



Tampere University of Technology  
Department of Information Technology

Juha Majalainen

## **Implementation of Policy Management Tool for Bandwidth Provisioning**

Master of Science Thesis

Subject approved by the department council on 10.04.2002

Supervisors: Prof. Jarmo Harju

Dr. Tech Mika Grundström

# Foreword

This Master of Science thesis was done as part of the ICEFIN Research Laboratory at Tampere University of Technology, Institute of Communications Engineering during the years of 2001 - 2002. Current areas of interest in the ICEFIN project are Quality of Service, IPv6, security and multicast in IP networks.

I would like to thank Jussi Lemponen and Heikki Vatiainen for their invaluable contribution to the project and to this thesis and Prof Jarmo Harju for guidance. Thanks also to Juha Laine and Jussi Lemponen for providing the  $\LaTeX$  class file.

Finally, I would like to thank my parents, my family and my friends for their support throughout these years.

Tampere, December 3, 2002

Juha Majalainen

Vaajakatu 5 C 51

33720 Tampere

Finland

majis@cs.tut.fi

# Abstract

## TAMPERE UNIVERSITY OF TECHNOLOGY

Department of Information Technology

Institute of Communications Engineering

**Juha Majalainen:** Implementation of Policy Management Tool for Bandwidth Provisioning

Master of Science thesis: 59 pages

Supervisors: Prof Jarmo Harju and Dr. Tech Mika Grundström

December 2002

Keywords: COPS, DiffServ, LDAP, PIB, policy, policy-based networking, policy management tool, XML

This Master's thesis describes the implementation of Policy Management Tool (PMT) for bandwidth provisioning on the Linux operating system. The thesis was done as part of the ICEFIN project at Tampere University of Technology, Institute of Communications Engineering.

The goal of policy-based networking is to define a relationship between the needs of the network users and available resources. Every user in the policy-based network has a pre-defined high-level policy statement. Using policy statements, user can request available network resources. The policy statement specifies which kinds of quality of service (QoS) a network user is receiving at what times of the day.

In this thesis, the DiffServ architecture and mechanisms, developed by IETF, are used as the framework for QoS. The PMT implementation uses XML to define high-level policy statements and every policy statement is stored into an LDAP directory service and local tree hierarchy for later use. Resources allocated by the user of the policy-based network are distributed to policy enforcer using COPS protocol and special platform independent Policy Information Base (PIB).

# Tiivistelmä

## TAMPEREEN TEKNILLINEN KORKEAKOULU

Tietotekniikan osasto

Tietoliikennetekniikan laitos

**Juha Majalainen:** Implementation of Policy Management Tool for Bandwidth Provisioning

Diplomityö: 59 sivua

Tarkastajat: Prof. Jarmo Harju ja TkT Mika Grundström

Joulukuu 2002

Avainsanat: COPS, DiffServ, LDAP, PIB, policy, policy-based networking, policy management tool, XML

Tässä diplomityössä esitellään policy-based -verkon policy management tool (PMT) -konseptin toteutus Linux-käyttöjärjestelmään. Työ tehtiin osana ICEFIN -tutkimusprojektia Tampereen teknillisen korkeakoulun tietoliikennetekniikan laitoksella.

Policy-based -verkko on konsepti, jossa pyritään automaattisesti löytämään verkon vapaina olevat resurssit ja varaamaan näitä resursseja verkon käyttäjien kesken tehokkaasti. Jokaiselle verkon käyttäjälle on etukäteen määritelty säännöt, joiden avulla vapaita resursseja varataan. Näiden sääntöjen avulla voidaan dynaamisesti määrittellä minkälaista palvelun laatua käyttäjä voi verkolta saada ja mihin aikaan.

Tässä diplomityössä palvelunlaadulla tarkoitetaan IETF:n kehittämää DiffServ-mekanismia, jota voidaan käyttää palvelunlaadun tarjoamiseen IP-verkoissa. PMT -toteutuksessa resurssien varaamiseen tarvittavat säännöt on määritelty XML-kielen avulla ja nämä säännöt on talletettu paikallisiin tietorakenteisiin ja LDAP-hakemistopalvelimelle myöhempää käyttöä varten. Käyttäjien varaamat resurssit levitetään verkon aktiivilaitteille COPS-protokollan ja laiteriippumattomien Policy Information Base (PIB) -taulujen avulla.

Policy management tool -toteutuksen avulla voidaan helposti hallita ja määrittellä policy-based -verkon resursseja. Policy-based -verkon resurssien varaaminen toimii hyvin hallinnollisesti saman verkon sisällä, mutta useiden hallinnollisesti eri verkkojen välillä resurssien hallinta on erittäin monimutkaista teknisistä ja poliittisista syistä johtuen.

# Table of Contents

<b>Foreword</b> . . . . .	i
<b>Abstract</b> . . . . .	ii
<b>Tiivistelmä</b> . . . . .	iii
<b>Table of Contents</b> . . . . .	iv
<b>List of Acronyms</b> . . . . .	vii
<b>1 Introduction</b> . . . . .	1
<b>2 Differentiated Services</b> . . . . .	3
2.1 Introduction . . . . .	3
2.2 Differentiated Service field . . . . .	4
2.3 Architecture . . . . .	5
2.4 Per-Hop Behaviors . . . . .	6
2.4.1 Default PHB . . . . .	6
2.4.2 Expedited Forwarding PHB . . . . .	7
2.4.3 Assured Forwarding PHB . . . . .	8
<b>3 Extensible Markup Language</b> . . . . .	9
3.1 Introduction . . . . .	9
3.2 Document Type Definition . . . . .	10
3.3 XML Elements . . . . .	11
3.3.1 Element type ANY . . . . .	12
3.3.2 Element type EMPTY . . . . .	12
3.3.3 Character element type . . . . .	13
3.3.4 Child elements . . . . .	13
3.3.5 Sequence list . . . . .	13
3.3.6 Choice list . . . . .	14
3.3.7 Mixed content . . . . .	14
3.3.8 Cardinality operators . . . . .	15
3.4 XML Attributes . . . . .	16
3.4.1 ATTLIST declaration . . . . .	16
3.4.2 Attribute default values . . . . .	17
3.5 XML declaration components . . . . .	18
3.6 Simple DTD example . . . . .	19
<b>4 Lightweight Directory Access Protocol</b> . . . . .	20
4.1 Introduction . . . . .	20
4.2 History . . . . .	20
4.3 Directory Entry . . . . .	21
4.4 Directory Service . . . . .	21
4.5 Directory Information Tree . . . . .	22
4.6 LDAP Client-Server Model . . . . .	23

4.7	LDAP Operations . . . . .	24
4.8	Schema Specification . . . . .	25
4.8.1	Attribute Type Specification . . . . .	25
4.8.2	Object Class Specification . . . . .	26
4.8.3	Subschema . . . . .	27
4.9	LDAP Security . . . . .	27
4.9.1	Authentication . . . . .	27
4.9.2	Authorization . . . . .	27
4.10	Directory Enabled Network . . . . .	28
<b>5</b>	<b>Policy-Based Networking . . . . .</b>	<b>29</b>
5.1	Introduction . . . . .	29
5.2	Policy Terminology . . . . .	30
5.2.1	High-Level policy . . . . .	31
5.2.2	Low-Level policy . . . . .	31
5.3	Architectural Elements . . . . .	32
5.3.1	Policy Management Tool . . . . .	33
5.3.2	Policy Repository . . . . .	34
5.3.3	Policy Decision Point . . . . .	35
5.3.4	Policy Enforcement Point . . . . .	36
5.4	IETF's Policy Schema . . . . .	37
<b>6</b>	<b>Policy Provisioning using COPS . . . . .</b>	<b>39</b>
6.1	Introduction . . . . .	39
6.2	Common Open Policy Service . . . . .	40
6.3	Common COPS operations . . . . .	41
6.4	Policy Information Base . . . . .	42
6.5	COPS Usage for Policy Provisioning . . . . .	43
6.6	The COPS-PR Message Content . . . . .	44
6.6.1	Request . . . . .	44
6.6.2	Decision . . . . .	44
<b>7</b>	<b>Policy Management Tool implementation . . . . .</b>	<b>45</b>
7.1	Introduction . . . . .	45
7.2	PBN Architecture Implementation Overview . . . . .	47
7.3	XML to LDAP Mapping . . . . .	48
7.4	Local Policy Tree . . . . .	50
7.5	Scheduler . . . . .	55
7.6	Test Network . . . . .	57
<b>8</b>	<b>Conclusions . . . . .</b>	<b>58</b>
	<b>References . . . . .</b>	<b>60</b>
 <b>Appendices</b>		
<b>A</b>	<b>LDAP Policy Schema . . . . .</b>	<b>63</b>
A.1	Schema attributes . . . . .	63

A.2	Schema objectclass . . . . .	65
A.2.1	BandwidthAllocationRequest . . . . .	65
A.2.2	ServiceLevelAgreement . . . . .	65
<b>B</b>	<b>XML Policy DTD . . . . .</b>	<b>66</b>
B.1	BandwidthAllocationRequest . . . . .	66
B.2	ServiceLevelAgreement . . . . .	66
B.3	top . . . . .	67
<b>C</b>	<b>Example XML Data . . . . .</b>	<b>68</b>
<b>D</b>	<b>Localtree C structures . . . . .</b>	<b>69</b>
D.1	Struct bar . . . . .	69
D.2	Struct pep . . . . .	69
D.3	Struct sla . . . . .	70
D.4	Struct times . . . . .	70
D.5	Struct event . . . . .	70
D.6	Struct sls . . . . .	71
<b>E</b>	<b>main_loop . . . . .</b>	<b>73</b>
<b>F</b>	<b>Example script for EF Edge . . . . .</b>	<b>76</b>
<b>G</b>	<b>Example script for EF Core . . . . .</b>	<b>77</b>

# List of Acronyms

<b>AF</b>	Assured Forwarding
<b>ARP</b>	Address Resolution Protocol
<b>ASN.1</b>	Abstract Syntax Notation One
<b>ATM</b>	Asynchronous Transfer Mode
<b>BA</b>	Behavior Aggregate
<b>BB</b>	Bandwidth Broker
<b>BER</b>	Basic Encoding Rule
<b>BE</b>	Best-Effort
<b>BNF</b>	Backus-Naur Form
<b>BN</b>	Boundary Node
<b>CBQ</b>	Class Based Queueing
<b>COPS-PR</b>	COPS Usage for Policy Provisioning
<b>COPS</b>	Common Open Policy Service Protocol
<b>CS</b>	Class Selector
<b>DEN</b>	Directory Enabled Networks
<b>DiffServ</b>	Differentiated Services
<b>DN</b>	Distinguished Name
<b>DSCP</b>	Differentiated Services Codepoint
<b>dsmark</b>	Differentiated Services Field Marker
<b>DS</b>	Differentiated Services
<b>DTD</b>	Document Type Definition
<b>EF</b>	Expedited Forwarding
<b>FIFO</b>	First In First Out

<b>FTP</b>	File Transfer Protocol
<b>GRED</b>	Generalized Random Early Detection
<b>HMAC</b>	Keyed-Hashing Message Authentication
<b>HTML</b>	Hypertext Markup Language
<b>ICMP</b>	Internet Control Message Protocol
<b>IETF</b>	Internet Engineering Task Force
<b>IntServ</b>	Integrated Services
<b>IN</b>	Interior Node
<b>IPSec</b>	Internet Protocol Security
<b>IP</b>	Internet Protocol
<b>ISP</b>	Internet Service Provider
<b>LDAP</b>	Lightweight Directory Access Protocol
<b>LDAP</b>	Lightweight Directory Access Protocol
<b>LPDP</b>	Local Policy Decision Point
<b>MAC</b>	Media Access Control
<b>MD5</b>	Message Digest 5
<b>MIB</b>	Management Information Base
<b>MPLS</b>	Multiprotocol Label Switching
<b>MTU</b>	Maximum Transmission Unit
<b>nfmark</b>	Netfilter Mark
<b>OSPF</b>	Open Shortest Path First
<b>PBN</b>	Policy-Based Networking
<b>PDB</b>	Per Domain Behavior
<b>PDP</b>	Policy Decision Point
<b>PEP</b>	Policy Enforcement Point

<b>PHB</b>	Per-Hop Behavior
<b>PIB</b>	Policy Information Base
<b>PIN</b>	Policy-Ignorant Node
<b>PMT</b>	Policy Management Tool
<b>PPRID</b>	Prefix Provisioning Instance Identifier
<b>PQ</b>	Priority Queueing
<b>PRC</b>	Provisioning Class
<b>PRID</b>	Provisioning Instance Identifier
<b>PRI</b>	Provisioning Instance
<b>qdisc</b>	Queueing Discipline
<b>QoS</b>	Quality of Service
<b>RAA</b>	Resource Allocation Answer
<b>RAR</b>	Resource Allocation Request
<b>RDN</b>	Relative Distinguished Name
<b>RED</b>	Random Early Detection
<b>RFC</b>	Request For Comments
<b>RR</b>	Round Robin
<b>RSVP</b>	Resource Reservation Protocol
<b>SFQ</b>	Stochastic Fair Queueing
<b>SGML</b>	Standard Generalized Markup Language
<b>SIBBS</b>	Simple Interdomain Bandwidth Broker Signalling
<b>SLA</b>	Service Level Agreement
<b>SLS</b>	Service Level Specification
<b>SNMP</b>	Simple Network Management Protocol
<b>SPPI</b>	Structure of Policy Provisioning Information

<b>SQL</b>	Structured Query Language
<b>TBF</b>	Token Bucket Filter
<b>TB</b>	Token Bucket
<b>TCB</b>	Transmission Control Block
<b>TCP</b>	Transmission Control Protocol
<b>ToS</b>	Type of Service
<b>UDP</b>	User Datagram Protocol
<b>UML</b>	Unified Modelling Language
<b>VoIP</b>	Voice over IP
<b>WFQ</b>	Weighted Fair Queueing
<b>WRR</b>	Weighted Round Robin
<b>XML</b>	Extensible Markup Language

# 1 Introduction

Exponential growth of the Internet and bandwidth hungry applications presents a real challenge to network administrators and service providers who must provide reliable bandwidth and guaranteed service levels to network users in an environment of finite resources.

The ability to identify users on the network and give them controlled access to network resources is generally referred to as policy-based networking. The basic policy-based network architecture is defined in RFC 2753 [1].

This Master's thesis describes the implementation of Policy Management Tool (PMT) for bandwidth provisioning. According to IETF (Internet Engineering Task Force) the policy management tool is responsible for such functions as policy editing, policy representation, rule translation, rule validation and conflict resolution. The goal of this thesis was to examine a policy-based network by implementing a policy management tool client and testing it as part of a policy-enabled network. The thesis was done as part of the ICEFIN project at Tampere University of Technology, Institute of Communications Engineering.

The structure of this thesis is as follows. A brief introduction to quality of service, general concepts of differentiated services and different Per-Hop Behaviors (PHBs) are presented in Chapter 2.

Chapter 3 describes Extensible Markup Language (XML) and gives an introduction to elements and attributes used to define a Document Type Definition (DTD). The Policy Management Tool implementation uses policy DTD and XML data to represent high-level poli-

cies.

Chapter 4 describes Lightweight Directory Access Protocol (LDAP) and LDAP directory service. LDAP section also gives a brief introduction to elements and attributes used to define new LDAP schemas. The PMT implementation uses LDAP directory service to store high-level policies.

The policy-based networking framework designed by IETF's RAP working group is depicted in Chapter 5, and Chapter 6 describes COPS (Common Open Policy Service) protocol developed in the RAP working group. The COPS is a protocol for exchanging network policy information between different policy-based network elements.

An implementation of the architecture in general, and specifically the implementation of the Policy Management Tool are described in Chapter 7. Finally, Chapter 8 summarizes the thesis.

## 2 Differentiated Services

### 2.1 Introduction

Present-day Internet only provides best-effort service. Traffic is processed as quickly as possible but there is no guarantee as to timeliness or actual delivery. With the transformation of the Internet into a commercial infrastructure, demands for service quality have rapidly developed.

Today's IP networks are large complex systems consisting of many different services. New multimedia applications need real-time performance guarantees such as bounded delay and minimum throughput. The current Internet does not support these Quality Of Service (QoS) parameters and it is a real challenge to integrate them in the existing architecture.

The differentiated services (DiffServ) architecture proposes the use of a well-defined set of building blocks from which a variety of services may be built. The DiffServ architecture was designed by the Internet Engineering Task Force's (IETF) DiffServ working group [2]. DiffServ Architecture is defined in RFC 2474 [3] and RFC 2475 [4]. In this architecture, each packet carries an information (DiffServ byte) used by each hop to give it a particular forwarding treatment in a DiffServ-aware router. DiffServ mechanisms allow network providers to allocate different levels of service to different users of the Internet.

## 2.2 Differentiated Service field

Currently the IP version 4 (IPv4, RFC 791 [5]) header includes a Type of Service field (ToS, Figure 2.1) intended to indicate the Quality of Service (QoS) Internet packets should receive. QoS quantifies delay, throughput and reliability that an IP or packet stream is receiving.

0	1	2	3	4	5	6	7
precedence			D	T	R	C	0

Figure 2.1: The ToS octet as defined in RFC 1349 [6]

The first 3 bits, labeled *precedence* in Figure 2.1, are assigned to represent the packet's precedence (relative priority), the next 4 bits, labeled D,T,R,C in Figure 2.1, indicate the desired type in routing/forwarding service. ToS bits are marked in the table as follows: D for minimize delay, T for maximize throughput, R for maximize reliability and C for minimize cost. The last bit was specified to be always zero.

0	1	2	3	4	5	6	7
DSCP						CU	

Figure 2.2: The DiffServ field defined in RFC 2474 [3]

In DiffServ, the 8-bit Type of Service (ToS) field in the IPv4 header is replaced by the Differentiated Services field (DiffServ field, Figure 2.2). The DiffServ field is defined in RFC 2474 [3]. The DiffServ field contains a value that DiffServ-enabled routers use to determine a specific forwarding treatment at each node along a traffic path. The treatment given to a packet with a particular DiffServ codepoint is called a per-hop behavior (PHB, see section 2.4) and is administered independently at each network node. The first six bits of the DiffServ field contain the DiffServ codepoint (DSCP). The DSCP indicates the PHB used at each DiffServ-aware node. The last two bits in the DiffServ field are designated Currently Unused (CU), and support non-DiffServ-enabled devices that use the original ToS byte to determine the forwarding treatment.

