Prototyping the DiffServ MIB

Remco van de Meent

Internet NG
Work Unit 2
Deliverable D2.15

University of Twente
Department of Computer Science
Telematics Systems and Services Group

Abstract

An increasing demand for Quality of Service on the Internet has led to various developments in that area. Differentiated Services is a technique to provide such Quality of Service in an efficient and scalable way.

Management of computer networks involves both monitoring of running services as well as the configuration of those services. On the Internet, the SNMP protocol is used to retrieve and set variables in a MIB. In order to facilitate the management of routers equipped with Differentiated Services, the IETF has created the DiffServ MIB (which is still work in progress).

This assignment involves building a prototype implementation of the DiffServ MIB using a router running the the GNU/Linux operating system, using the Network Traffic Control facilities in the kernel and the net-snmp SNMP agent software.

The IETF diffserv WG is still working on the DiffServ MIB. The result of implementation work is valuable to the MIB authors as it may help in improving the MIB specification. Therefore any results should be reported back to the IETF community.

Samenvatting

Internet heeft de afgelopen decennia een sterke groei doorgemaakt. Een gemis dat een steeds grotere rol speelt, is het gebrek aan zogenaamde Quality of Service: een garantie voor de gebruiker dat een bepaalde kwaliteit dienstverlening gewaarborgd is. Met name bij toepassingen als telefonie en multimedia over Internet is Quality of Service erg belangrijk.

Een belangrijk aspect in dit gebeuren is de ontwikkeling binnen de IETF, de wereldwijde standaardisatieorganisatie voor Internet en Internet-gerelateerde protocollen, van Differentiated Services. Met deze techniek wordt het mogelijk om op een efficiënte en schaalbare manier onderscheid te maken tussen diverse diensten, en daardoor een zekere kwaliteit te kunnen garanderen.

Bij het beheren van computernetwerken wordt veel gebruik gemaakt van het SNMP protocol, zo ook bij het beheren van Differentiated Services. Wàt er precies beheerd kan worden wordt gedefinieerd door een Management Information Base (MIB). Deze doctoraal-opdracht behelst het onderzoeken van de mogelijkheden om de DiffServ MIB te implementeren, alsmede de uitslag van dit onderzoek aan de IETF te rapporteren.

De opdracht wordt uitgevoerd op een DiffServ router gebaseerd op het Linux besturingssysteem, met behulp van de net-snmp SNMP software.

Contents

Chapter 1

Introduction

1.1 Background

Today's Internet provides a best effort service. It processes traffic as quickly as possible, but there is no guarantee at all about timeliness or actual delivery: it just tries its best. However, the Internet is rapidly growing into a commercial infrastructure, and economies are getting more and more dependend on a high service level with regard to the Internet. Massive (research) efforts are put into transforming the Internet from a best effort service into a network service users can really rely upon.

Commercial demands gave rise to the idea of having various classes of service. For instance one can imagine that companies might offer (or buy, for that matter) either a gold, silver or bronze service level. Each of them having their own characteristics in terms of bandwidth and latency with regard to network traffic. This is called Quality of Service (QoS).

The Internet Engineering Task Force (IETF), one of the main driving forces behind Internet related technologies, has proposed several architectures to meet this demand for QoS. Integrated Services and Differentiated Services, developed in the "intserv" and "diffserv" IETF Working Groups, are probably the best known models and mechanisms.

The IETF diffserv WG has also defined a DiffServ Management Information Base, a virtual storage place for management information regarding DiffServ. At time of writing, this MIB is still work in progress. This assignment contributes to the development of the DiffServ MIB by writing a prototype implementation of a DiffServ MIB agent and giving feedback to the IETF community. One of the likely uses of the DiffServ MIB is that it may act as part of a bigger policy-based management framework. Therefore an implementation of the DiffServ MIB might also help development in that area.

Note that Quality of Service is not the only solution when talking about having an internet that can do better than just "best effort". Another well-known and deployed idea is just throwing bandwidth at it: make sure that there is enough bandwidth for every user of the network, and nobody will complain and hence a best effort network service will suffice...

1.2 Assignment

William Stallings wrote in [27] that a large network cannot be put together and managed by human effort alone. Automated network management tools are necessary to accomplish these tasks in complex systems. The model that is used for TCP/IP network management includes the following key elements:

1. Management Station
2. Management Agent
3. Management Information Base
4. Network Management Protocol

This assignment focuses on a prototype implementation of a management agent for Differentiated Services, given the (draft) DiffServ MIB and SNMP as "implementations" of the 3rd and 4th elements. In 2000, Oscar Sanz has carried out a feasibility study of this work[24]. Some work has been done on a manager application (for more information, see[21]) for a DiffServ MIB agent, but given the current status of the DiffServ MIB it'll take some time, although existing generic commandline utilities like `snmpget` can be used as well.

