locationIsChanged (newLocation)
do
      // policies[] is the set of all deployed policies
      policy = select (policies[], newLocation) ->
      policy.enable() -> policy.trigger(); }
```

An event conveying information about the new location of the user will trigger the policy PolicyForMobileDevice to invoke the action select, which will calculate which policy from Table A.2 should be enabled and triggered. An implementation of the selection algorithm can choose the "closest" policy in terms of the closest location of the display in the three dimensional space. Figure A.1 shows graphically the use of this selection algorithm. Since the user is closest to the center of the desk region, policy 1 will be enabled and triggered by the policy PolicyForMobileDevice so that the display is moved to the monitor on the computer desk.

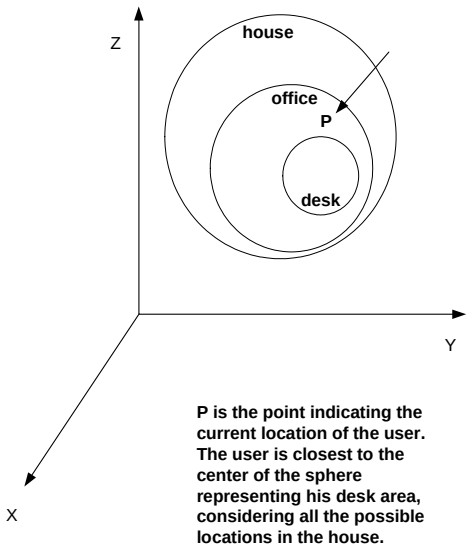


Figure A.1 Policy selection algorithm in a ubiquitous system