## 2.3 Architecture

A network using particular DiffServ mechanisms to provide edge-to-edge QoS is referred to as a differentiated services domain (*DiffServ domain*). A differentiated services domain can represent different administrative domains or autonomous systems, different trust regions, different network technologies, hosts and routers, etc. Edge routers surrounding a DiffServ domain are known as DiffServ *Boundary nodes*. Core routers providing transit service are known as DiffServ *Interior nodes*. Figure 2.3 depicts this architecture. Boundary nodes are pictured with the symbol *BN*, interior nodes with *IN*, Non-DiffServ nodes with *N* and customer equipment with *CE*. Customer equipment is a common term for servers, desktop computers, VoIP phones etc.

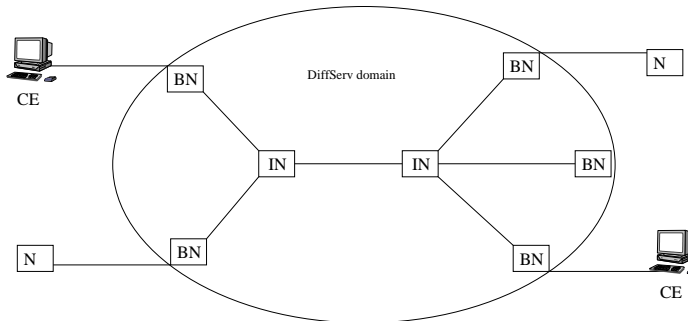


Figure 2.3: DiffServ – basic architecture

DiffServ boundary nodes sit at the edges of a DiffServ domain. DiffServ boundary nodes function as both DiffServ ingress and egress nodes for different directions of traffic flows (Figure 2.4). When functioning as a DiffServ ingress node, a DiffServ boundary node is responsible for the classification, marking, and possibly conditioning of ingress traffic. It classifies each packet based on an examination of the packet header, and then writes the DSCP to indicate one of the PHB supported within the DiffServ domain.

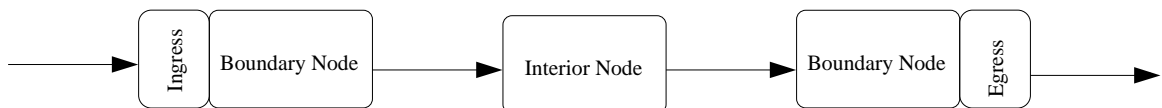


Figure 2.4: Ingress and Egress nodes

DiffServ interior nodes select the forwarding behavior applied to each packet, based on an examination of the packet's DSCP. DiffServ interior nodes map the DSCP to one of the PHB groups supported by all of the DiffServ interior nodes within the DiffServ domain.

## 2.4 Per-Hop Behaviors

A per-hop behavior (PHB) is a description of the externally observable forwarding behavior of a DiffServ node applied to a particular DiffServ behavior aggregate [3]. Behavior Aggregate is a collection of packets with the same codepoint crossing a link in a particular direction. The concept of forwarding behavior can be interpreted to mean those aggregate actions that interior nodes perform with packets having a similar codepoint in the DiffServ field. The DiffServ architecture supports the delivery of scalable service discrimination based on this hop-by-hop resource allocation mechanism. PHBs are usually needed in cases when several behavior aggregates are competing on resources in a node, which is then able to make service discrimination based on the defined PHBs in that DiffServ node.

The DiffServ working group has defined three PHBs that have a recommended codepoint for mapping. The Expedited Forwarding (EF) PHB, the Assured Forwarding (AF) PHB group, and the Default (BE) PHB.

The EF PHB can be used to implement a low latency, low jitter, low loss, end-to-end service such as a virtual leased line (VLL). AF is a family of PHBs, called a PHB group, that is used to classify packets into various drop precedence levels. The drop precedence assigned to a packet determines the relative importance of a packet within the AF class. The BE PHB is the traditional best-effort service model.

### 2.4.1 Default PHB

RFC 1812 [7] specifies the default PHB as the conventional best-effort (BE) forwarding behavior. When no other agreements are in place, all packets are assumed to belong to this traffic aggregate. A packet assigned to this aggregate may be sent into a network without following any specific rules, and the network will deliver as many of these packets as fast as possible. The recommended DSCP for the default PHB is 000000b.

## 2.4.2 Expedited Forwarding PHB

RFC 3246 [8] specifies the EF PHB. The EF approach is to divide packets into two classes. Most packets are forwarded using the best effort available under current network conditions. A small subset of network traffic is given a special EF codepoint. This subset of traffic provides the highest QoS possible. Expedited Forwarding PHB is suggested for applications that require a hard guarantee on the delay and jitter. Typically mission critical applications such as Voice over IP (VoIP), video, and online trading applications would require this service.

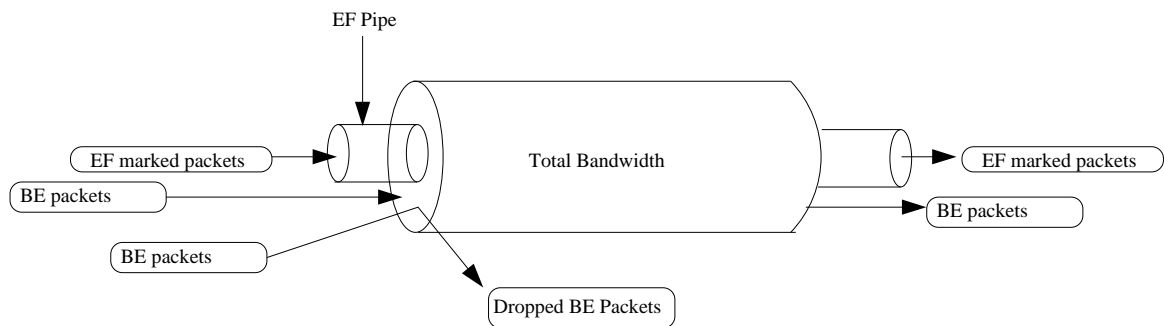


Figure 2.5: Virtual leased line using EF

The EF PHB is designed to provide *low loss*, *low delay*, *low jitter* and *assured bandwidth* end-to-end service. In effect, the EF PHB simulates a virtual leased line to support highly reliable voice or video and to emulate dedicated circuit services.

Figure 2.5 depicts a virtual leased line using EF. All the packets marked with EF PHB are forwarded and some of the BE packets on the network are dropped to make room for EF marked packets when congestion arises.

The Expedited Forwarding treatment polices and drops packets on a network boundary nodes' ingress interface, and shapes traffic on egress interface to make sure that the connection to the next provider is at the same priority level. The recommended DSCP for the EF PHB is 101110b.

### 2.4.3 Assured Forwarding PHB

The Assured Forwarding PHB Group is defined in RFC 2597 [9]. Assured forwarding groups packets into one of four classes. Each class has three drop precedence levels, *low*, *medium* and *high*. Each class or type of traffic is independent of the other classes and can have its own unique drop precedence. In case of congestion, the drop precedence of a packet determines the relative importance of the packet within the AF class. A congested DiffServ node tries to protect packets with a lower drop precedence value from being lost by preferably discarding packets with a higher drop precedence value.

AF provides forwarding of IP packets in four independent AF classes. Within each AF class, an IP packet is assigned one of three different levels of drop precedence. An IP packet that belongs to an AF class  $i$  and has drop precedence  $j$  is marked with the AF codepoint  $AF_{ij}$ , where  $1 \leq i \leq 4$  and  $1 \leq j \leq 3$ . Currently, four classes with three levels of drop precedence in each class are defined for general use. Table 2.1 describes the codepoints defined for AF PHB.

Drop precedence	<b>class 1</b>	<b>class 2</b>	<b>class 3</b>	<b>class 4</b>
<b>low</b>	001010b	010010b	011010b	100010b
<b>medium</b>	001100b	010100b	011100b	100100b
<b>high</b>	001110b	010110b	011110b	100110b

Table 2.1: AF PHB Codepoints

## 3 Extensible Markup Language

### 3.1 Introduction

Extensible Markup Language (XML) [10] is a subset of the Standardized Generalized Markup Language (SGML) [11]. Markup languages (such as HTML, XML, or SGML) are designed to add structure and convey information about documents and data. XML provides an application independent way of sharing data. In markup languages, the main mechanism for supplying structural and semantic information is by decorating the document with *elements* including a *start tag*, optionally some content, and an *end tag*. For example the tags used to markup HTML documents and the structure of HTML documents are predefined. XML allows the author to define his own tags and his own document structure using XML building blocks.

An XML document primarily consists of a strictly nested hierarchy of elements with a single root. Elements can contain character data, child elements, or a mixture of both (see Section 3.3). In addition, they can have attributes (see Section 3.4). Child character data and child elements are strictly ordered; attributes are not. Section 3.2 describes how to create a new grammar (known as a Document Type Definition (DTD) ) for XML documents using elements and attributes. The names of the elements and attributes and their order in the hierarchy form the XML markup language used by the document (see example in Section 3.6). This language can be defined by the document author or it can be inferred from the document's structure.

## 3.2 Document Type Definition

The XML document grammar is described using a *Document Type Definition* (DTD) [10]. The DTD describes allowable elements and attributes in the XML document. Elements are described in Section 3.3 and attributes are described in Section 3.4. The DTD also describes the structure of those elements and attributes. An XML document that is structured according to the rules defined in the XML specification is termed *well formed*. An XML document is well-formed if it contains at least one unique root element which contains the whole document and all the tags are correctly nested. The root element also must have correct opening and closing tags. In addition all the tags and attributes must conform to the rules for writing tags, and all the values of the attributes must be quoted. In addition to being well formed, an XML document can also be *valid*. A valid XML document must contain a DTD, and the grammar of the document must conform to that specified in the DTD. The process of testing to make sure that an XML document conforms to the description of the grammar described in the DTD is commonly termed *validation*.

A DTD can be either internal or external. An internal DTD actually places the structure within the XML document. While this makes it very simple to validate the document, the DTD is not reusable – it must be included with every XML document. An external DTD is stored separately from the XML document, and the document points to the DTD via a line at the beginning of the XML document.

When using internal DTD's the DTD must be placed between the XML declaration and the first element (root element) in the document. The keyword DOCTYPE must be followed by the name of the root element in the XML document and keyword DOCTYPE must be in upper case (see example in Section 3.6).

An external DTD is stored separately from the XML document. The XML document points to the external DTD via a DOCTYPE line at the beginning of the XML document. External DTD's are useful for creating a common DTD that can be shared between multiple documents (see Appendix B). Any changes that are made to the external DTD automatically updates all the documents that reference it. Internal DTD declarations have priority over external DTD declarations. Each XML document can be associated with exactly one DTD using DOCTYPE declaration.

### 3.3 XML Elements

XML consists of many building blocks such as elements and attributes. Elements are the fundamental building blocks of XML data. Elements are logical units of information in a DTD – they represent information objects. Elements either contain information (text), or have a structure of subelements. Any useful document will include several element types, some nested within others using *choice lists*, *sequence lists* and *cardinality operators*. Choice lists are described in Section 3.3.6, sequence lists in Section 3.3.5 and cardinality operators in Section 3.3.8.

In the DTD, XML elements are declared with an element declaration. Figure 3.1 depicts an element declaration syntax.

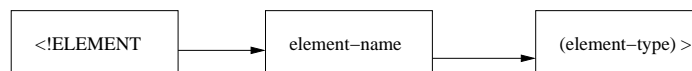


Figure 3.1: Element Declaration

*Element-name* is the name of the element in XML document and *element-type* represent a piece of information for that element. Element declaration must end with the “>” character.

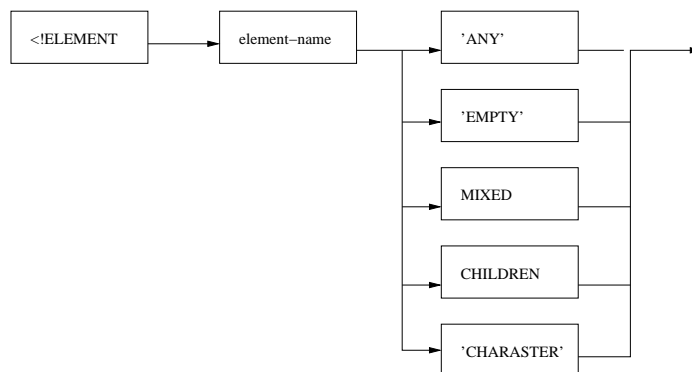


Figure 3.2: Element Declarations

Element may have different element types (see Figure 3.2). Usually element type is either EMPTY (see Figure 3.4) or character element type (PCDATA/CDATA). Element type EMPTY is described in Section 3.3.2 and character element type is described in Section 3.3.3. Element may also have child elements. Child element type is described in Section 3.3.4. Other element types are ANY (see Section 3.3.1) and mixed content (see Section 3.3.7).

### 3.3.1 Element type ANY

Element type ANY may contain any well-formed XML data. Element type ANY means that the element can contain zero or more child elements of any declared type, as well as character data.

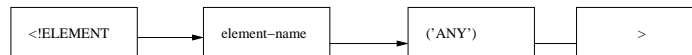


Figure 3.3: Element type ANY

Figure 3.3 depicts element type ANY declaration. Element type ANY can include character data, other elements, comments and anything else as long it is well-formed. Element type ANY should be avoided because it entirely defeats the purpose of using a DTD.

### 3.3.2 Element type EMPTY

Sometimes an element has no data. Element type that does not contain any data is called EMPTY element. Figure 3.4 depicts EMPTY element declaration. EMPTY element means that the element has no child elements or character data and it is used only as a holder for attributes.



Figure 3.4: Element type EMPTY

Empty elements can be used for configuration files that use name-value pairs. For example:

```
<!ELEMENT BandwidthAllocationRequest EMPTY>

<!ATTLIST BandwidthAllocationRequest
  SlaId CDATA #REQUIRED
  BarId CDATA #REQUIRED
  ...>
```

Element *BandwidthAllocationRequest* is declared as EMPTY element type but it is also used as a holder for attributes *SlaId* and *BarId*. Attributes are described in Section 3.4.

### 3.3.3 Character element type

Character element type is one of the fundamental data types in XML. Character types can be parsed by the parser (PCDATA) or ignored by the parser (CDATA). PCDATA stands for *parsed character data* and CDATA stands for *character data* that contains no mark-up.

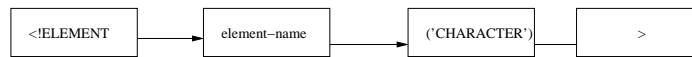


Figure 3.5: Character element type

### 3.3.4 Child elements

Complex XML documents also need child elements. Child element types are declared using parentheses in the parent element type's declaration. For example:

```
<!ELEMENT parent_name (child_name)>
<!ELEMENT child_name allowable_contents>
```

In this example there must be one child element and the child element must contain some well-formed data. The child element must be declared in a separate element type declaration. Child elements may be constrained to appear in a specific sequence (*sequence list*, see Section 3.3.5) and/or limited to list of several mutually exclusive choices (*choice list*, see Section 3.3.6). The number of occurrences of a given child element may be specified using single character *cardinality operator*. Cardinality operators are described in Section 3.3.8.

### 3.3.5 Sequence list

Sequence list can be used separating child elements with comma (,) operator.

```
<!ELEMENT parent_name (child1_name,child2_name)>
<!ELEMENT child1_name allowable_contents>
<!ELEMENT child2_name allowable_contents>
```

In the example above, there must be two child elements, and they must always appear in the sequence shown in the declaration.

### 3.3.6 Choice list

XML document can also use choice lists. Choice lists are described using vertical bar (|).

Using choice list, only one of several child elements is permitted.

```
<!ELEMENT parent_name (child1_name|child2_name)>
<!ELEMENT child1_name allowable_contents>
<!ELEMENT child2_name allowable_contents>
```

In the example above, there may be two child elements. In this case the child elements' order is not important. The parent element can consists child1 element or child2 element.

Combining choice lists and sequence lists is also permitted. Using both of these two types provides a very powerful and complex content model for describing elements in XML. These combinations can be nested using additional pairs of parentheses but parentheses must always be matched in pairs.

### 3.3.7 Mixed content

Mixed content can be used using character element type and child elements together. If character element is present, the element type is either mixed or character element type. The absence of character element type indicates that element type is element only.

```
<!ELEMENT element_name (character_type | child1_name | child2_name)>
```

In the example above element *element\_name* is declared using choice list, character element and child elements. If element is declared as *mixed content*, the declaration must place character element type first in the group and the group must be a choice list. Such groups are generally referred to as *mixed content* (as opposed to element-only groups or *element content*). Technically, mixed content refers to any element containing character data. A mixed element is of very limited usefulness. It is usually better to place the text in an attribute and use a standard element declaration as described above.

### 3.3.8 Cardinality operators

Cardinality operators define how many child elements may appear in a declaration. In XML, cardinality operators are *question mark* (?), *asterisk* (\*) and *plus* (+).

*Question mark* indicates that a child element is optional and single-valued (zero to one) which means that the child element can appear at most once, but may not appear at all.

*Asterisk* indicates that the child element is optional and multi-valued (zero to many) which means that the child element can appear multiple times, but may not appear at all.

*Plus* indicates that the child element is mandatory and multi-valued (1 to many) which means that the child element may appear multiple times, but must appear at least once.

The absence of cardinality operator means that there must always be exactly one of each of a child elements. This means that all child elements are required. Sequence lists and/or choice lists can be used together with cardinality operators.

```
<!ELEMENT A ( (B|C)?, D+, E*, F )>
<!ELEMENT B #PCDATA>
<!ELEMENT C #PCDATA>
<!ELEMENT D #PCDATA>
<!ELEMENT E #PCDATA>
<!ELEMENT F #PCDATA>
```

The example shown above uses cardinality operators, choice lists and sequence lists. Element A contains child elements B,C,D,E and F. Child elements B and C are described in a choice list and there may be only zero or one of elements B and C. In this case the child elements' order is not important because we are using choice list. The rest of the elements are in a sequence list and in this case the child elements order must always appear in the sequence shown in the declaration. There must be one or more child element D, zero or more child element E and one and only one of child element F. So combination of choice lists, sequence lists and cardinality operators provides for very powerful and complex element declaration.