A prototyping environment is selected and a prototype implementation is developed. This prototype focuses on the "monitoring" part of the DiffServ MIB. It is not possible to do DiffServ configuration using this MIB implementation. The following aspects are the goals this M.Sc assignment:

- Does the DiffServ MIB, with this prototype implementation, solve the management issues it is intended to address? In other words, is it possible to manage a DiffServ router with the MIB, especially in the selected prototyping environment?
- Instrumenting the MIB might cause discovery of some "problems" with the current draft version, like under- or over-specification. The IETF DiffServ Working Group will be interested, so giving them feedback about the results of this work is important.

1.3 Approach

The prototyping environment chosen for this assignment is a DiffServ MIB Agent on a Linux-based router, using the `net-snmp` suite. Some motivations for using these software packages:

- The Linux operating system is freely available, including all of its source code. Linux has an excellent network traffic control infrastructure, support for differentiated services being part of it. The availability of the source code makes it possible to have a close and in-depth look at the DiffServ functionality in the kernel (since the kernel version 2.4 series).
- Linux, although being a server and desktop operating system by nature, is gaining influence on the marketplace for Internet routers as well. Many of the protocol implementations running on Cisco (et al.) routers, e.g. BGP routing, are also implemented on Linux systems nowadays. Hence it is likely that Linux' share will continue growing in the future, but if and only if implementors keep up with the developments in the global router market.
- `net-snmp` is a so-called "open source" implementation of SNMP and provides applications with a relatively easy to use interface for SNMP communication. The suite supports SNMPv1, version 2 with community-based security and version 3.

This makes Linux and `net-snmp` an obvious choice for a "proof of concept" implementation of the DiffServ MIB.

The DiffServ MIB Agent that has been developed may be downloaded from the Internet for free and is licensed under the GNU General Public License. URL for more information:

```
http://www.simpleweb.org/nm/education/assignments/previous-assignments.html
```

1.4 Report Structure

All the ideas, mechanisms, techniques and protocols mentioned in the assignment description will be explained in this report. Also the general structure of the resulting programming work will be discussed.

The report consists of 8 chapters. Chapter 2 starts with an introduction on Internet and Network Management. Then the Differentiated Services Architecture as well as the management of DiffServ will be discussed in chapters 3 and 4. The background of Linux Network Traffic Control is described in chapter 5. Subsequently the relation between DiffServ and Linux Network Traffic Control is discussed in chapter 6. After these theoretical chapters, the tools and techniques used for and an overview of the implementation will be given in chapter 7. Finally, the results and recommendations are presented in the last chapter, chapter 8. An outline is given in figure 1.1.

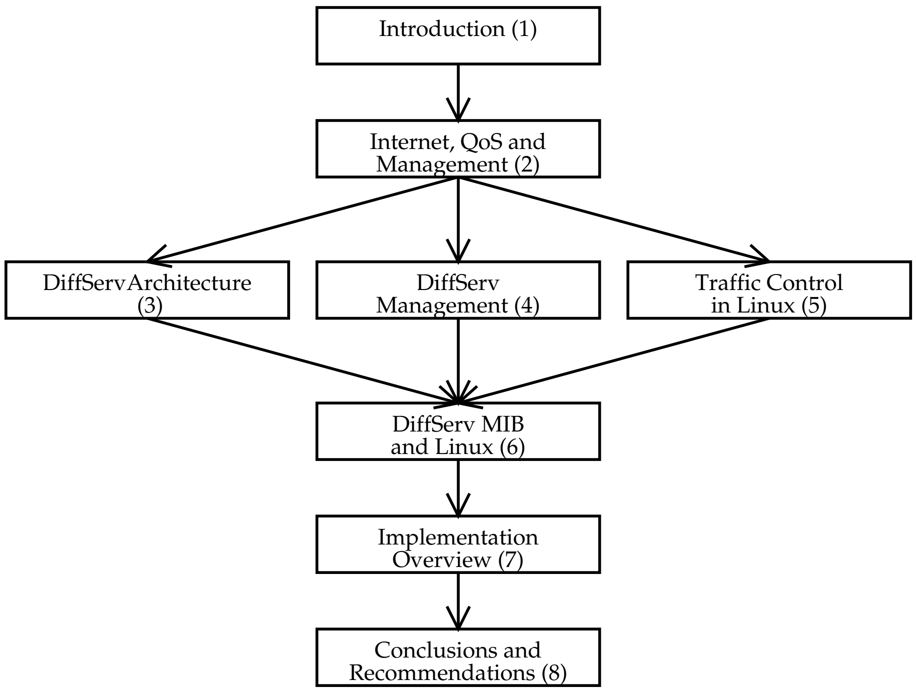

Introduction (1)
Internet, QoS and Management (2)
DiffServArchitecture (3)
DiffServ Management (4)
Traffic Control in Linux (5)
DiffServ MIB and Linux (6)
Implementation Overview (7)
Conclusions and Recommendations (8)