## 3.4 XML Attributes

XML attributes are normally used to describe XML elements, or to provide additional information about elements. XML elements can have attributes only in the start tag. Attributes can be used to describe the meta-data or properties of the associated element. Attributes are also an alternative way to markup document data. Element attributes are described using the *ATTLIST declaration* (see Figure 3.6).

### 3.4.1 ATTLIST declaration

In the DTD, XML element attributes are declared with an ATTLIST declaration. Figure 3.6 depicts an attribute declaration syntax.

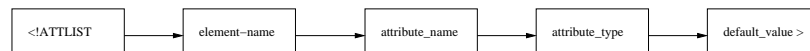


Figure 3.6: ATTLIST Declaration [10]

ATTLIST declaration defines the element which can have the attribute, the name of the attribute, the type of the attribute, and the default attribute value. XML attribute types are described in Table 3.1 and default values are described in Section 3.4.2. More detailed information for attribute defaults and attribute types are described in XML standard [10].

Values	Description
CDATA	The value is character data
(eval eval ..)	The value must be an enumerated value
ID	The value is an unique id
IDREF	The value is the id of another element
IDREFS	The value is a list of other id's
NMTOKEN	The value is a valid XML name
NMTOKENS	The value is a list of valid XML names
ENTITY	The value is an entity
ENTITIES	The value is a list of entities
NOTATION	The value is a name of a notation
xml:	The value is predefined

Table 3.1: Attribute type values [10]

### 3.4.2 Attribute default values

There are four different attribute defaults in XML.

#### #DEFAULT

means that the attribute must have a value every time this element is listed. Although there is no default value provided by the DTD, the attribute when actually implemented in an XML document must define a value.

#### #FIXED

Sometimes it is wanted to provide a default value that the document author may not modify. In that case, #FIXED attribute default can be used. An attribute declaration may specify that an attribute has a fixed value. In this case, the attribute is not required, but if it occurs, it must have the specified value.

Values	Description
#DEFAULT value	The attribute has a default value
#REQUIRED	The attribute value must be included in the element
#IMPLIED	The attribute does not have to be included
#FIXED value	The attribute value is fixed

Table 3.2: Attribute default values [10]

#### #IMPLIED

The attribute value is not required, and no default value is provided. If a value is not specified, the XML processor must proceed without one. The processor ignores this attribute unless it is used as part of this element. It does not assume any default value.

#### #FIXED value

An attribute can be given any legal value as a default. The attribute value is not required on each element in the document, but if it is not present, it will appear to be the specified default. Value (default values) provides a default value for that attribute. If the attribute is not included in the element, the processing program assumes that this is the attributes value.

## 3.5 XML declaration components

Every XML document should use an XML declaration, to state its nature to XML document readers. XML declaration components are described in Table 3.3. Editors, browsers and document processors use the declaration to determine how a document should be processed. The declaration becomes increasingly important with large and complex documents but should also be used in smaller and experimental documents. The XML declaration includes information on markup language, markup language version, presence of external markup declarations and character encoding.

Component	Description
<?xml	Starts the XML declaration.
version	XML 1.0 is the current and only version of XML.
standalone	Specify whether external markup declarations may exist.
encoding	Specify the character encoding.
?>	Closes the XML declaration.

Table 3.3: XML declaration components [10]

```
<?xml version="1.0" standalone="no"?>
```

In the example above xml version 1.0 is used and external markup declarations may exist. Currently XML version 1.0 is the only version of XML. XML declaration must be situated at the first position of the first line in the XML document and the version number attribute is mandatory. If other attributes are declared (*standalone* or *encoding*) in an XML declaration, they must be placed so that *standalone* attribute is first and *encoding* attribute is next (attribute order can shown in Table 3.3).

Most common encoding character sets are:

UTF8, UTF16, ISO10646UCS2, ISO10646UCS4, ISO88591 to ISO88599, ISO2022JP, Shift\_JIS, EUCJP.

For a full list of encodings check the IANA's website [12].

More detail information for XML declaration components is described in XML standard [10]

## 3.6 Simple DTD example

This is a very simple example of DTD data within the doctype declaration:

```
<!DOCTYPE top [  
<!ELEMENT top ( ServiceLevelAgreement )>  
<!ELEMENT ServiceLevelAgreement (#PCDATA)>  
>
```

1. `!DOCTYPE` is the tag for starting a document type declaration, which specifies the type of document that is validating against. It contains either the validation data or a pointer to the location of that data.
2. `top` is the name that is given this type of document
3. `[` announces the beginning of DTD data
4. `!ELEMENT` starts the element definition.
5. `ServiceLevelAgreement` is the name given to the element. This will be the tag used in XML document.
6. `(#PCDATA)` describes the type of data that will be contained in the element. In this example, the data will be "Parsed Character Data" – `#PCDATA`. This basically means textual content.
7. `]` ends the set of DTD data.
8. `>` ends the `!DOCTYPE` tag.

In the DTD shown above, the language contains two elements: *top* and *ServiceLevelAgreement*. The *top* element (root element) contains a single *ServiceLevelAgreement* element.

The same example using external DTD can be made for example:

```
<?xml version="1.0"?>  
<!DOCTYPE top SYSTEM "policy.dtd">  
<top>  
<ServiceLevelAgreement SlaId="FASTER_DEMO"  
</top>
```

where *policy.dtd* consist declaration for elements *top* and *ServiceLevelAgreement*

The external DTD used in Policy Management Tool (*policy.dtd*) is much more complex and it is defined in Appendix B. The main reason to explicitly define the document type definition (DTD) is that documents can be checked to conform to it. For example, if a document type definition is defined for the *top*, authors using this DTD could use a validating parser to ensure that their documents are similar to the document type definition.

## 4 Lightweight Directory Access Protocol

### 4.1 Introduction

The Lightweight Directory Access Protocol (LDAP) is an open standard protocol for accessing on-line directory services. Currently, the LDAP has two versions, version 2 (v2) and version 3 (v3). The LDAP v2 is the original version when LDAP was developed. It is defined in the RFC1777 [13] specification. The LDAP v3, defined in RFC2251 [14], is designed to provide more security (see Section 4.9) and other functions that the LDAP v2 lacks. LDAP runs directly over Transmission Control Protocol (TCP), and can be used to access a stand-alone LDAP directory service or to access a directory service that is back-ended by X.500 (see Section 4.2).

### 4.2 History

LDAP was originally developed as a front end to X.500, the OSI directory service. X.500 defines the Directory Access Protocol (DAP) for clients to use when contacting directory servers. DAP is a heavyweight protocol that runs over a full OSI stack and requires a significant amount of computing resources to run. LDAP runs directly over TCP and provides most of the functionality of DAP at a much lower cost. This use of LDAP makes it easy to access the X.500 directory.

## 4.3 Directory Entry

The LDAP information model is based on directory entries. *Entry* is the basic building block of a directory. Each directory entry contains one or more attributes and an *attribute* (see Section 4.8.1) contains type and one or more values. The types are typically mnemonic strings, like *cn* for common name, or *mail* for email address. The values depend on what type of attribute it is.

LDAP Directories can hold information on entries of any kind. *Objectclass* is a special attribute that tells the LDAP server which attributes are required and which attributes may be allowed in a particular LDAP entry. The objectclass attribute supports the object-oriented concept of inheritance in that you can create a new objectclass that inherits all of the required and allowed attributes of its parent.

Typical information about the entries includes email addresses, fax and telephone numbers, photographs, security information (public key certificates and passwords), as well as an object's capabilities and the policies that control them. Table 4.1 depicts an LDAP entry.

<b>Name</b>	<b>Value</b>
dn:	uid=majis,ou=cs,ou=tut,dc=fi
cn:	Juha Majalainen
mail:	majis@cs.tut.fi
telePhoneNumber:	+358 40 8376125
uid:	majis

Table 4.1: LDAP Entry

## 4.4 Directory Service

A *directory* is used to record information about a particular group of entries. *Directory service* stores and retrieves information from the directory on behalf of a set of authorized users. A directory is a specialized database optimized for reading, browsing and searching. Directories are tuned to give quick response to high-volume lookup or search operations. They may have the ability to replicate information widely in order to increase availability and reliability, while reducing response time. Directories tend to contain descriptive, attribute-based information and support sophisticated filtering capabilities.

## 4.5 Directory Information Tree

Each entry must have a unique name called a distinguished name (DN). The full DN format is described in RFC2253 [15]. The DN describes the path to an entry in an LDAP directory. The DN is constructed by taking the name of the directory entry itself called the Relative Distinguished Name or RDN and concatenating the names of its ancestor entries. RDN is the left-most component of a distinguished name. It must be an unique element in its location in a directory tree. For example a DN `uid=majis,ou=cs,ou=tut,dc=fi` have a RDN `uid=majis` (see Figure 4.1).

Directory entries are organized as a simple tree-like hierarchy, which is called Directory Information Tree (DIT, see Figure 4.1). The topmost level is the root or the source directory, which is generally the domain name component of a company, organization or a country (`dc=fi`). This level branches out to organizational units like departments (`ou=tut`), branches (`ou=cs`), divisions, etc. Below that are entries for individuals with common name, such as users (`uid=majis`), specific network resources.

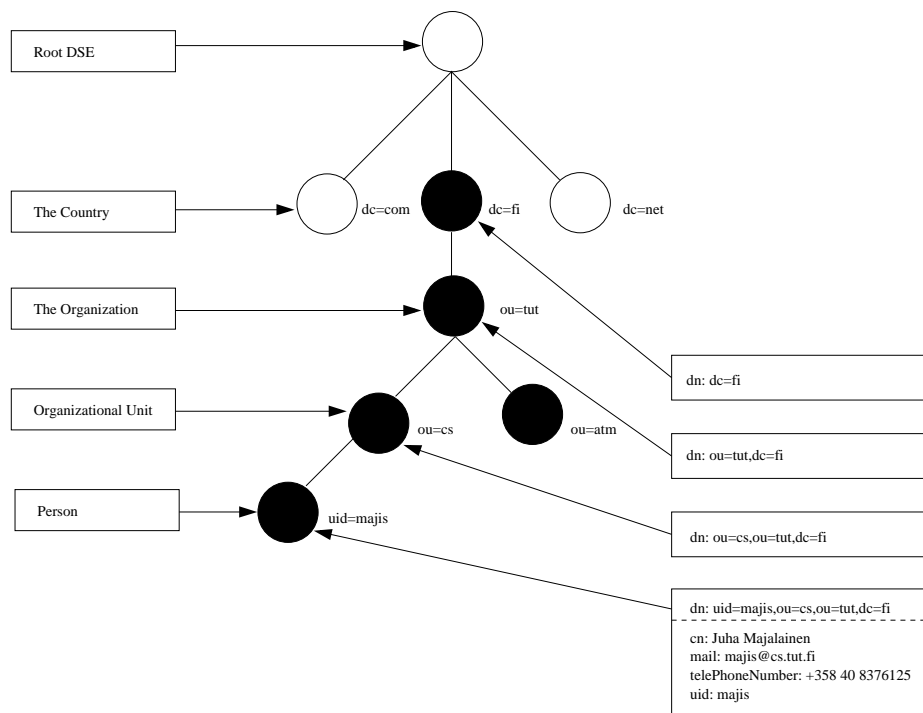


Figure 4.1: LDAP Directory Information Tree

## 4.6 LDAP Client-Server Model

An LDAP directory service is based on a client-server model. One or more LDAP servers contain the data elements that form the directory tree. Client-Server configuration without any referrals is described in Figure 4.2. A *referral* is an entity that is used to redirect a client's request to another server. A referral contains the names and locations of other entries. It is sent by the server to indicate that the information that the client has requested can be found at another location (or locations), possibly at another server or several servers.

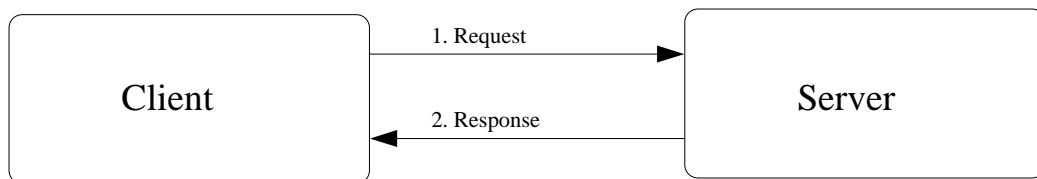


Figure 4.2: Client-Server configuration

In the example in Figure 4.2 the LDAP client connects to the LDAP server to obtain a set of information or to request the server to perform an operation. After that the server performs the operation or provides the requested information. If the server is unable to fulfill the client request, it refers the client to another LDAP server that might be able to perform the requested tasks (*referral*). A global directory service enables the LDAP client to connect to any available LDAP server when accessing a specific LDAP directory tree. This is an important feature of a global directory service, like LDAP. Client-Server configuration with referrals is described in Figure 4.3.

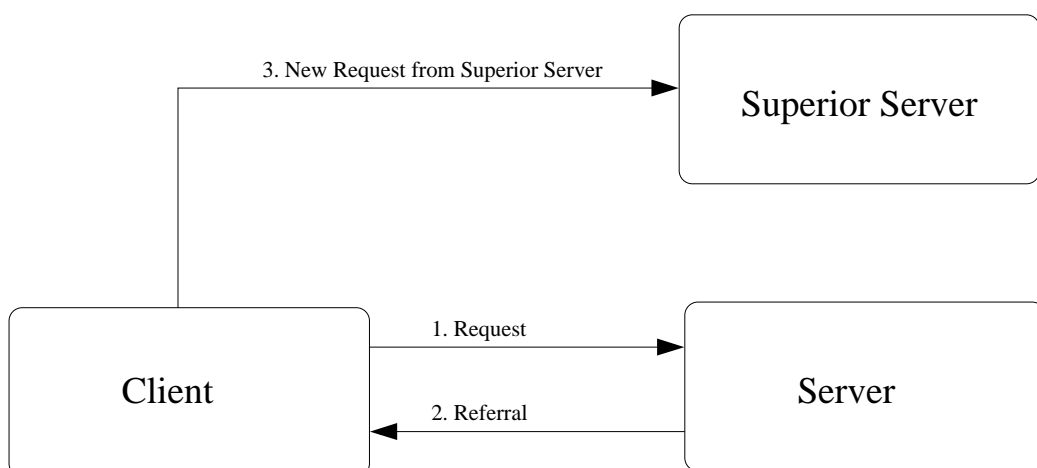


Figure 4.3: Client-Server configuration with referrals

## 4.7 LDAP Operations

LDAP functional model Determines how clients access and update information in an LDAP directory, as well as how data can be manipulated. LDAP offers nine basic functional operations: Add, delete, modify, bind, unbind, search, compare, modify DN (distinguished name), and abandon. Basic LDAP commands are described in Figure 4.2. Add, delete, and modify operations govern changes to directory entries. Bind and unbind operations enable and terminate the exchange of authentication information between LDAP clients and server, granting or denying end-users access to specific directories. Search operation locates specific users or services in the directory tree. Compare operation allows client applications to test the accuracy of specific values or information using entries in the LDAP directory. Modify DN operation makes it possible to change the name of an entry and abandon operation allows a client application to tell the directory server to drop an operation in progress.

LDAP command	Description
add	Add a new directory entry
delete	Delete a particular directory entry
modify	Modify a particular directory entry
bind	Start a session with an LDAP server
unbind	End a session with an LDAP server
search	Search the directory for matching directory entry's
compare	Compare one directory entry to a set of data
modify DN	Rename or modify the distinguished name of a directory entry
abandon	Abandon an operation previously sent to an LDAP server

Table 4.2: Basic LDAP commands

A typical LDAP exchange between a client and server might consist of the following steps:

1. The client opens a TCP connection to an LDAP server and submits a bind request. The bind operation includes the name of the directory entry as well as the credentials that will be used to authenticate the client. Credentials can be a simple password or a digital certificate.
2. After verifying the bind credentials submitted by the client, the server notifies the client that the bind operation has been successfully completed.
3. The client submits a search request to the server.
4. The server performs the search request and returns the matching entries to the client.
5. The client submits an unbind request to the server and closes the connection.
6. The server fulfills the unbind request.

## 4.8 Schema Specification

The LDAP schema is the definition of what can be stored in the directory. The basic thing in an entry is an *attribute* (see Section 4.8.1). Each attribute is associated with a syntax that determines what can be stored in that attribute (plain text, binary data, encoded data of some sort), and how searches against them work (case sensitivity, for example).

An *objectclass* (see Section 4.8.2) says what other attributes can or should be present. Schema may be extended to support additional syntaxes, matching rules, attribute types, and object classes (see Section 4.8.3).

### 4.8.1 Attribute Type Specification

An attribute type definition specifies the attributes syntax and how attributes of that type are compared and sorted. The attribute types in the directory form a class hierarchy.

<b>Identifier</b>	<b>Description</b>
NUMERICOID (mandatory)	AttributeType identifier
NAME	name used in AttributeType
DESC	description
OBSOLETE	true if obsolete; false or absent otherwise
SUP	derived from this other
EQUALITY	Matching Rule name
ORDERING	Matching Rule name
SUBSTRING	Matching Rule name
SYNTAX	Numeric OID of the syntax of values of this type
SINGLE-VALUE	default multi-valued
COLLECTIVE	default not collective
NO-USER-MODIFICATION	default user modifiable
USAGE	Description of attribute usage

Table 4.3: AttributeTypeDescription

Table 4.3 correspond to the definition of *AttributeTypeDescription* in RFC2252 [16].

```
attributetype ( <oid>
  NAME      'SlaId'
  EQUALITY  octetStringMatch
  SYNTAX    1.3.6.1.4.1.1466.115.121.1.26
  SINGLE-VALUE )
```

For example attributetype *SlaId* is a single valued octet string.

## 4.8.2 Object Class Specification

All LDAP entries in the directory are typed. That is, each entry belongs to object classes that identify the type of data represented by the entry. The object class specifies the mandatory and optional attributes that can be associated with an entry of that class. *Objectclass* is a special attribute that tells the LDAP server which attributes are required and which attributes may be allowed in a particular LDAP entry. The objectclass attribute supports the object-oriented concept of inheritance in that you can create a new objectclass that inherits all of the required and allowed attributes of its parent. An objectclass is similar to a table in a relational database.

Identifier	Description
NAME	Objectclass identifier
DESC	Description
OBSOLETE	true if obsolete, false or absent otherwise
SUP	Superior Objectclass
ABSTRACT	see RFC 2252.
STRUCTURAL	see RFC 2252.
AUXILIARY	see RFC 2252.
MUST	Attributes that must be present
MAY	Attributes that may be present

Table 4.4: ObjectClassDescription

Table 4.4 correspond to the definition of *ObjectClassDescription* in RFC2252 [16]. The format for representation of object classes is defined in X.501 [17]. In general every entry will contain an abstract class (*top* or *alias*), at least one structural object class, and zero or more auxiliary object classes.

```
objectclass ( <oid>
NAME      'ServiceLevelAgreement'
SUP       top
STRUCTURAL
MUST     SlaId)
```

For example objectclass *ServiceLevelAgreement* is structural objectclass and it has one mandatory attribute *SlaId*.

### 4.8.3 Subschema

LDAP v3 specifies that each directory entry may contain an operational attribute that identifies its subschema subentry. A subschema subentry contains the schema definitions for the object classes and attribute type definitions used by entries in a particular part of the directory tree. If a particular entry does not have a subschema subentry, then the subschema subentry of the root DSE is used. Root DSE is an entry that is located at the root of the DIT.

## 4.9 LDAP Security

To access the LDAP service, the LDAP client first must *authenticate* (see Section 4.9.1) itself to the service. It must tell the LDAP server who is going to be accessing the data so that the server can decide what the client is allowed to see and do. If the client authenticates successfully to the LDAP server, then when the server subsequently receives a request from the client, it will check whether the client is allowed to perform the request. This process is called *authorization* (see Section 4.9.2).

### 4.9.1 Authentication

The authentication mechanisms supported in LDAP version 3 are anonymous, simple, and the Simple Authentication and Security Layer (SASL) authentication. The SASL mechanism is described by RFC 2222 [18]. SASL allows secure authentication and encryption of the LDAP data between the client and the server. SASL acts as a security framework and can be used with various security packages, such as Kerberos [19], Generic Security Services API (GSSAPI) [20], or Transport Layer Security (TLS) [21].

### 4.9.2 Authorization

Authorization to LDAP directory entries is based on access control list (ACL). The ACL is composed of a series of access control instructions (ACIs) where the ACI is an attribute that can be assigned to any directory entry. Access control is supported in different ways by different LDAP server implementations. It is not discussed in more detail in this section.

## 4.10 Directory Enabled Network

LDAP standard defines basic elements for a number of fundamental applications such as white pages, e-mail, and user and group management. But as directories become increasingly successful and the number of applications relying on them increases, there is a corresponding need for new schema. The Directory Enabled Networks (DEN) [22] initiative, supported by numerous network software vendors, is a promising start.

DEN is an effort to define models and schema for network level devices and services, such as routers, switches, and VPNs (virtual private networks). Through DEN, these devices and services use LDAP to implement authentication and policy services. The DEN is one where users and applications interact in a controlled way with network elements and network services to provide predictable and repeatable services to users, while also strengthening security and simplifying provisioning and management of network resources.

# 5 Policy-Based Networking

## 5.1 Introduction

Mission-critical network applications such as Voice over IP (VoIP) and video applications coupled with the explosive growth of the Internet presents a real challenge to network managers and service providers who must provide reliable bandwidth and guaranteed service levels to network consumers in an environment of finite resources. Mission-critical applications are competing for bandwidth with low priority traffic such as FTP. Individual business priorities must be defined to ensure minimum software requirements for high priority applications.

Defining a relationship between the needs of the network consumers and available resources is the goal of Policy-Based Networking (PBN). The PBN framework is designed by the IETF's Resource Allocation Protocol working group (RAP-WG). The basic architecture is defined in RFC 2753 [1].

Policy-based networking is the management of a network so that various kinds of traffic such as data, voice, and video get the priority of availability and bandwidth needed to serve the network's consumers effectively. Using *policy statements* (see Section 5.3), network administrators can specify which kinds of service to give priority at what times of day on which parts of their network. This kind of management is often known as Quality of Service (QoS) management.

## 5.2 Policy Terminology

Terms commonly used with Policy-Based Management are defined in RFC 3198 [23] and RFC 2753 [1].

According to RFC3198 [23] a policy can be defined from two perspectives: A policy can be a definitive goal, course or method of action to guide and determine present and future decisions. Policies are implemented or executed within a particular context. In the network context policies can be defined as a set of rules to administer, manage, and control access to network resources.

In this section policies are defined from two levels: *Low-Level Policies* and *High-Level Policies*. High-level policies are for the network administrators and low-level policies are for the network devices. Sections 5.2.2 and 5.2.1 take a closer look at these two levels of policy.

Section 5.3 describes a policy-based network elements and architecture. According to IETF policy elements can be separated into Policy Enforcement Point (PEP), Policy Decision Point (PDP), Policy Repository and Policy Management Tool (PMT, see Section 7). The PDP is a logical entity that makes policy decisions for itself or for other network elements that request such decisions (see Section 5.3.3) and the PEP is a logical entity that enforces policy decisions (see Section 5.3.4) made by PDP. *Policy repository* is a specific data store that holds *policy rules*, their *conditions* and *actions*, and related policy data (see Section 5.3.2).

*Policy rule* is a basic building block of a policy-based system. It is the binding of a set of actions to a set of conditions - where the conditions are evaluated to determine whether the actions are performed [23]. *Policy conditions* is a representation of the necessary state and/or prerequisites that define whether a policy rules' actions should be performed. *Policy action* is a definition of what is to be done to enforce a policy rule, when the conditions of the rule are met.

### 5.2.1 High-Level policy

High-Level policies are specified in a format that is intuitive to a network administrator. That means that policy definition must be done in terms that are familiar to the network administrator.

The high-level policies required for network management can be specified in many different ways. For example administrators can use XML (see Section 3) to specify high-level policies. XML provides an application independent way of sharing data. Since XML is a tagged syntax that allows the marking of different types of information, mapping from the XML elements or attributes to high-level policies is relatively straightforward.

A high-level policy is usually quite static compared with the state of the network. The configuration of a device within the network can change more dynamically as compared to the high-level policy definition [24].

The IETF Policy work group [25] is defining a policy information model [26] to represent, manage, share, and reuse policies and policy information in a vendor-independent, interoperable, and scalable manner (see Section 5.4).

### 5.2.2 Low-Level policy

Low-Level policies must be specified precisely and consistently for each device in the network. Mapping from high-level policies to low-level policies is not so straightforward. A low-level policy must be specified to a level of detail that can be understood by a network element. From the point of view of network element, the policy must be specified with all the details that is needed for the enforcement of a policy. The low-level policies must be specified to match the device's capabilities which means that each device needs the individual configuration of a specific device [24].

## 5.3 Architectural Elements

A network *policy statement* is a formal set of statements that define how the network's resources are to be allocated among its consumers. Consumers can be individual users, departments, host computers, or applications. Resources can be allocated based on many factors such as time of day and availability of resources. Allocation can be static or dynamic. In order for policies to be active in the network, a network administrator needs to define them and the policy enforcement points need to enforce them.

Figure 5.1 depicts PBN architectural elements.

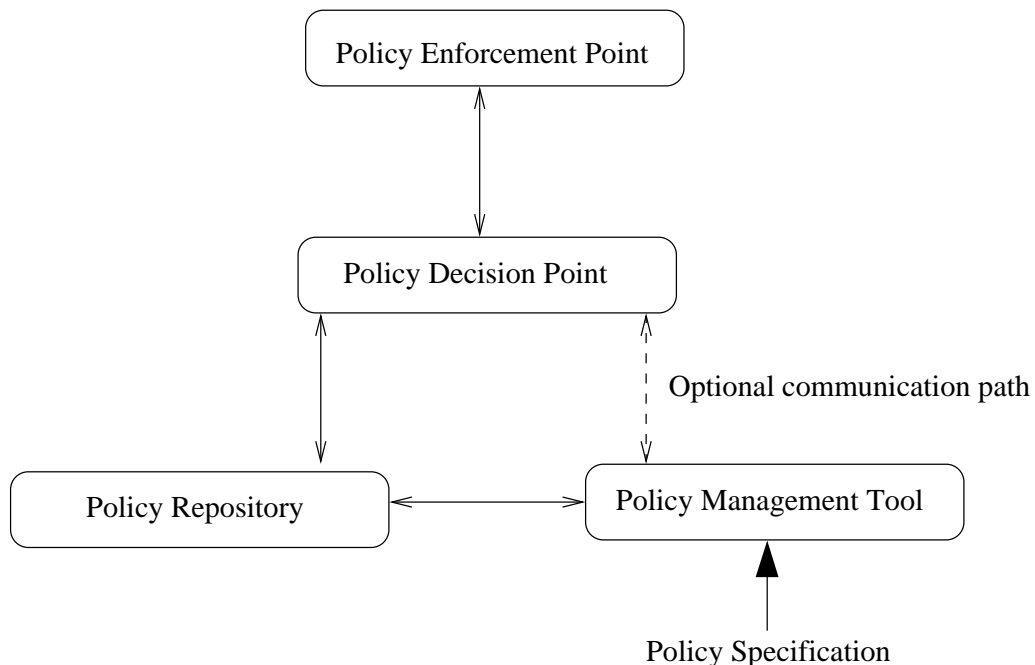


Figure 5.1: Architectural Elements [25]

High-Level policies and policy statements are created by network managers using a *Policy Management Tool* (PMT, see Section 7) and stored in a *Policy Repository* (see Section 5.3.2). During network operation, the policies are retrieved from the policy repository by a *Policy Decision Point* (PDP, see Section 5.3.3) and enforced to a *Policy Enforcement Point* (PEP, see Section 5.3.4). More detailed information about Policy-Based Management can be found in article [27].

### 5.3.1 Policy Management Tool

An administrator uses the policy management tool to define the different policies that are to be enforced in the network. The PMT needs to support the notion of policies specified as *high-level* abstractions as opposed to the policies specified as *low-level* abstractions. The high-level policies are used to express a network administrators objectives, which are then translated into a low-level policies that are interpreted by each of the devices.

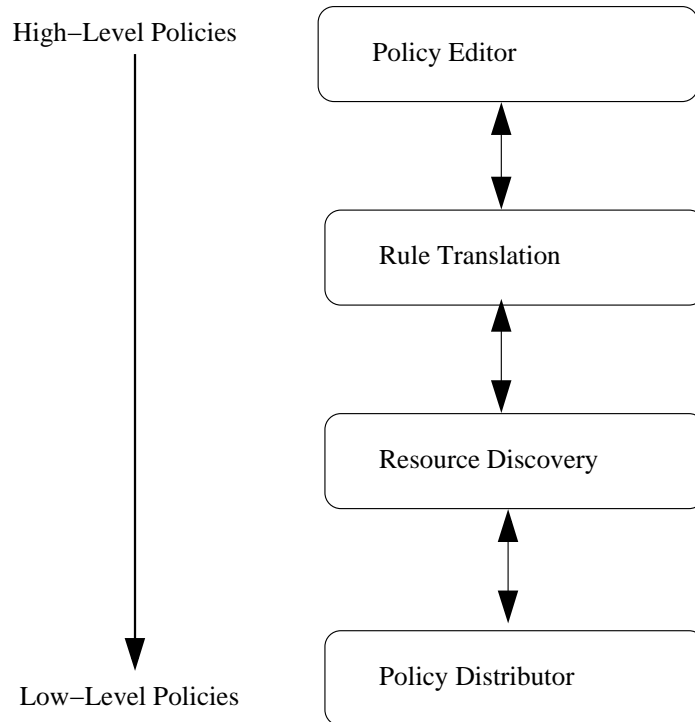


Figure 5.2: Policy Management Tool

Figure 5.2 depicts the policy management tool components.

*Policy Editor* is the means by which an administrator can edit and input the high-level policies within the network. It presents a user interface to network administrator. Policy editor is optimized for ease of use using appropriate techniques to depict the structure of the high-level policy information being entered at the level that is more human-friendly than that represented in the low-level policy.

The *rule translation* component is responsible for ensuring that the high-level policies are mutually consistent, correct and feasible with the existing capacity and topology of the policy-enabled network. Policies are considered inconsistent if they specify different con-

flicting actions to be taken for the same traffic flow. The rule translation validates the syntactic and semantic correctness of administrators input and checks that proper and acceptable values are specified for the different parameters that constitute the high-level policies.

The capabilities and topology of the network must be known to determine which policies are applicable to which set of policy targets. Topology of the network, the users and applications operational in the network is determined by the *resource discovery* component.

The *policy distributor* is responsible for that low-level policies are distributed to the *policy repository* and detect when the stored policies are modified.

### 5.3.2 Policy Repository

The policy repository is used to *store* the policies generated by the policy management tool. Policies are *retrieved* by the policy decision point or policy enforcement point (see Section 5.3). The central repository can be used to distribute and share the same policies among multiple policy systems.

Among the various alternatives, the LDAP-accessible (see Section 4) directory is the major contender. LDAP is the natural repository for policies. The directory service has a common name space which means that policies defined for a component can be reused or shared by other components.

LDAP directories are also hierarchical so policies can have precedence defined by their position in the DIT (see Figure 4.1). The directory services are distributed databases with master and slave concepts. The replication can be total or partial. The benefit is that the policies can be located closer to where they will be enforced. The LDAP directory is also a central repository. The benefit is that the policy is managed centrally, from a single point.

A schema (see Section 5.4) for storing policies in LDAP is under standardization at the IETF's Policy WG [25]. The LDAP mapping of the policies is specified in [28] and the information model is defined in [26]. A policy is stored as multiple linked entries in the directory. Each entry has generic attributes and specialized attributes for QoS, Security etc.

### 5.3.3 Policy Decision Point

The PDP is responsible for interpreting the policies stored in the policy repository and enforcing them to PEP. The PDP acts as a central command center for policies in the network.

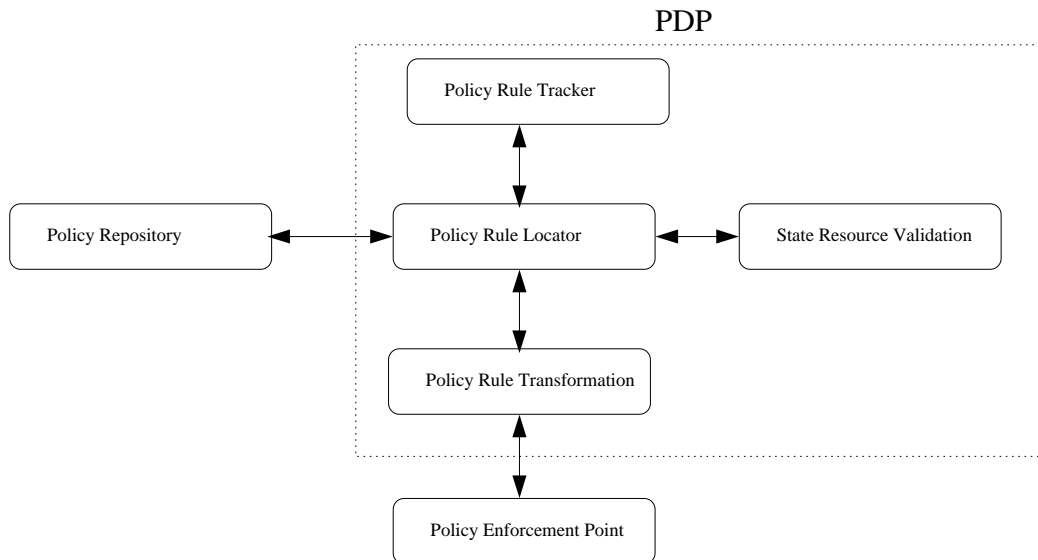


Figure 5.3: Policy Decision Point

Figure 5.3 depicts the policy decision tool components. PDP consist four components: *policy rule locator*, *policy rule transformation*, *state resource validation* and *policy rule tracker*. Policy Rule locator component is the means by locating the set of policy rules that is applicable to any PEP it is managing and retrieves the policy rules from the repository.

If PDP successfully retrieves the policies from the repository, it checks the current state of the network and updates the local state repository using the state resource validation component and also validates that the conditions required for the application of any policy is satisfied.

The policy rule transformation component translates the set of rules it retrieves from the policy repository to the format and syntax that is understood by the PEP. If the state resource validation component allows more reservations the PDP enforces the translated set of rules to the PEP. Using policy rule tracker component, The PDP monitors the repository keeping track of the changes in the policies that might take place. Policy rule tracker also listens notifications from the policy management tool.

### 5.3.4 Policy Enforcement Point

A device that can apply and execute the different policies is known as PEP. The PEP is responsible for executing the policies as defined by the PDP. It is also responsible for monitoring any statistic or other information relevant to its operation and reporting it to the appropriate places. The PEPs could be different devices, for example a network device such as router or switch, a firewall, a server or a host system.

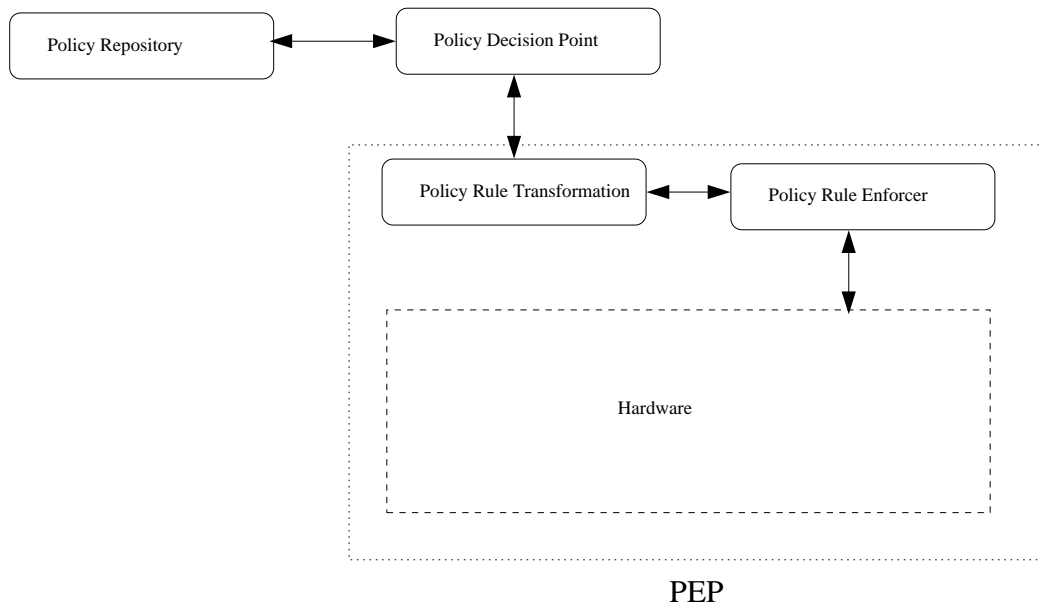


Figure 5.4: Policy Enforcement Point

Figure 5.4 depicts the policy enforcement point. The PEP consists of at least two components such as *policy rule transformation* component and *policy rule enforcer* component.