Figure 1.1: Structure of this report

Those who have background knowledge in the field of Internet Management may skip that chapter. Chapters 3, 4 and 5 describe the state of the art in the respective areas. From chapter 6 onwards, new work and ideas is presented.

1.5 Intended Audience

This report is intended for those who are:

- designing and implementing the DiffServ MIB
- interested in Quality of Service on the Internet
- interested in Network Management on the Internet
- interested in Differentiated Services and Management of DiffServ routers
- interested in SNMP Agent programming
- interested in Linux Network Traffic Control programming

This report may be read without a lot of background knowledge in these fields, as most if not all of the concepts are introduced and described in the report itself. Those who want to know more about the topics covered in this report may want to look at the literature list at the end of this report for more information.

Chapter 2

Internet, Quality of Service and Network Management

This chapter will give an introduction to the Internet Protocol (IP) and discusses various techniques for providing Quality of Service using IP. It then continues with an introduction to SMI and SNMP, terms that are closely related to Internet Management.

Something that won't be discussed is Internet Management itself. This is a huge research and operational area. More information about research in the Internet Management area can easily be found on the Internet, a good starting point being the SimpleWeb website[29].

2.1 IP Networks

"IP" stands for Internet Protocol. The Internet is the largest combination of computer networks ever built, and still rapidly growing. Enormous amounts of IP network traffic is sent over the Internet, while this figure seems to double every nine months.

The roots of the Internet go back to the early 60s, when the United States Department of Defense demanded a robust command and control network, linking computers from various DoD sites. This network was called the ARPA network, named after the DoD's research institute, the (Defense) Advanced Research Projects Agency. The network they created had a packet-switched architecture. This basically means that every packet on the network is routed independently of other, possibly related, packets; this is in contrast with circuit-switched like most public telephony networks are. During the 70s and 80s this network was extended and a lot of research efforts were undertaken to provide more and more services like naming schemes and electronic mail. The 90s and the 21st century so far have seen an incredible growth of the (public) Internet.

The Internet protocol stack, known as the *TCP/IP reference model*[28], is depicted in figure 2.1. A short description of these four layers follow.

- The *link layer* (sometimes called "host-to-network" layer) is not extensively discussed in this reference model. A host should be able to setup a connection with the network, using some protocol in order to be able to send IP packets through the network. The protocol used in this mechanism is varies from host to host and network to network. Well-known examples are Ethernet on a Local Area Network and PPP on dialup phone lines.
- On top of this access mechanism the *internet protocol layer* is stacked. This is the connectionless key element of the whole Internet architecture. It enables hosts to send packets to arbitrary networks, independent of each other in routing and sequence.

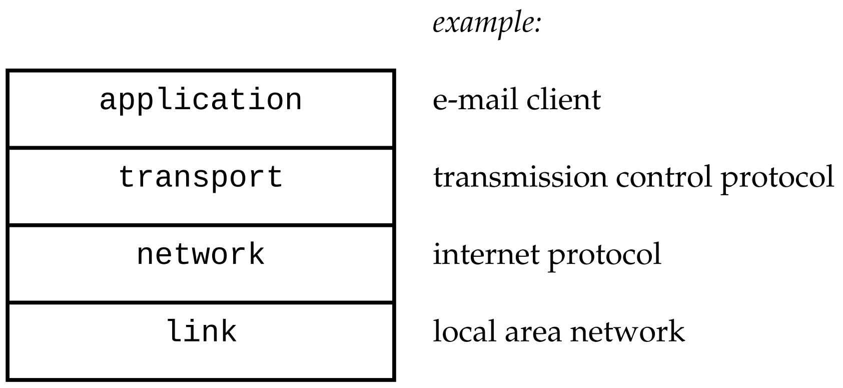


Figure 2.1: TCP/IP Reference Model

- The *transport layer's* sole purpose is to provide peer entities on both the source and destination hosts with services to have a "sensible" conversation with each other. There are two well-known transport protocols: TCP (Transport Control Protocol) and UDP (User Datagram Protocol). TCP is a reliable connection-oriented protocol, which means that it can deliver a stream of data without errors to the destination host. In case errors occur (like missing packets, or packets arriving in the wrong order), the TCP protocol will detect and correct them. UDP doesn't provide these services and is connectionless, but is used when a quick delivery is more important than accurate delivery. Errors might occur, but that's not always a problem, e.g. in case of voice data it doesn't really matter if one out of those many thousand packets get lost.
- On top of the stack, the *application layer* is what is most visible to the user. Examples are WWW browsers and e-mail clients. These applications make use of the underlying TCP transport layer to communicate with the server, using IP over PPP using a dialup telephony line, for example.