The policy rule transformation component translates the set of rules it retrieves from the PDP to the format and syntax that is understood by the policy rule enforcer. The policy rule enforcer executes the policies retrieved from the policy rule transformation component to the format and syntax that is understood by the underlying hardware.

The policy rule enforcer is also responsible for monitoring any statistics or other information relevant to its operation and for reporting it to the appropriate places.

## 5.4 IETF's Policy Schema

The IETF Policy work group [25] is defining a policy information model [26] to represent, manage, share, and reuse policies and policy information in a vendor-independent, inter-operable, and scalable manner. Figure 5.5 depicts that model. The IETF has chosen a rule-based policy representation in its specification. This model is an abstraction and representation of the entities in a managed environment and is independent of any specific repository, application, protocol, or platform. The policy classes and associations defined in this model are sufficiently generic to allow them to represent policies related to anything. However, it is expected that their initial application in the IETF will be for representing policies related to QoS and to IPsec.

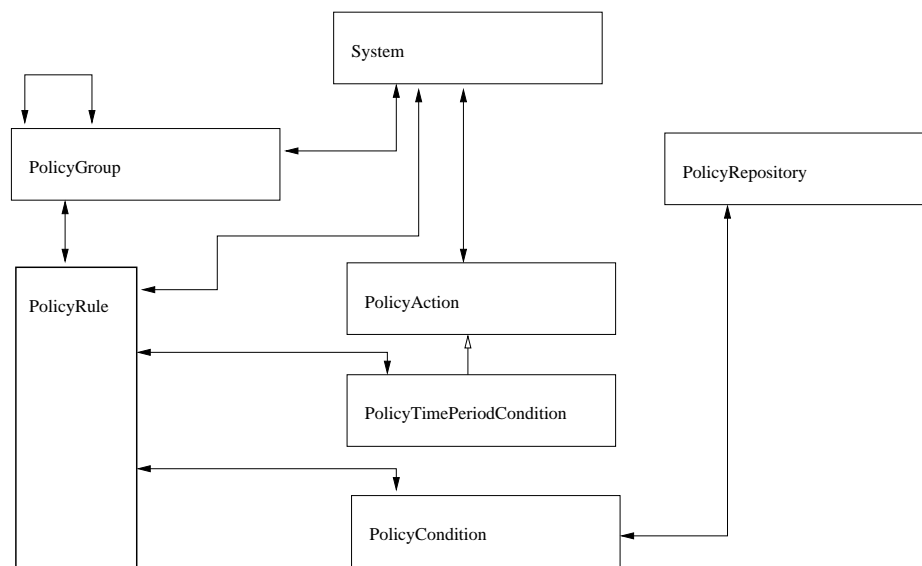


Figure 5.5: Overview of the Core Policy Classes and Relationships [26]

The IETF is also defining a mapping of that model to a directory schema [28] so that a LDAP directory can be used as a repository. The policy model defines a policy rule and its component policy conditions and policy actions. This model defines two hierarchies of object classes: structural classes representing information for representing and controlling policy data as specified in [26], and relationship classes that indicate how instances of the structural classes are related to each other. Classes are also added to the LDAP schema to improve the performance of a client's interactions with an LDAP server when the client is retrieving large amounts of policy-related information.

The IETF has also defined extensions to that model called QoS Policy Information Model (QPIM) [29], which contains a set of abstractions specific to differentiated services and integrated services management. That model presents an object-oriented information model for representing policies that administer, manage, and control access to network QoS resources.

## 6 Policy Provisioning using COPS

### 6.1 Introduction

The COPS (Common Open Policy Service) protocol was developed in the Resource Allocation Protocol (RAP) Working Group [30] of the IETF. The current COPS protocol is specified in RFC2748 [31]. The COPS is a protocol for exchanging network policy information between a policy enforcement point (PEP, see Section 5.3.4) and a policy decision point (PDP, see Section 5.3.3).

The COPS protocol is a client/server model where the PEP (client) sends requests (see Section 6.6.1) to the remote PDP and the PDP (server) returns decisions (see Section 6.6.2) back to the PEP.

Section 6.2 takes a closer look in COPS (Common Open Policy Service) [31]. The COPS runs directly over TCP which guarantees reliable exchange of messages between policy clients and a server (see Section 6.3). Therefore, no additional mechanisms are necessary for reliable communication between a server and its clients. Section 6.5 describes the use of the COPS protocol for support of policy provisioning (COPS-PR) [32]. The data carried by COPS-PR is a set of policy data known as a Policy Information Base (PIB, see Section 6.4) and COPS-PR message content used for efficient transport of policy data is described in Section 6.6.

## 6.2 Common Open Policy Service

The COPS is a protocol for exchanging network policy information between the PEP and the PDP. Figure 6.1 depicts the usage of various policy components in a typical COPS example. COPS protocol provides message level security for authentication, replay protection, and message integrity. The connection between the PEP and the remote PDP is reliable, because the COPS uses TCP as its transport protocol. The PDP listens on a well-known TCP port number (COPS=3288 [12]). The location of the remote PDP can either be configured, or obtained via a service location mechanism. If the remote PDP is not available, e.g. due to a network error, the PEP must try to connect to a remote backup PDP or revert to a local PDP (LPDP). At the beginning the PEP initiates the TCP connection and after that the communication between the PDP and the PEP is mainly stateful request / decision exchange.

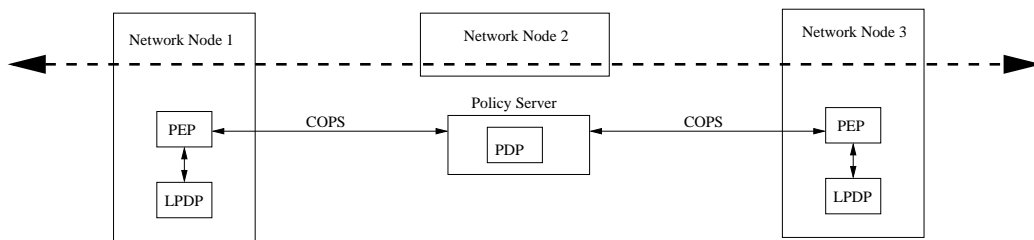


Figure 6.1: A COPS Usage [31]

The COPS protocol is stateful when request/decision state is shared between client and server. This means that request from the PEP are installed or remembered by the remote PDP until they are explicitly deleted by the PEP. At the same time, decisions from the remote PDP can be generated asynchronously at any time for a current installed requests. State from various events may also be inter-associated which means that the server may respond to new queries differently because of previously installed Request/Decision state(s) that are related. Additionally, the COPS is stateful in that it allows the server to push configuration information to the client, and then allows the server to remove such state from the client when it is no longer applicable [31].

The Figure 6.1 also depicts the framework of the policy-based admission control. The resources are allocated or released inside a PEP. There is at least one policy server in each policy domain. According to RFC3198 [23] administrative domain is a collection of ele-

ments and services, and/or a portion of an Internet over which a common and consistent set of policies are administered in a coordinated fashion.

### 6.3 Common COPS operations

The COPS protocol operates using a TCP connection between the PEP and the PDP. Figure 6.2 depicts standard COPS operations. The PEP connects to the PDP by using a client-open message (OPN) that describes PEPs capabilities and the type of the policy decisions it can enforce. Mandatory option for OPN message is `<PEP ID>`. The PEP ID is used to identify the PEP client to the remote PDP. The PDP responds with a client-accept (CAT) message that contains parameters for maintaining connection between PEP and PDP. Mandatory parameter for CA message is `<Keep alive Timer Value>`.

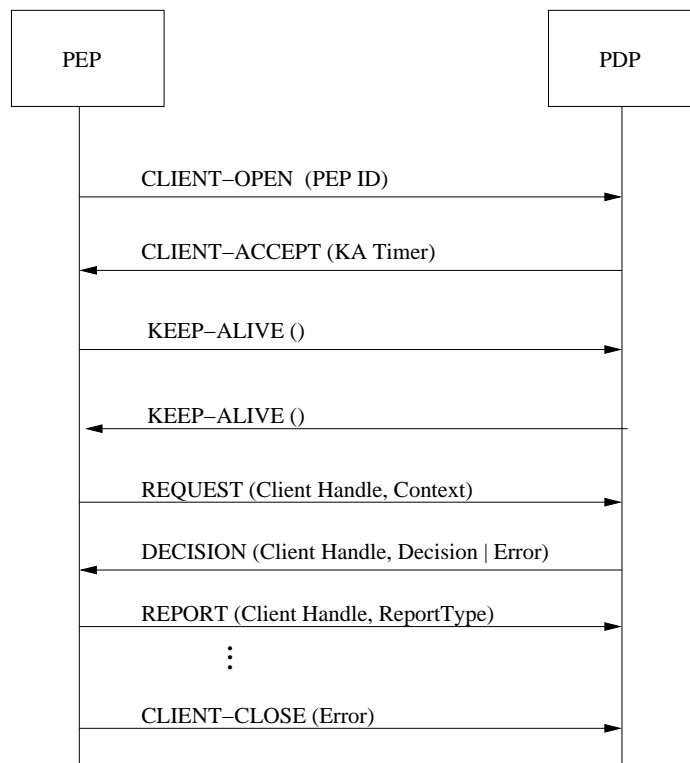


Figure 6.2: A standard COPS operations [31]

While the connection is open the PEP and the PDP exchange keep-alive (KA) messages, in order to ensure that the PEP and the PDP are in consistent state. When the PDP receives a KA message from a PEP, it echo a KA back to the PEP. When the PEP or PDP wants

to terminate the connection, they exchange a client-close (CC) message. The CC message includes the <Error> parameter to describe the reason for the close.

When a PEP needs a policy decision from the PDP it sends a COPS request (REQ) message to the PDP. This message contains <Context> parameter information that is a configuration request message. The PDP responds with a COPS decision (DEC) message advising what action the PEP should take. The PEP reports the results of enforcing these decisions to the PDP using a COPS report (RPT) message.

## 6.4 Policy Information Base

This section introduces the Policy Information Base (PIB) data transmitted by COPS-PR (see Section 6.5) and COPS. The PIBs are defined using the Structure of Policy Provisioning (SPPI) language. The SPPI language is defined in RFC3159 [33]. The SPPI uses the adapted subset of SNMP's [34] Structure of Management Information (SMI) [35] to write Policy Information Base (PIB) modules. The structure of SPPI is specified by Abstract Syntax Notation One (ASN.1) structures. ASN.1 is a standard [36] that defines a formalism for the specification of abstract data types. It is a formal notation used for describing data transmitted by telecommunications protocols.

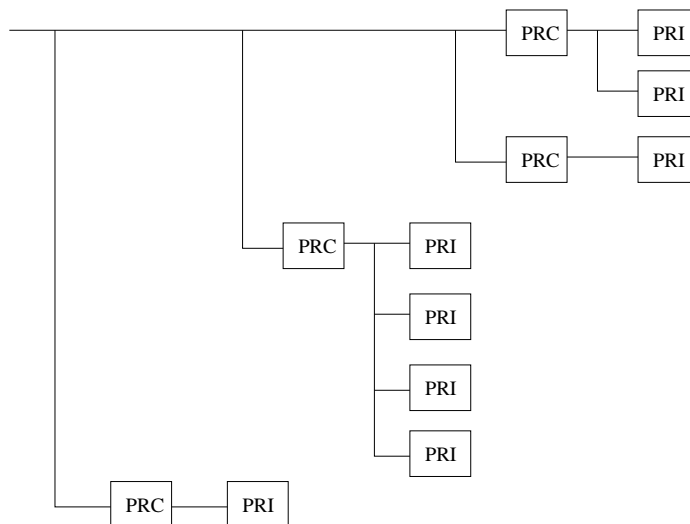


Figure 6.3: The PIB Tree [32]

The idea behind the use of PIBs for COPS is a general way to specify policy information.

The PIB can be described as a conceptual tree namespace where the branches of the tree represent structures of data or Classes (PRCs), while the leaves represent various instantiations of Provisioning Instances (PRIs). According to RFC3198 [23] PRC is an ordered set of attributes representing a type of policy data and PRI is an instance of the PRC. Figure 6.3 depicts the PIB tree. Two PIB modules, the framework PIB [37] and the DiffServ PIB [38], are needed in this content. More details on those modules can be found in the master's thesis of Jussi Lemponen [39] and details on the PIBs and SPPI can be found at the home page of the Resource Allocation Protocol (RAP) working group [30].

## 6.5 COPS Usage for Policy Provisioning

COPS Usage for Policy Provisioning architecture is defined in RFC3084 [32]. COPS-PR is optimized for provisioning policies across the PEP and the PDP, based on the requirements defined in Resource Allocation Protocol working group [30]. The COPS-PR protocol is an extension for the COPS protocol and it describe objects and the message formats used to carry the modeled policy data (PIB, see Section 6.4).

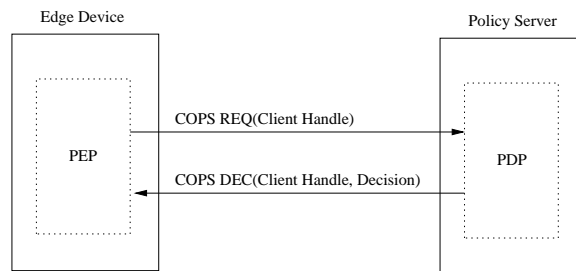


Figure 6.4: A standard COPS-PR operations [32]

COPS-PR operations are the same as in COPS because COPS-PR runs directly over COPS. COPS operations are described in Section 6.3 and COPS-PR message contents are described in Section 6.6. When the COPS connection is open (see Section 6.3), the PEP sends COPS REQ message to the PDP and tells information about itself. This information can include hardware type, software release, configuration information and so on. In response, the PDP send COPS DEC message to PEP which contains all provisioned policies that are currently relevant to that device. After that PEP maps them into its local QoS mechanisms, and installs them.

## 6.6 The COPS-PR Message Content

Because the COPS-PR messages form a hierarchical structure, it is easy to present them with the Backus-Naur Form (BNF) notation. Table 6.1 in Section 6.6.1 depicts COPS-PR request (REQ) message presented with the BNF notation and Table 6.2 in Section 6.6.2 is the BNF presentation for COPS-PR decision (DEC) message.

### 6.6.1 Request

The REQ message has the following format:

```
<Request> ::= <Common Header>
             <Client Handle>
             <Context = config request>
             *( <Named ClientSI> )
             [ <Integrity> ]
```

Table 6.1: Request (REQ) PEP -> PDP [32]

More details on the COPS-PR REQ message can be found on the COPS-PR specification (see RFC3084 [32]).

### 6.6.2 Decision

The DEC message has the following format:

```
<Decision Message> ::= <Common Header>
                       <Client Handle>
                       *( <Decision> ) | <Error>
                       [ <Integrity> ]

<Decision> ::= <Context>
               <Decision: Flags>
               [ <Named Decision Data: Provisioning > ]
```

Table 6.2: Decision (DEC) PDP -> PEP [32]

More details on the COPS-PR DEC message can be found on the COPS-PR specification (see RFC3084 [32]).

# 7 Policy Management Tool implementation

## 7.1 Introduction

The Policy-Based Networking (PBN) architecture, introduced in Section 5, describes policy elements which can be separated into a PEP (see Section 5.3.4), a PDP (see Section 5.3.3), and a PMT (see Section 7). All PBN architectural elements are implemented in Tampere University of Technology (TUT), Institute of Communications Engineering (ICE) laboratory. This chapter describes a part of that implementation, namely the PMT implementation.

The whole project overview is given in Section 7.2. Following that, the rest of this chapter covers the implementation of Policy Management Tool for bandwidth provisioning, which was the actual focus of this thesis.

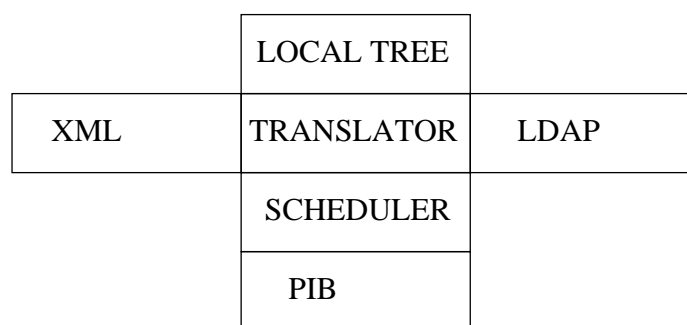


Figure 7.1: The PMT Implementation

The PMT implementation consists of six different components. Figure 7.1 depicts the relation of the components of the PMT. The XML component translates the high-level policy

data (see example in Appendix C) to C structures (see Appendix D). C structures are added to local tree (see Section 7.4) using translator component.

The translator component validates the information provided in the high-level policies and transforms them into the C structures and adds to the tree hierarchy. The component ensures that the policies specified are mutually consistent and that they cover all aspects of interest to the network administrator.

The High-Level policy data is created using XML and policy DTD specification (see Appendix B). The translator component validates the information provided in the high-level policies, transforms them into low-level policies and takes care that policy information in local tree and in LDAP policy repository does not conflict. The policy translation logic is also responsible for syntactical checks as well as semantical checks of policies.

All policy information is transferred to the LDAP policy repository using LDAP component. The PMT implementation uses OpenLDAP [40] as a policy repository to store policies. The OpenLDAP API defines a C language application program interface to the lightweight directory access protocol (LDAP). The OpenLDAP Software is an open source implementation of the Lightweight Directory Access Protocol. The LDAP server uses policy LDAP schema (see Appendix A) to represent policies in LDAP directory. The LDAP server relies on a schema for its database design.