Within the scope of this assignment, the Internet Protocol is by far the most important layer in this model, hence a more detailed look at it is relevant. The current Internet uses IP version 4; development on its successor IPv6 is going strong however, and it's expected that IPv6 will take over the Internet in the decades to come.

An IP packet consists of a header and a data part, the latter one often referred to as payload. Figure 2.2 depicts the structure of an IP (version 4) packet. To get an idea of the size of such packets: each row in the header part is 16 bits, the option part has a variable size, hence the minimum size of an IP header is 20 bytes. The payload's maximum size is 64 kilobyte, but usually the packet's total size doesn't exceed 1500 octets.

Within the depicted header, the *type of service* field is important when speaking about Quality of Service in IP networks. Chapter 3 discusses this part in detail, as it is used by the Differentiated Services protocol.

Note that "on the wire" an IP packet is prepended by network technology dependent information, and the first part of the payload is usually filled with upper layer protocol information, e.g. a TCP header and HTTP (the protocol used when browsing the WWW) information. Protocol analyzers like Ethereal[13] may help in understanding the IP network protocol stack.

Now that the basic elements of the Internet Protocol have been outlined, it's time to talk about an enhancement to the services IP delivers to the user: Quality of Service.

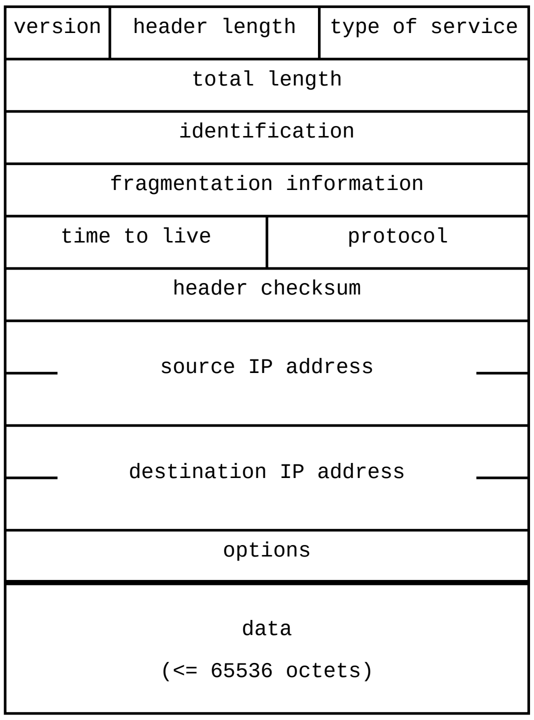


Figure 2.2: IP packet structure

2.2 Quality of Service in IP Networks

The Internet is, as mentioned before, a best-effort network. No guarantees about available resources are made. However, there is a clear need for relatively simple and coarse methods of providing different classes of service for Internet traffic, to support various types of applications, and specific business requirements. Quality of Service parameters are associated with every service request primitive (i.e. IP packet). They specify the performance of the network service the user expects from the network provider.

Within the IETF, various mechanisms that (help to) provide QoS services have been developed. Some of the more relevant models are outlined below.

- *Multiprotocol Label Switching* (MPLS) is an approach to apply label-switching to large-scale networks. The key concept of label-switching is identifying and marking IP packets with labels at the ingress of an MPLS domain, and basing any further forwarding in that MPLS domain on those labels[18].
- *Constraint-based routing* (also known as QoS routing) is used to identify an end-to-end path through a network (or series of networks) that has sufficient resources to satisfy a set of constraints (like available bandwidth and latency)[7].
- *Integrated Services* (sometimes called "intserv") provides end-to-end Quality of Service by means of resource reservation. Each traffic flow may request resources using the RSVP (resource reservation) protocol[16].
- *Differentiated Services* ("diffserv" or "DS") is an architecture that uses small and well-defined building blocks to provide QoS in networks. An extensive description of this mechanism can be found in chapter 3.

Before continuing with a detailed description of the diffserv protocol in the next chapter, an elaboration on QoS IP Networks follows.