Section 7.5 presents the heart of the PMT implementation, the scheduler. The scheduler component is responsible for scheduling events (see Appendix D.5). Every event has a time attribute and all events are sorted in `events` tree. Scheduler reads data from local tree and distributes that data to the PEP using PIB interface depending on the event time.

Finally, Section 7.6 describes a test network used to test all the implemented components (PDP, PEP and PMT). Configuration scripts used with test network (see Figure 7.11) are in Appendix F and G.

## 7.2 PBN Architecture Implementation Overview

The ultimate goal of implementation for PBN architecture is to provide users and applications with high quality data delivery services. The general PBN administration framework can be considered an adaptation of the IETF policy framework. This implementation partly follows the PBN architecture specifications introduced in Section 5. The aim of the project was to produce a prototype rather than a full-fledged product. Figure 7.2 depicts the PBN architecture implementation overview. This thesis covers XML, PMT, LDAP and partly PDP implementations. The PEP implementation is described in [39].

The implementation is carried out on Linux operating system but it should work with other UNIX operating systems also. Only the PEP part of the implementation is specific to Linux since it uses Linux DiffServ.

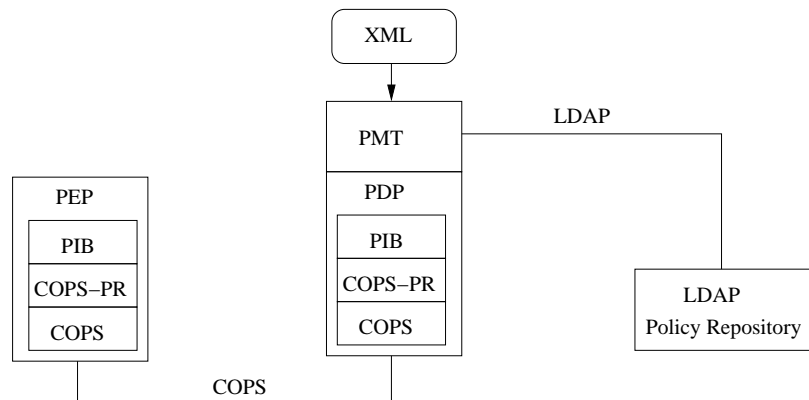


Figure 7.2: The PBN Architecture Implementation

Typical for a software project of this magnitude, the source code for the PBN architecture project including e.g. the PMT, the PEP and the PDP is held in a Concurrent Versions System (CVS) repository to facilitate version control. Also such common tools as `libtool`, `autoconf` and `automake` are employed to make the compilation of the source code as portable as possible and facilitate dependency checks. The whole project amounts to some 10000 lines of C code, over 2000 lines of header files, about 1400 lines of `perl` and 200 lines of `lex`.

## 7.3 XML to LDAP Mapping

The high-level policies required for network management can be specified in many different ways. The PMT implementation defines a new policy DTD to represent high-level policies with XML.

The policy DTD defines three new elements to represent high-level policies: *top* (which is the root element of the policy DTD), *BandwidthAllocationRequest* (BAR) (see Appendix B.1) and *ServiceLevelAgreement* (SLA) (see Appendix B.2).

```
<!ELEMENT top (
  BandwidthAllocationRequest |
  ServiceLevelAgreement
)*>
```

The element *top* is declared using choice list. Using choice list, only one of the child elements is permitted. The top element contains child elements *BandwidthAllocationRequest* and *ServiceLevelAgreement*. Cardinality operator *\** indicates that the child elements are optional and multi-valued (zero to many). BAR and SLA elements also have attributes which are defined in Appendix B.1 and B.2. XML attributes are used to describe XML elements.

*ServiceLevelAgreement* (SLA) elements provide a simple mechanism for allocating blocks of a particular service to specific customers. This facilitates long term bandwidth reservation for organizations or users, but doesn't constrain the allocation to specific flows. Once an SLA has been established, portions of this service can be assigned to specific flows. The SLA indicates allowed bandwidth, source address, destination address etc.

*BandwidthAllocationRequest* (BAR) elements are used by clients to request portions of a service allocated to them by an SLA, for individual flows.

Relationship between the *BandwidthAllocationRequest* and the *ServiceLevelAgreement* is done using attributes *Slaid* and *BarId*. The *Slaid* identifies the SLA and *BarId* identifies the BAR.

The policy DTD can be described as a conceptual tree name-space where the branches

of the tree represent ServiceLevelAgreements (SLAs), while the leaves represent various instantiations of BandwidthAllocationRequests (BARs). There may be multiple BARs for any given SLA. Figure 7.3 depicts a policy DTD hierarchy.

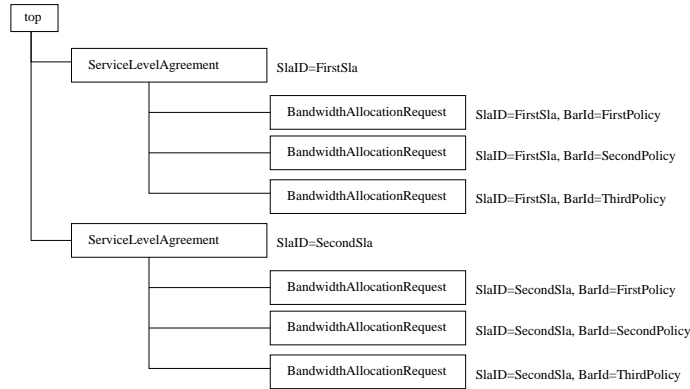


Figure 7.3: the Policy DTD hierarchy

The Policy DTD is defined so that mapping from high-level policy data (XML) to a LDAP directory is as easy as possible. For the structural elements in the policy DTD, the mapping is basically one-for-one: policy DTD elements map to LDAP classes, policy DTD attributes map to LDAP attributes. Figure 7.4 depicts mapping from XML to LDAP repository.

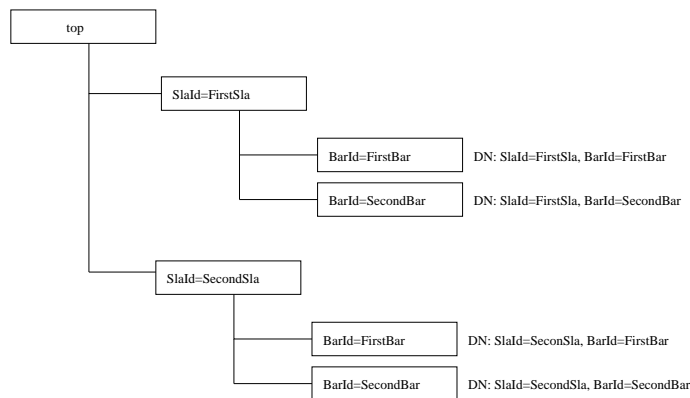


Figure 7.4: The LDAP Policy Repository

Instances in a directory are identified by distinguished names (DNs), which provide the same type of hierarchical organization that a file system provides in a computer system. Attributes SlaId and BarId are representing distinguished name (DN) references, and relationships in the Directory Information Tree (DIT).

## 7.4 Local Policy Tree

This section describes structures and all the important functions used to handle the local tree hierarchy. Figure 7.5 depicts the local tree and all the important structures and pointers. The local tree hierarchy consists of five different trees: *dn*tree, *slatree*, *timetree*, *bartree*, and *events*. Pointers in the Figure 7.5 are depicted using dotted line. The PMT implementation uses Red-Black trees as a container for C structs.

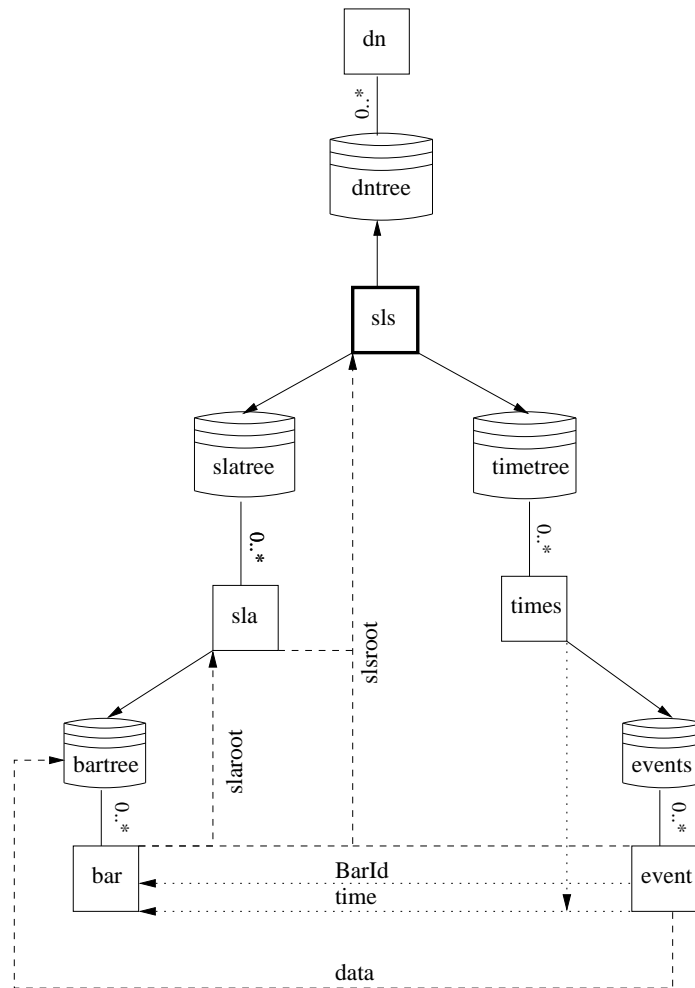


Figure 7.5: The PMT Local Tree and Structures

The checking for policy conflicts and dominance needs to be performed every time when adding policy data to local tree hierarchy. One of the key validation steps is to verify that the policies do not contradict each other. The central point in the tree hierarchy is a *struct sls* (see Appendix D.6) and every modification to other trees is done through it. The *struct sls* consists of *slatree*, *timetree* and *dn tree*. The *struct sls* can be allocated using a function

```
struct sls *alloc_sls();
```

The PMT implementation uses a one global `struct sls` variable as a root for the tree hierarchy:

```
struct sls *root = alloc_sls();
```

The XML policy data is first mapped, using translator component, to `struct dn` and that allocated struct is added to a `dntree`. A new `struct dn` can be allocated using function:

```
struct dn *alloc_dn(char *name);
```

and adding this allocated struct to the `dntree` can be done using the global root and function pointers:

```
struct dn *dn = alloc_dn('`dn_name`');  
root->addDN(root, dn);
```

The `dntree` is used to store policy data from XML and LDAP and all the synchronizations between XML and LDAP is done using this tree. When a synchronization part between LDAP and XML is done, the policy data is mapped from the `dntree` to other trees depending on the content.

ServiceLevelAgreement elements are mapped to `struct sla` (see Appendix D.3). After that, allocated SLAs are added to `slatree` hierarchy so that first incoming SLA initializes and allocates a `slatree`. A new `struct sla` can be allocated using `struct dn` and allocation function:

```
struct sla *alloc_sla(struct dn *dn);
```

Adding this allocated struct to the `slatree` can be done using the global root and function pointers:

```
struct sla *sla = alloc_sla(dn);  
root->addsla(root, sla);
```

Figure 7.6 depicts an example of a populated `slatree` in a `struct sls`.

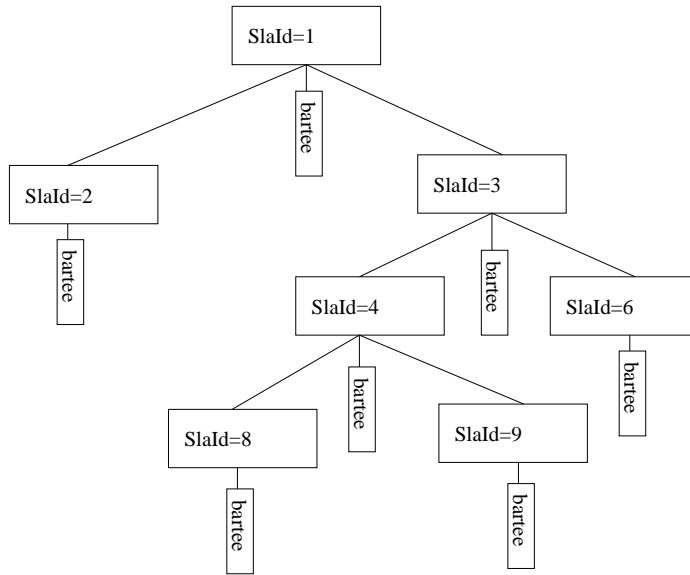


Figure 7.6: slatree

BandwidthAllocationRequests elements are mapped to struct `bar` (see Appendix D.1). The struct `bar` can be allocated using allocation function:

```
struct bar *alloc_bar(struct dn *dn);
```

As one can see in the Figure 7.6 a bartree is created into every SLA entry when SLA is added to a slatree. The bartree is initialized and allocated in the same way than a slatree.

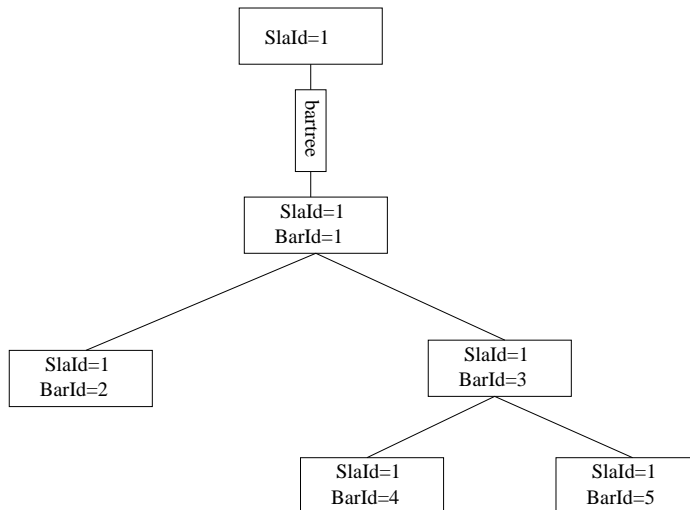


Figure 7.7: bartree

When adding a new BAR to bartree, a correct SLA must be searched from the slatree using `SlaId` as a search key. After the correct SLA is found from the slatree, the BAR can be

added to the bartree. Figure 7.7 depicts an example of a populated bartree in a SLA entry.

Adding this allocated struct to bartree can be done using the global root and function pointers:

```
struct bar *bar = alloc_bar(dn);  
root->addbar(root,bar);
```

The Timetree and the events trees are populated automatically at the same time when adding SLAs and BARs to a tree hierarchy. The timetree is used as a repository for a struct times (see Appendix D.4) and events tree is used as a repository for a struct event (see Appendix D.5).

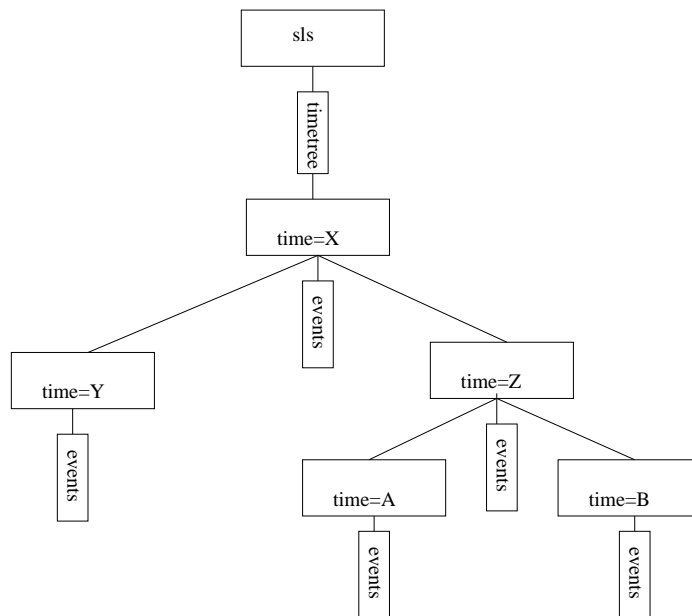


Figure 7.8: timetree

The struct times consists of a pointer to the events tree and to a struct tm \*time. Every struct bar consists of a struct tm BarStartTime and a struct tm BarEndTime. Both times are added to timetree which means that when start time and end time events are used, the BAR is also used.

As one can see in the Figure 7.8 a events tree is created into every entry when struct times is added to timetree. The events tree is initialized and allocated same way than other trees.

Figure 7.8 depicts an example of a populated `timetree` in a struct `sls`. Searching times from `timetree` can be done using `struct tm *time` as a search key.

Now all the BARs are added to the `bartree`, SLAs to the `slatree` and times from BARs to the `timetree` but as one can see in Figure 7.8, every struct `times` entry also has a pointer to a `events` tree.

The `events` tree is the last tree used to implement local tree hierarchy. The `events` tree is initialized same way than other trees and populated automatically. The `events` tree uses a `struct tm *time` as a search key. Figure 7.9 depicts an example of two events pointing to the struct `bar` which is somewhere in the `bartree`.