Quality of Service is not something that can be achieved by adding diffserv functionality, or intserv for that matter, to just one component in the network. Packets sent across the Internet typically go through a large number of network devices such as hubs, switches and routers before reaching their final destination. To provide end-to-end QoS, it is necessary to have a guaranteed service level on every possible link and every possible device packets may go through on their way from one end to the other[1]. Given the packet switched architecture of the Internet, i.e. having little control over what links various packets in a flow of traffic are traversing, this is something that might be very hard to achieve.

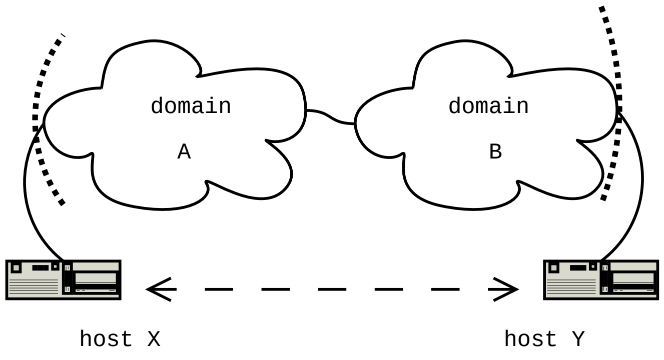


Figure 2.3: two hosts communicating over the Internet

Typically an ISP or network operator may provide so-called edge-to-edge Quality of Service. Figure 2.3 shows a situation where two hosts are connected via 2 different network providers who have their own transit networks, also referred to as 'domains". A domain may consist of a very complex mesh of network links and devices. When host X sends traffic to host Y, it first enters domain A at one of its edges (marked with the dotted line). Provider A now guarantees some level of service (depending on characteristics of the traffic flow) throughout its network. The packets leave the A domain at some other edge and enter the second domain. In order to guarantee the same level of service throughout network B as the packets got in the first domain, provider A and B should have some Service Level Agreement which regulates such exchanges of traffic. Operator B now sends it on to the destination host.

In short: edge-to-edge Quality of Service may be extended to combinations of networks, and ultimately towards (near) end-to-end QoS.

Network domains are bounded by edge devices. The main characteristic of such edge devices is that they have the capability to recognise packets based on predefined parameters (e.g. "this is voice-over-IP traffic") and take action based upon that recognition. Such action might be a certain routing decision, but also labeling packets belonging to that flow, etcetera.

[1]In practice passive network equipment like hubs will probably not support this, as there is little need for such support.

2.3 Structure of Management Information and MIBs

This section gives an introduction to the language that is used to describe "management information". But first the question arises what managent information actually is. Quoting [11], the document standardizing version 2 of SMI:

> Management information is viewed as a collection of managed objects, residing in a virtual information store, termed the Management Information Base (MIB). Collections of related objects are defined in MIB modules. These modules are written using an adapted subset of OSI's Abstract Syntax Notation One, ASN.1

An example of management information is some counter that keeps track of the number of octets that are sent through a certain network interface, in SMIv2 notation, is given in appendix A.

The Structure of Management Information is divided into three parts:

1. *Module definitions* are used to identify various modules, using the `MODULE-IDENTITY` element. Each MIB has a unique identifier. The example above was taken from the MIB identified by `ifMIB`, also known as the "Interfaces MIB". The module definition also keeps track of revisions and authors of that MIB.
2. *Object definitions* describe managed objects, e.g. the `ifOutOctets` counter `OBJECT-TYPE`. As it is important that no doubt may exist about what the managed object is, the definition includes a consise description and type definition, in the example the type is a 32 bit counter.
3. *Notification definitions* are used to describe the possibilities for unsollicited (see the next section) transmission of management information, e.g. between a manager and an agent, using the `NOTIFICATION-TYPE` ASN.1 macro.

A name-to-number mapping exists throughout all MIBs, in order to access specific instances of managed objects. These are called "object identifiers" (OIDs) and are administratively assigned. An OID consists of a prefix and a part pointing to a specific instance. E.g. to access the ifOutOctets counter of the second network interface concatenate `1.3.6.1.2.1.2.2.1.16` (the unique prefix for `ifOutOctets` counters) and `.2` (this table in the Interfaces MIB is indexed by the number of the network interface).

It is possible to specify new abstract datatypes for MIBs using SMIv2, though 11 base types have been defined:

```
INTEGER               IpAddress             TimeTicks
OCTET STRING          Counter32             Opaque
OBJECT IDENTIFIER     Gauge32               Counter64
                      Unsigned32            Integer32
```

It is not within the scope of this document to give a detailed description of all these types and their uses.

A MIB consists of a number of SMI constructs, and acts as a virtual information store for the objects described by those SMI constructs. As a MIB is hierarchically ordered, a common way to represent MIBs is a tree. An example can be found in figure 2.4, which depicts part of the Interfaces MIB.