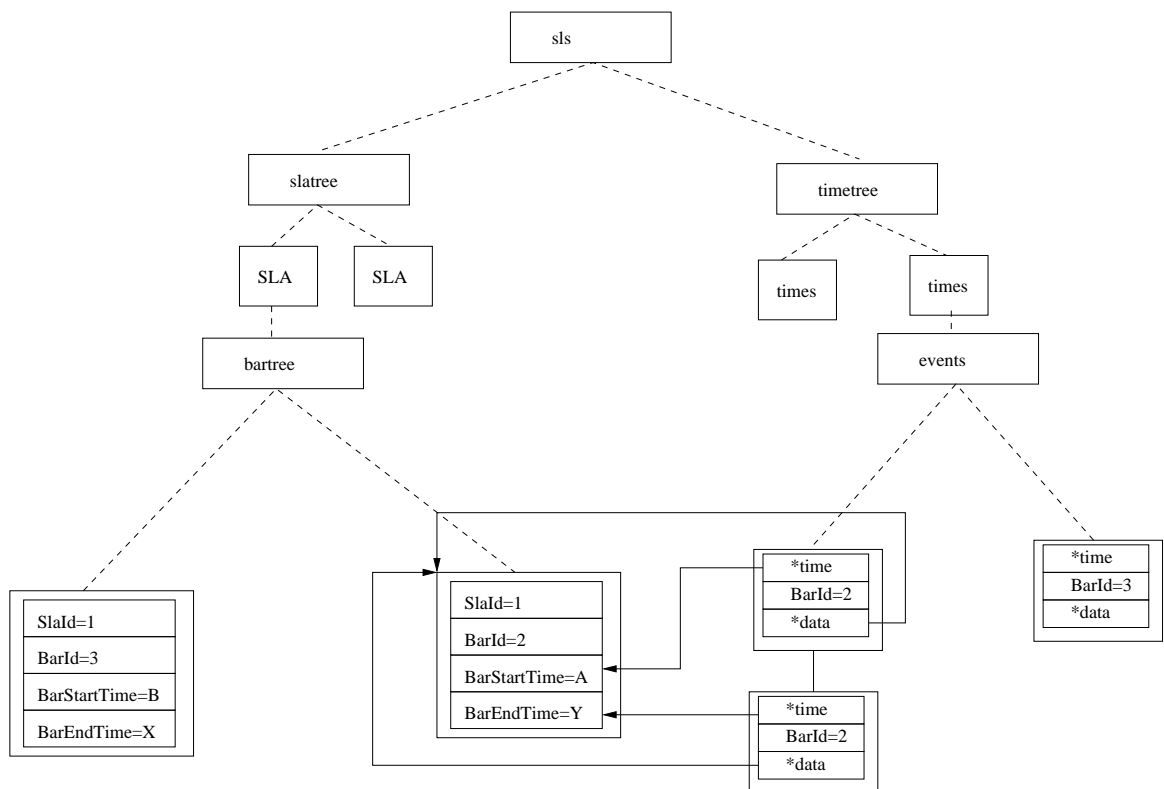


Figure 7.9: events tree

As one can see in the Figure 7.9, it is quite easy to find a correct BAR if we have a struct event. Actually we don't have to know which tree holds that entry because we can just use pointers to point to it from struct event.

## 7.5 Scheduler

This section presents the heart of the PMT implementation, the scheduler. The scheduler component is responsible for scheduling events and distributing those events to the PEP.

The function

```
int main_loop(struct sls *sls);
```

contains the PMT's main loop. The function is called right after the local tree is initialized and populated. The `main_loop` function is described in Appendix E.

The main loop basically consist of using the system call `select` to check whether new policies are needed to be sent to the PEP, to handle COPS messages and to get new policies from the LDAP policy repository. In addition to this, the function `cops_do_timers()` is called to handle sending COPS keep-alive messages periodically. Figure 7.10 depicts the main loop procedure.

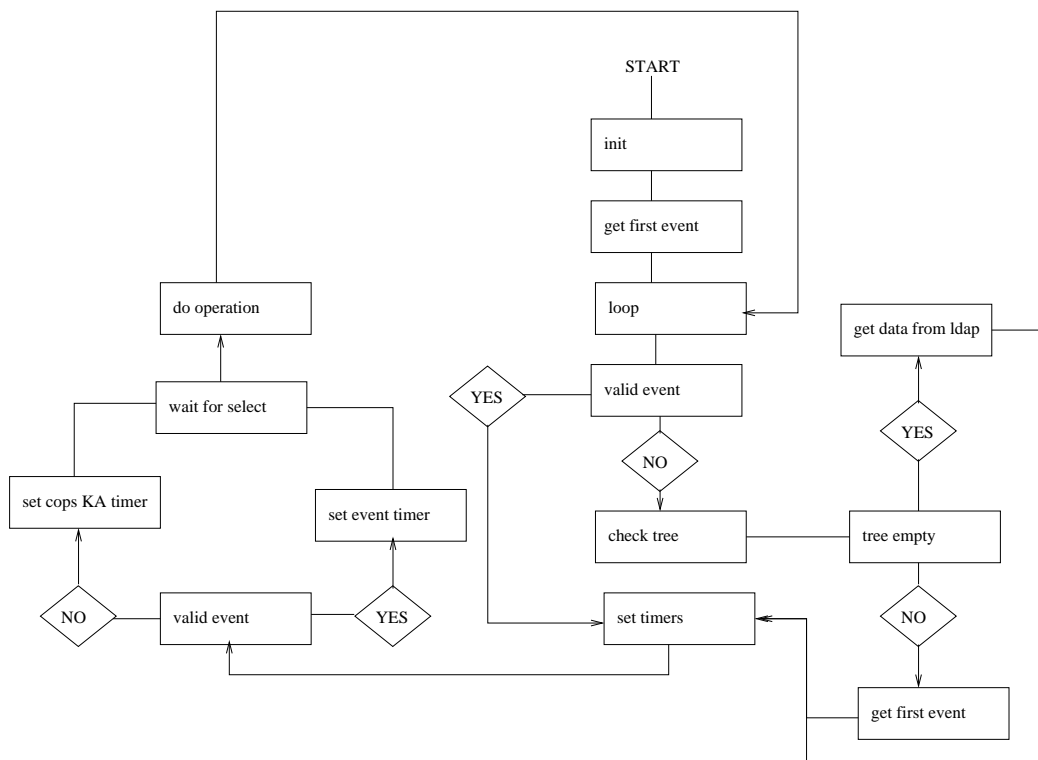


Figure 7.10: `main_loop()`

At the beginning in the main loop procedure, all the variables are initialized. After that,

`get_first_event` function pointer is used to get first event from the events tree.

```
event = sls->get_first_event(sls);
```

It is fast to get first event from the events tree because the events tree is sorted depending on the event time. That means, the first event is the first entry in the tree also. If the event is not valid or it is missing, the scheduler procedure checks the event tree again. If the events tree is still empty it tries to check if there is a new policy data in the LDAP policy server. The scheduler uses LDAP component to connect policy server and transfers all the policy data from LDAP to PMT's local tree hierarchy.

```
if(event == NULL && sls->countevent(sls) == 0){
    if(sls->getldapdata(sls) == 0)
        event = sls->get_first_event(sls);
}
```

As one can see, the procedure gets the first event from the events tree after getting the policy data from the LDAP server.

Next part of the scheduler is to set COPS timer with function

```
next_time = set_timer(event);
```

That function checks which COPS message is needed to be sent first: COPS keep-alive message or COPS-PR decision message. This `next_time` variable is used with `select` function. `select` function waits until timer expires. The functions `select` can also wait for a number of file descriptors to change status. Incoming COPS messages are also handled with `select` function. COPS packet handling is done using function `handle_packet(conn->conn)`.

The loop procedure also checks if it is time to install event or remove event (see Appendix E). After sending or receiving COPS messages, the main loop start again and check if the event is still valid. If the event is not valid anymore, scheduler check if there is more valid events in the events tree. If the tree is empty, scheduler checks the policy server like earlier. The loop actually loops forever.

## 7.6 Test Network

This section introduces a PBN test network used to test all the implemented components. The test network was designed to ensure that our implementation works as specified in the PBN concept. Figure 7.11 depicts the test network, The test network consists of two types of main entities: access nodes (DS boundary nodes) and core nodes (DS interior nodes). The PEP implementation acts as a DS boundary node (pictured on the Figure 7.11 with the symbol PEP) and Linux PC's with a correct DiffServ configurations as a DS interior node (pictured on the Figure 7.11 with the symbol IN). The DiffServ configurations used with interior nodes are specified in Appendix G.

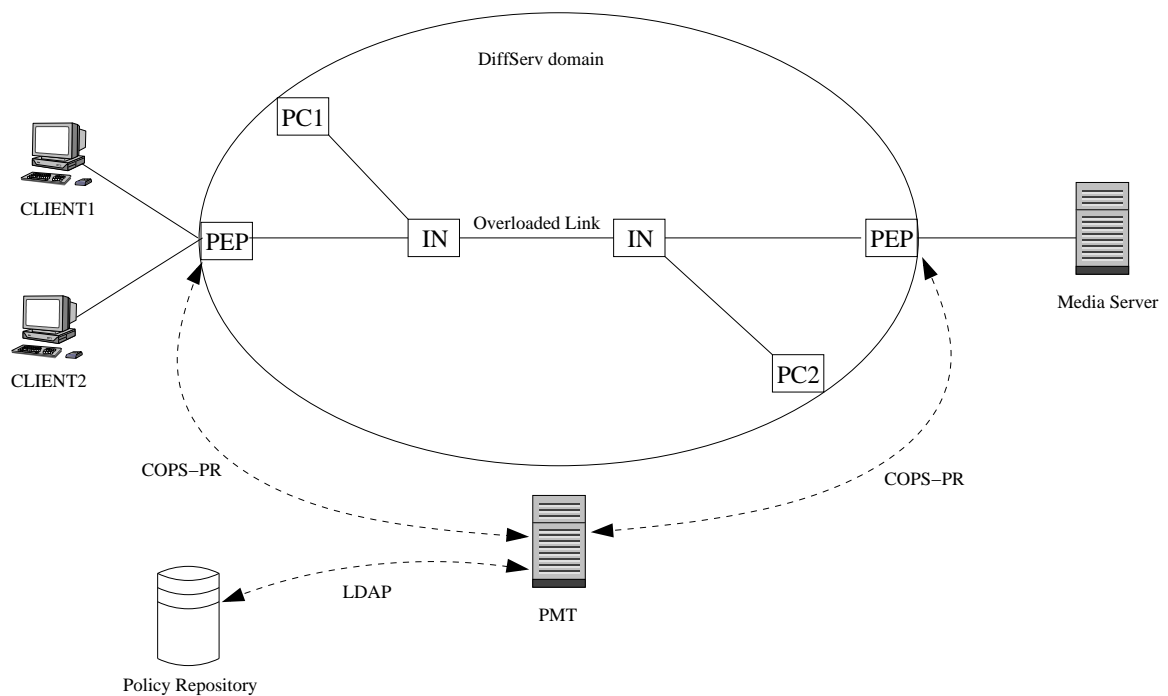


Figure 7.11: Test Network

The PMT component translates high-level policy data to low-level policies and transfers it to the policy repository using LDAP. When a PEP boots up, it contacts the PMT and opens a COPS connection which is maintained as long as both parties are up. The PMT configures the PEP using a policy data (see Appendix C).

## 8 Conclusions

The goal of this thesis was to implement a Policy Management Tool for bandwidth provisioning. In practice this was achieved with many different components, complex data structures and mapping high-level policy data to many different low-level policy formats, e.g., LDAP, PIB and local tree hierarchy.

The biggest problem during the implementation was caused by the lack of industrial policy standards. The IETF Policy working group is defining an abstract policy information model to represent policies. Mapping from this abstract informational model to low-level policy formats is still under development. The easiest way to map high-level policy data to low-level policy formats was to define an own policy language. In this case XML was used. Mapping from XML to LDAP is basically one-to-one: policy DTD elements map to LDAP classes, policy DTD attributes map to LDAP attributes. Actually mapping from LDAP to PIB tables is one-to-one mapping too.

The next problem was caused by the synchronization and checking for policy conflicts. Every time when adding a new policy data to the system, high-level policy data must be in sync with other low-level formats. XML data is mapped to LDAP directory and local tree hierarchy. In this case local tree hierarchy is representing a kind of state repository where synchronization and every policy translation were done.

The policy management tool implementation only supports intra-domain resource reservations at this moment. Inter-domain end-to-end signaling between policy-based networks is extremely complex to implement. Inter-domain policy-based networking is not only a tech-

nical problem but also a political issue. Different service providers have different service level specifications, infrastructure and needs. There is not any single protocol or mechanism to handle this kind of situation yet.

Even with the implementation problems policy-based networking is becoming an attractive mechanism for service providers to control network services. However, there is a concern that policy-based networking is too complicated. Networks have become more heterogeneous, complicated, distributed environments which are increasingly mission-critical. The next few years will probably show how well policy-based networking is accepted by the service providers and networking professionals.

# References

- [1] RFC 2753; R. Yavatkar, D. Pendrakis, and R. Guerin. *A Framework for Policy-based Admission Control*, January 2000.
- [2] *The IETF's Differentiated Services Working Group Charter*. <http://www.ietf.org/html.charters/diffserv-charter.html>, December 2000.
- [3] RFC 2474; K. Nichols, S. Blake, F. Baker, and D. Black. *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, December 1998.
- [4] RFC 2475; S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss. *An Architecture for Differentiated Services*, December 1998.
- [5] RFC 791; J. Postel. *Internet Protocol*, September 1981.
- [6] RFC 1349; P. Almquist. *Type of Service in the Internet Protocol Suite*, July 1992.
- [7] RFC 1812; F. Baker. *Requirements for IP Version 4 Routers*, June 1995.
- [8] RFC 3246; B. Davie, A. Charny, J.C.R. Bennett, K. Benson, J.Y. Le Boudec, W. Courtney, S. Davari, V. Firoiu, and D. Stiliadis. *An Expedited Forwarding PHB (Per-Hop Behavior)*, March 2002.
- [9] RFC 2597; J. Heinänen, F. Baker, W. Weiss, and J. Wroclawski. *Assured Forwarding PHB Group*, June 1999.
- [10] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, and Eve Maler. *Extensible Markup Language (XML) 1.0 (Second Edition)*. <http://www.w3.org/>, October 2000.
- [11] ISO 8879. *Standard Generalized Markup Language*, October 1986.
- [12] *Internet Assigned Numbers Authority*. <http://www.iana.org>.
- [13] RFC 1777; W. Yeong, T. Howes, and S. Kille. *Lightweight Directory Access Protocol*, March 1995.
- [14] RFC 2251; M. Wahl, T. Howes, and S. Kille. *Lightweight Directory Access Protocol (v3)*, December 1997.
- [15] RFC 2253; M. Wahl, T. Howes, and S. Kille. *LDAP (v3): UTF-8 String Representation of Distinguished Names*, December 1997.
- [16] RFC 2252; M. Wahl, A. Coulbeck, T. Howes, and S. Kille. *LDAP (v3): Attribute Syntax Definitions*, December 1997.
- [17] CCITT Recommendation X.501. *The Directory: Models*. ITU-T Recommendation X.501, 1993.

- [18] RFC 2216; J. Myers. *Simple Authentication and Security Layer (SASL)*, October 1997.
- [19] RFC 1510; J. Kohl and C. Neuman. *The Kerberos Network Authentication Service (V5)*, September 1993.
- [20] RFC 2078; J. Linn. *Generic Security Service Application Program Interface, Version 2*, January 1997.
- [21] RFC 2246; T. Dierks and C. Allen. *The TLS Protocol Version 1.0*, January 1999.
- [22] *Directory Enabled Network*. [http://www.dmtf.org/standards/standard\\_den.php/](http://www.dmtf.org/standards/standard_den.php/), April 2002.
- [23] RFC 3198; A. Westerinen, J. Schnizlein, J. Strassner, M. Scherling, B. Quinn, S. Herzog, A. Huynh, M. Carlson, J. Perry, and S. Waldbusser. *Terminology for Policy-Based Management*, November 2001.
- [24] Dinesh C. Verma. *Policy-Based Networking - Architecture and Algorithms*, November 2000.
- [25] *The IETF's Policy Framework Working Group Charter*. <http://www.ietf.org/html.charters/policy-charter.html>, April 2002.
- [26] RFC 3060; B. Moore, E. Ellesson, J. Strassner, and A. Westerinen. *Policy Core Information Model – Version 1 Specification*, February 2001.
- [27] IEEE NETWORK. *Simplifying Network Administration Using Policy-Based Management*, march/april 2002 Vol. 16 No. 2.
- [28] J. Strassner, E. Ellesson, B. Moore, and R. Moats. *Policy Core LDAP Schema*. draft-ietf-policy-core-schema-16.txt, October 2002. Internet-draft, work in progress.
- [29] B. Moore. *Policy Core Information Model Extensions*. <http://www.ietf.org/internet-drafts/draft-ietf-policy-pcim-ext-08.txt>, May 2002. Internet-draft, work in progress.
- [30] *The IETF's Resource Allocation Protocol Working Group Charter*. <http://www.ietf.org/html.charters/rap-charter.html>, December 2000.
- [31] RFC 2748; D. Durham, J. Boyle, R. Cohen, S. Herzog, R. Rajan, and A. Sastry. *The COPS (Common Open Policy Service) Protocol*, January 2000.
- [32] RFC 3084; K. Chan, J. Seligson, D. Durham, S. Gai, K. McCloghrie, S. Herzog, F. Reichmeyer, R. Yavatkar, and A. Smith. *COPS Usage for Policy Provisioning (COPS-PR)*, March 2001.
- [33] K. McCloghrie, M. Fine, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, and F. Reichmeyer. *Structure of Policy Provisioning Information (SPPI)*, August 2001.

- [34] RFC 1905; J. Case, M. Rose, and S. Waldbusser. *Protocol Operations for Version 2 of the Simple Network Management Protocol (SNMPv2)*, January 1996.
- [35] RFC 2578; K. McCloghrie, D. Perkins, and J. Schoenwaelder. *Structure of Management Information Version 2 (SMIV2)*, April 1999.
- [36] ASN.1; ITU-T Rec. X.680 (1997) | ISO/IEC 8824-1:1998. *Abstract Syntax Notation One (ASN.1) - Specification of Basic Notation*.
- [37] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, R. Sahita, A. Smith, and F. Reichmeyer. *Framework Policy Information Base*. draft-ietf-rap-frameworkpib-09.txt, June 2002. Internet-draft, work in progress.
- [38] M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, C. Bell, A. Smith, and F. Reichmeyer. *Differentiated Services Quality of Service Policy Information Base*. draft-ietf-diffserv-pib-09.txt, June 2002. Internet-draft, work in progress.
- [39] Jussi Lemponen. Implementation of differentiated services policy information base for linux. Master's thesis, Tampere University of Technology, <http://www.atm.tut.fi/faster/qbone/linux-pep.pdf>, August 2000.
- [40] <http://www.openldap.org/>, December 2002.

# Appendix A: LDAP Policy Schema

## A.1 Schema attributes

```
attributetype ( 5.5.5.1 NAME      'BarId'
                EQUALITY octetStringMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
                SINGLE-VALUE )
attributetype ( 5.5.5.2 NAME      'SrcAddr'
                EQUALITY octetStringMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
                SINGLE-VALUE )
attributetype ( 5.5.5.3 NAME      'SrcAddrMask'
                EQUALITY octetStringMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
                SINGLE-VALUE )
attributetype ( 5.5.5.4 NAME      'DstAddr'
                EQUALITY octetStringMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
                SINGLE-VALUE )
attributetype ( 5.5.5.5 NAME      'DstAddrMask'
                EQUALITY octetStringMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
                SINGLE-VALUE )
attributetype ( 5.5.5.6 NAME      'SrcPortMin'
                EQUALITY integerMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
                SINGLE-VALUE )
attributetype ( 5.5.5.7 NAME      'SrcPortMax'
                EQUALITY integerMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
                SINGLE-VALUE )
attributetype ( 5.5.5.8 NAME      'DstPortMin'
                EQUALITY integerMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
                SINGLE-VALUE )
attributetype ( 5.5.5.9 NAME      'DstPortMax'
                EQUALITY integerMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
                SINGLE-VALUE )
attributetype ( 5.5.5.10 NAME     'Dscp'
                EQUALITY integerMatch
                SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
                SINGLE-VALUE )
attributetype ( 5.5.5.11 NAME     'Protocol'
```

```

    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 5.5.5.12 NAME    'PepAddr'
    EQUALITY octetStringMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
    SINGLE-VALUE)
attributetype ( 5.5.5.13 NAME    'BitRate'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 5.5.5.14 NAME    'BarStartTime'
    EQUALITY generalizedTimeMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
    SINGLE-VALUE)
attributetype ( 5.5.5.15 NAME    'BarEndTime'
    EQUALITY generalizedTimeMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
    SINGLE-VALUE)
attributetype ( 5.5.5.16 NAME    'BurstSize'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 5.5.5.17 NAME    'Interval'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 5.5.5.18 NAME    'Priority'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 5.5.5.19 NAME    'FilterNegation'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 6.6.6.1 NAME    'SlaId'
    EQUALITY octetStringMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
    SINGLE-VALUE)
attributetype ( 6.6.6.2 NAME    'MaxNRes'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 6.6.6.3 NAME    'MaxBitRate'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 6.6.6.4 NAME    'MaxOneBitRate'
    EQUALITY integerMatch
    SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
    SINGLE-VALUE)
attributetype ( 6.6.6.5 NAME    'MaxBurstSize'

```

```

EQUALITY integerMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.27
SINGLE-VALUE)
attributetype ( 6.6.6.6 NAME      'SlaStartTime'
EQUALITY generalizedTimeMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
SINGLE-VALUE)
attributetype ( 6.6.6.7 NAME      'SlaEndTime'
EQUALITY generalizedTimeMatch
SYNTAX 1.3.6.1.4.1.1466.115.121.1.26
SINGLE-VALUE)

```

## A.2 Schema objectclass

### A.2.1 BandwidthAllocationRequest

```

objectclass ( 7.7.7.1 NAME      'BandwidthAllocationRequest'
SUP          top
STRUCTURAL
MUST        (SlaId $ BarId $ BarStartTime $ BarEndTime $
SrcAddr $ SrcAddrMask $ DstAddr $
DstAddrMask $ PepAddr $
Dscp $ Protocol $ SrcPortMin $
SrcPortMax $ DstPortMin $
DstPortMax $ BurstSize $ BitRate $
Interval $ Priority $ FilterNegation))

```

### A.2.2 ServiceLevelAgreement

```

objectclass ( 7.7.7.2 NAME      'ServiceLevelAgreement'
SUP          top
STRUCTURAL
MUST        (SlaId $ SlaStartTime $
SlaEndTime $SrcAddr$
SrcAddrMask $ DstAddr $
DstAddrMask $ Dscp $ Protocol $
SrcPortMin $ SrcPortMax $
DstPortMin $ DstPortMax $
MaxBurstSize $ MaxNRes $
MaxBitRate $ MaxOneBitRate))

```

# Appendix B: XML Policy DTD

## B.1 BandwidthAllocationRequest

```
<!ELEMENT BandwidthAllocationRequest EMPTY>
<!ATTLIST BandwidthAllocationRequest
  SlaId CDATA #REQUIRED
  BarId CDATA #REQUIRED
  BarStartTime CDATA #REQUIRED
  BarEndTime CDATA #REQUIRED
  SrcAddr CDATA #REQUIRED
  SrcAddrMask CDATA #REQUIRED
  DstAddr CDATA #REQUIRED
  DstAddrMask CDATA #REQUIRED
  PepAddr CDATA #REQUIRED
  Dscp CDATA #REQUIRED
  Protocol CDATA #REQUIRED
  SrcPortMin CDATA #REQUIRED
  SrcPortMax CDATA #REQUIRED
  DstPortMin CDATA #REQUIRED
  DstPortMax CDATA #REQUIRED
  BurstSize CDATA #REQUIRED
  BitRate CDATA #REQUIRED
  Interval CDATA #REQUIRED
  Priority CDATA #REQUIRED
  FilterNegation CDATA #REQUIRED>
```

## B.2 ServiceLevelAgreement

```
<!ELEMENT ServiceLevelAgreement EMPTY>
<!ATTLIST ServiceLevelAgreement
  SlaId CDATA #REQUIRED
  SlaStartTime CDATA #REQUIRED
  SlaEndTime CDATA #REQUIRED
  SrcAddr CDATA #REQUIRED
  SrcAddrMask CDATA #REQUIRED
  DstAddr CDATA #REQUIRED
  DstAddrMask CDATA #REQUIRED
  Dscp CDATA #REQUIRED
  Protocol CDATA #REQUIRED
```

```
SrcPortMin CDATA #REQUIRED
SrcPortMax CDATA #REQUIRED
DstPortMin CDATA #REQUIRED
DstPortMax CDATA #REQUIRED
MaxBurstSize CDATA #REQUIRED
MaxNRes CDATA #REQUIRED
MaxBitRate CDATA #REQUIRED
MaxOneBitRate CDATA #REQUIRED>
<!ELEMENT top (
  BandwidthAllocationRequest |
  ServiceLevelAgreement)*>
```

## B.3 top

```
<!ELEMENT top (
  BandwidthAllocationRequest |
  ServiceLevelAgreement
)*>
```

## Appendix C: Example XML Data

```
<!-- test xml data -->
<?xml version="1.0"?>
<!DOCTYPE top SYSTEM "policy.dtd">
<top>
  <ServiceLevelAgreement
    SlaId="FASTER_DEMO"
    SlaStartTime="01/20/02 18:00:00"
    SlaEndTime="02/20/02 18:00:00"
    SrcAddr="130.230.52.0"
    SrcAddrMask="255.255.255.128"
    DstAddr="130.230.83.0"
    DstAddrMask="255.255.255.0"
    Dscp="-1"
    Protocol="-1"
    SrcPortMin="0"
    SrcPortMax="65536"
    DstPortMin="0"
    DstPortMax="65536"
    MaxBurstSize="20000"
    MaxNRes="1000"
    MaxBitRate="10000"
    MaxOneBitRate="10000"/>
  <BandwidthAllocationRequest
    SlaId="FASTER_DEMO"
    BarId="0"
    BarStartTime="01/21/02 16:05:00"
    BarEndTime="01/21/02 16:06:55"
    SrcAddr="130.230.52.27"
    SrcAddrMask="255.255.255.255"
    DstAddr="130.230.83.210"
    DstAddrMask="255.255.255.255"
    PepAddr="130.230.52.1"
    Dscp="-1"
    Protocol="-1"
    SrcPortMin="0"
    SrcPortMax="65536"
    DstPortMin="0"
    DstPortMax="65536"
    BurstSize="200"
    BitRate="2000"
    Interval="10"
    Priority="1"
    FilterNegation="2"/>
</top>
```

# Appendix D: Localtree C structures

## D.1 Struct bar

```
struct bar{
    char                *ldapDn;
    char                *slaId;
    char                *barId;
    struct tm           BarStartTime;
    struct tm           BarEndTime;
    struct in_addr      SrcAddr;
    struct in_addr      SrcAddrMask;
    struct in_addr      DstAddr;
    struct in_addr      DstAddrMask;
    int32_t             Dscp;
    int32_t             Protocol;
    uint32_t            SrcPortMin;
    uint32_t            SrcPortMax;
    uint32_t            DstPortMin;
    uint32_t            DstPortMax;
    int32_t             BurstSize;
    int32_t             BitRate;
    uint32_t            Interval;
    uint32_t            Priority;
    enum pib_truth_value FilterNegation; /* PIB_TRUE || PIB_FALSE, see pib.h */
    uint32_t            install;
    uint32_t            remove;
    struct sla          *slaroot;
    struct sls          *slsroot;
};
```

## D.2 Struct pep

```
struct pep{
    uint32_t            MeterSucceedNextInstance;
    uint32_t            MeterFailNextInstance;
    uint32_t            ClassifierId;
    struct conn         *conn;
};
```

## D.3 Struct sla

```
struct sla{
    char                *ldapDn;
    int                 (*init)(struct sla *sla);
    void                (*free)(struct sls *sls,char *SlaId);
    char                *SlaId;
    struct tm           SlaStartTime;
    struct tm           SlaEndTime;
    struct in_addr      SrcAddr;
    struct in_addr      SrcAddrMask;
    struct in_addr      DstAddr;
    struct in_addr      DstAddrMask;
    int32_t             Dscp;
    int32_t             Protocol;
    uint32_t            SrcPortMin;
    uint32_t            SrcPortMax;
    uint32_t            DstPortMin;
    uint32_t            DstPortMax;
    int32_t             MaxBurstSize;
    uint32_t            MaxNRes;
    int32_t             MaxBitRate;
    int32_t             MaxOneBitRate;
    struct sls          *slsroot;
    dict                *bartree;
};
```

## D.4 Struct times

```
struct times{
    struct tm           *time;
    dict                *events;
};
```

## D.5 Struct event

```
struct event{
    struct tm           *time;
    char                *BarId;
    struct bar          *data;
    struct sls          *slsroot;
    struct sla          *slaroot;
};
```

## D.6 Struct sls

```
struct sls{
    int                                (*init)(struct sls *sls,
                                             uint32_t MaxBar,
                                             uint32_t bwLimit,
                                             LDAP *ld,
                                             char *ldaphost,
                                             int ldapport,
                                             int version,
                                             char *binddn,
                                             char *passwd,
                                             char *base, int verbose);

    void                                (*free)(struct sls *sls);
    int                                (*addsla)(struct sls *sls,struct sla *sla);
    int                                (*delsla)(struct sls *sls,char *SlaId);
    struct sla                          *(*getsla)(struct sls *sls,char *SlaId);
    int                                (*checksla)(struct sls *sls,
                                                  struct sla *sla,int bwLim);

    int                                (*countsla)(struct sls *sls);
    void                                (*printsla)(struct sla *sla);
    int                                (*addbar)(struct sls *sls,struct bar *bar);
    int                                (*delbar)(struct sls *sls,char *SlaId,
                                             char *BarId);

    struct bar                          *(*getbar)(struct sls *sls,char *SlaId,
                                                  char *BarId);
    int                                (*checkbar)(struct sls *sls,struct bar *bar,
                                                  int bwLim);

    int                                (*countbar)(struct sla *sla);
    void                                (*printbar)(struct bar *bar);
    int                                (*addDN)(struct sls *sls,struct dn *dn);
    int                                (*delDN)(struct sls *sls,char *name);
    struct dn                            *(*getDN)(struct sls *sls,char *name);
    void                                (*printDNtree)(struct sls *sls, FILE *fp);
    void                                (*delDNtree)(struct sls *sls);
    void                                (*dn2ldap)(struct sls *sls,struct dn *dn);
    int                                (*getldapdata)(struct sls *sls);
    struct event *(*get_first_event)(struct sls *sls);
    int                                (*countevent)(struct sls *sls);
    int                                (*delrefs)(struct sls *sls, struct bar *bar);
    void                                (*stat)(struct sls *sls);
    uint32_t                             MaxBar;
    uint32_t                             bwLimit;
    struct dn                             *dnlist;
    /* ldap connection structure */
    LDAP                                  *ld;
    char                                  *ldaphost;
    char                                  *binddn;
    char                                  *passwd;
```

```
char          *base;
uint32_t     ldapport;
uint32_t     version;
int32_t      verbose;
/* tree's */
dict         *slatree;
dict         *time_tree;
dict         *dntree;
int          refs;
};
```

## Appendix E: main\_loop

```
int main_loop(struct sls *sls)
{
    struct pep cfg;
    struct event *event = NULL;
    struct bar_time next_time;

    int ret;
    struct conn *conn = NULL;
    struct cops_conn *new = NULL;
    next_time.time_to_go_bar = 0;

    /* get first event */
    event = sls->get_first_event(sls);

    for (;;) {
        int fd;
        fd_set rfds, wfds;

        FD_ZERO(&rfds); FD_ZERO(&wfds);
        FD_SET(listen_socket, &rfds);
        for (conn = conns; conn != NULL; conn = conn->next)
            FD_SET(cops_get_conn_fd(conn->conn), &rfds);

        /* if event tree empty, get new data from ldap */
        if(event == NULL && sls->countevent(sls) == 0)
        {
            if(sls->getldapdata(sls) == 0)
                /* get next event */
                event = sls->get_first_event(sls);
        }
        /* set next timer: cops KA or event */
        next_time = set_timer(event);

        ret = select(15, &rfds, NULL, NULL, (next_time.next_time.tv_sec < 0) ?
                    NULL : &next_time.next_time);
        if (ret < 0){
            perror("select");
            return 1;
        }

    start:
        for (conn = conns; conn != NULL; conn = conn->next) {
            if (!FD_ISSET(cops_get_conn_fd(conn->conn), &rfds))
                continue;
        }
    }
```

```

ret = handle_packet(conn->conn);

if (ret <= 0) {
    close(cops_get_conn_fd(conn->conn));
    cops_pdp_free_conn(conn->conn);
    conns = remove_conn(conns, conn);
    goto start;
}
}
if(event != NULL && next_time.time_to_go_bar == 0){
    /* time to install bar */
    if(event->time == &event->data->BarStartTime){
        for (conn = conns; conn != NULL; conn = conn->next){
            cfg.MeterSucceedNextInstance = 1;
            cfg.MeterFailNextInstance = 2;
            cfg.ClassifierId = 1;
            cfg.conn = conn;
            printf("\nINSTALL:\n");
            /* install filter */
            install_filter(event->data,&cfg);
        }
    }
    else{
        /* time to remove bar */
        for (conn = conns; conn != NULL; conn = conn->next){
            printf("\nREMOVE:\n");
            /* remove filter */
            //remove_filter(event->data,&cfg);
            /* delete bar, if not needed */
        }
        if(event && event->data->install == 0 && event->data->remove == 0){
            /* delete bar from ldap tree */
            ldapDelete (sls->ld,event->data->ldapDn,0,sls->verbose);
            /* delete bar from local tree */
            sls->delbar(sls,event->data->SlaId, event->data->BarId);
            event = NULL;
        }
    }
    /* get next event */
    event = sls->get_first_event(sls);
}
/* if not time to go bar, do cops-KA */
else if (ret == 0) {
    cops_do_timers();
}
/* Did we receive a new connection? */
if (FD_ISSET(listen_socket, &rfd)) {
    fd = accept(listen_socket, NULL, NULL);
    if (fd < 0) {
        perror("accept");
        return 1;
    }
}

```

```
    }
    new = cops_pdp_alloc_conn(fd);
    if (new == NULL) {
        printf("cops_pdp_add_conn: out of memory\n");
        return 1;
    }
    conns = add_conn(conns, new);
}
}
return 0;
}
```

## Appendix F: Example script for EF Edge

```
#!/bin/sh
tc qdisc del dev eth0 root
tc qdisc add dev eth0 root handle 1:0 dsmark indices 64 default_index 2
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    police rate 5Mbit burst 200k drop \
    match ip dst 130.230.83.210 \
    match ip src 130.230.52.27 \
    classid :1
tc filter add dev eth0 parent 1:0 protocol ip prio 1 u32 \
    police rate 5Mbit burst 200k drop \
    match ip src 130.230.83.210 \
    match ip dst 130.230.52.27 \
    classid :1

tc class change dev eth0 classid 1:1 dsmark mask 0x3 value 0xb8
tc class change dev eth0 classid 1:2 dsmark mask 0x3 value 0x0

tc qdisc add dev eth0 parent 1:0 handle 2:0 cbq bandwidth 100Mbit avpkt 1000
tc class add dev eth0 parent 2:0 classid 2:1 cbq bandwidth 100Mbit \
    rate 2000kbit weight 1 avpkt 1000 prio 1 bounded allot 1514
tc class add dev eth0 parent 2:0 classid 2:2 cbq bandwidth 100Mbit
    rate 50000kbit weight 1 avpkt 1000 prio 7 bounded allot 1514
tc filter add dev eth0 parent 2:0 protocol ip prio 1 handle 1 \
    tcindex classid 2:1
tc filter add dev eth0 parent 2:0 protocol ip prio 2 handle 2 \
    tcindex classid 2:2
```

## Appendix G: Example script for EF Core

```
#!/bin/sh
# ef for core routers
for i in eth0 eth1 eth2; do
tc qdisc del dev $i root
tc qdisc add dev $i handle 1:0 root dsmark indices 64 set_tc_index
tc filter add dev $i parent 1:0 protocol ip prio 1 tcindex mask 0xfc \
  shift 2
tc qdisc add dev $i parent 1:0 handle 2:0 cbq bandwidth 10Mbit avpk \
  1000 mpu 64 allot 9200

tc class add dev $i parent 2:0 classid 2:1 cbq bandwidth 10Mbit rate \
  8Mbit avpkt 1000 prio 1 bounded isolated allot 9200

tc class add dev $i parent 2:0 classid 2:2 cbq bandwidth 10Mbit rate \
  5Mbit avpkt 1000 prio 2 allot 9200
tc qdisc add dev $i parent 2:1 pfifo limit 500

tc filter add dev $i parent 2:0 protocol ip prio 1 handle 0x2e tcindex \
  classid 2:1 pass_on
tc filter add dev $i parent 2:0 protocol ip prio 2 handle 0 tcindex
  mask 0 classid 2:2 pass_on
done
```