Thousands of different MIBs have been specified now. A lot of them are published by the IETF in various RFC documents, but vendors may specify their own proprietary MIBs as well.

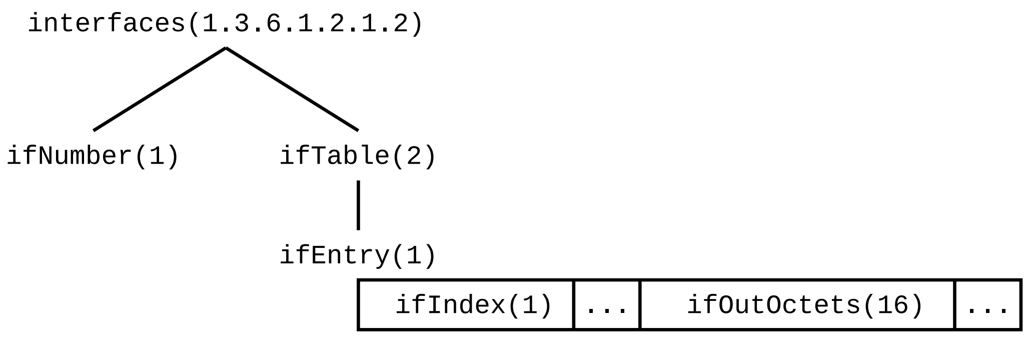


Figure 2.4: Part of the Interfaces MIB

2.4 Simple Network Management Protocol

The notion that computer networks and devices attached to networks should be managable has existed for about as long as those networks themselves. As mentioned in section 1.2, the model of network management that is used for TCP/IP network management includes the following key elements:

- Management Station
- Management Agent
- Management Information Base
- Network Management Protocol

The communication between the Management Station (*manager*) and Management Agent (*agent*) uses the Simple Network Management Protocol (SNMP), of which several versions exist. The relation between these elements is shown in figure 2.5.

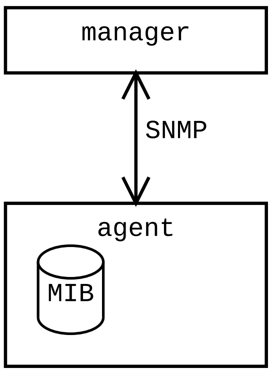


Figure 2.5: TCP/IP network management model

The only operations that are supported in SNMP are the inspection and alteration of variables. Three general-purpose operations may be performed on scalar objects:

- `Get`: a management station retrieves a scalar object value from a managed station

- `Set`: a management station updates a scalar object value in a managed station
- `Trap`: a managed station sends an unsolicited scalar object value to a management station

Note that access is only allowed to "leaf" objects, hence it's not possible to access an entire table in one atomic action for example.

These operations have resulted in the definition of 7 message types in the SNMPv2 protocol:

`GetRequest`	Request information about objects inside a MIB
`GetNextRequest`	Request information about the next object
`GetBulkRequest`	Request transfer of a potentially large amount of data
`SetRequest`	Request to set objects in a MIB
`Response`	Response to one of the four former messages
`SNMPv2-Trap`	Alarm message from an agent to warn the manager
`InformRequest`	Used for manager-to-manager communication

Authentication of SNMP messages is performed using *community strings*, which is kind of a password. An SNMP message has also some fields to give information about errors in processing the request. Figure 2.6 depicts the layout of an SNMPv2 message.

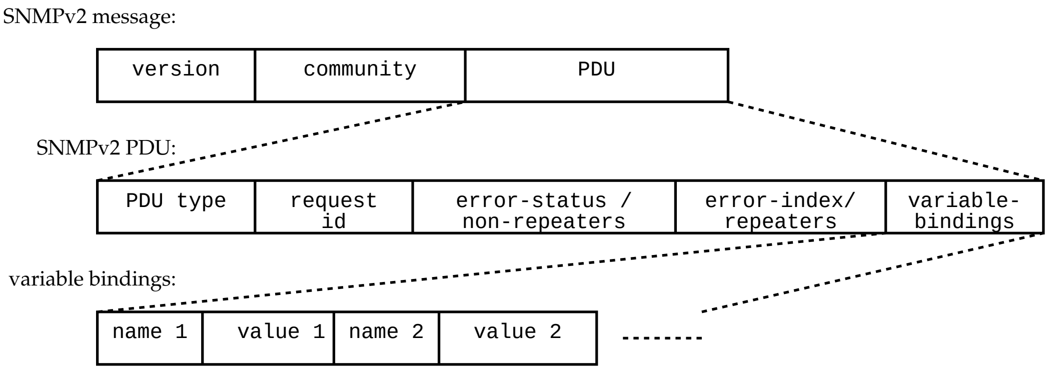


Figure 2.6: SNMPv2 message layout

The network management of an IP network consists of "network management stations" (*managers*) communicating with "network elements". The pieces of software running on those network elements faciliting management operations are called *agents*. This communication can be two-way: the manager may ask the agent for some information, or put some information into the agent, orthe other way around, when the agent sends some information to the manager without sollication thereof (notifications).

A little explanation about the internal workings of an SNMP agent is necessary to understand certain design decisions taken in this project. As discussed earlier, an SNMP agent uses a virtual information store, the MIB, to get and set values. The agent manages the devices (or certain parts of it) it is running on. However, it is unlikely that one piece of software can take full advantage of all the device's capabilities. Especially in multi-vendor environments, it is clear that one may want to run multiple SNMP agents on the same box. A possibility is to run the various agents of different UDP ports, but then the manager would have to talk to multiple agents as well. Hence another solution has been developed: the mechanism of master and slaves.

There is one master agent, and zero or more subagents. Each of these subagents implement a distinct part[2] of the MIB, and a master agent dispatches requests from the manager to the correct sub agent, as showed in figure 2.7 (taken from [25]).

[2]Note that this is not always true. In case of overlap certain algorithms need to solve this.

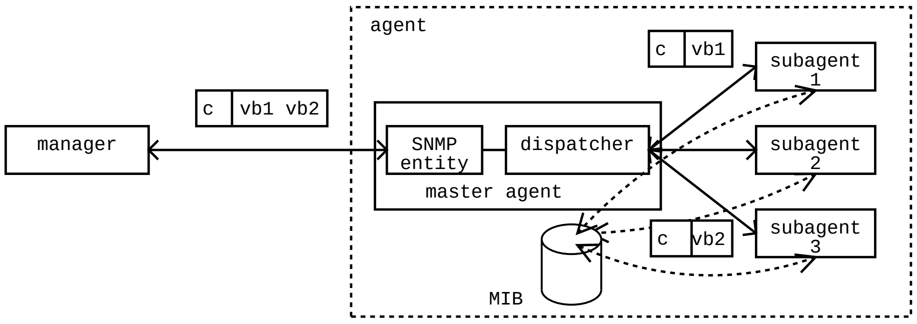


Figure 2.7: a manager, a master agent and three sub agents

Each of this sub agents is responsible for its own part of the MIB, and is more or less ignorant about the SNMP protocol spoken between the manager and (master) agent. Various protocols have been defined for master and sub agent interaction.

Suppose a sub agent receives a request from the master agent to return the value of a certain object identifier. It is likely that it doesn't know in the actual value of that object yet, e.g. when the `ifOutOctets` object is requested, the sub agent has to lookup this value (or a representation of it) in the kernel of the device.It is up to the sub agent to implement this, and there are no guidelines about protocols. Looking up values from the kernel might be a matter of parsing some (special) file, or even sending some other protocol messages to the kernel and parsing the results. The result of this operation is sent back to the master agent, who creates an SNMP PDU and passes it on to the manager. In the prototype implementation developed as part of the assignment, this technique of separating master and sub agent has been used.

Chapter 3

Differentiated Services Architecture

3.1 Introduction

The necessity of Quality of Service for the Internet, implemented in a simple and scalable way, has been stressed multiple times now. This chapter covers the DiffServ protocol: how does it achieve the quality of service it pretends to do while maintaining managebility and scalability?

The key question answered in this chapter is:

> *What is DiffServ and what can it do?*

DiffServ is a protocol for specifying and controlling network traffic by classes so that certain types of traffic get precedence. For example voice traffic, which requires a relatively uninterrupted flow of data, might get precedence over other kinds of traffic, like e-mail.

A Service Level Agreement (SLA) is a service contract between a customer and a service provider that, in this case, specificies the forwarding service a customer should receive. Part of an SLA may be a Traffic Conditioning Agreement that contains the more technical details of how the service contract is "executed". SLAs and DiffServ technologies are tightly coupled, as will become clear from this chapter.

3.2 Overview

The Differentiated Services Architecture lays the foundation for implementing service differentiation in the Internet in an efficient and scalable way. The IETF DiffServ Working Group charter[3] states:

> (...) "The differentiated services approach to providing quality of service in networks employs a small, well-defined set of building blocks from which a variety of aggregate behaviors may be built. A small bit-pattern in each packet, in the IPv4 TOS octet or the IPv6 Traffic Class octet, is used to mark a packet to receive a particular forwarding treatment, or per-hop behavior, at each network node. A common understanding about the use and interpretation of this bit-pattern is required for inter-domain use, multi-vendor interoperability, and consistent reasoning about expected aggregate behaviors in a network." (...)

The Differentiated Services architecture[6] is based on a relatively simple model. Traffic is classified when entering a network and possibly conditioned (e.g. shaped to a certain maximum rate) at the network's boundaries. Also behaviour aggregates are assigned. This means that a predefined number, the DiffServ Code Point (DSCP) is written into the first 6 bits of the IPv4 packets' Type-of-Service header field (figures 3.1 and 2.2). A behaviour aggregate is a collection of packets with the same DSCP value crossing a link in a particular direction.

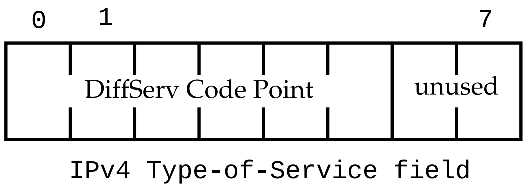


Figure 3.1: Structure of the DSCP field in an IPv4 header

The core of a network may consist of a mesh of links, routers, switches and other networking equipment. Each router packets traverse is called a "hop". Packets, classified at the edge of the network, are forwarded according to the so-called *per-hop behaviour* (PHB) throughout the core of the network. The PHB is associated with the DSCP value.

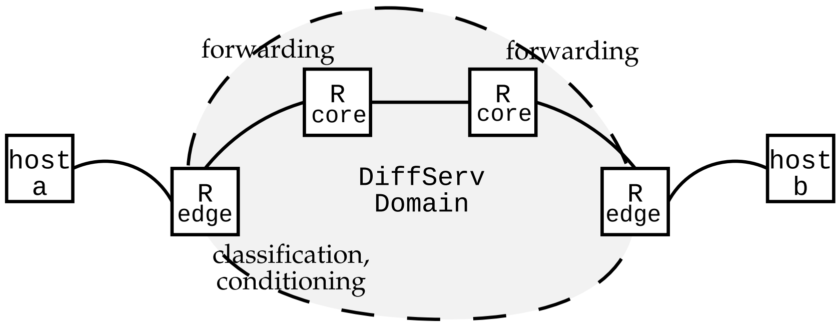


Figure 3.2: Traffic classification, condition and forwarding in a DiffServ Domain

Packets may be forwarded across multiple networks on their way from source to destination. Each of those networks is called a DiffServ Domain. Or more specific, a DiffServ Domain is a set of routers implementing the same set of PHBs. Obviously, SLAs between the various operators of those networks are needed if customers want a particular level of service: a DSCP value of 1 in the network A may be associated with an completely different level of service than it is in network B. Reassigning DSCP values when entering the next network is possible, but it would save resources if it is avoided[12]. Note that the inclusion of non-DiffServ-compliant nodes within the path a packet traverses may result in unpredictable performance and hence affect the ability of satisfying the SLA.

3.3 DiffServ Building Blocks

The two main elements of the DiffServ conceptual model are *Traffic Classification* and *Traffic Conditioning*. A DiffServ router consists of numerous functional elements that implement these tasks. This section discusses these building blocks, with configuration and management in mind.

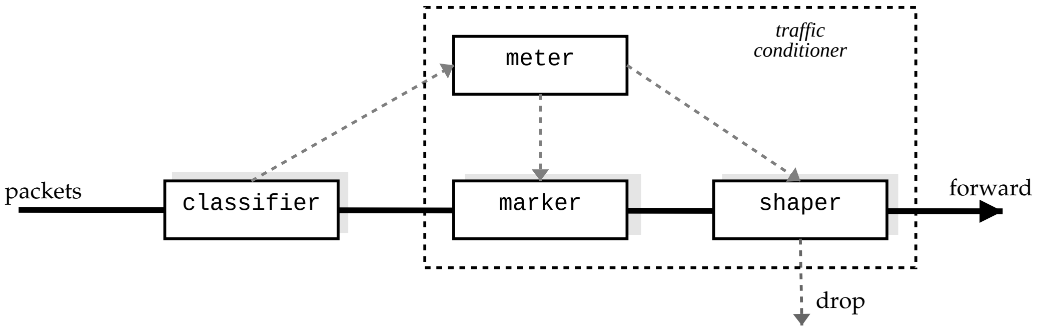


Figure 3.3: Logical View of a Traffic Classifier and Conditioner

In figure 3.3 the logical view of the Classifier and Conditioner elements is given. Packets, when they arrive at the ingress interface of a DiffServ router, typically get classified, after which some actions are performed before the packets are forwarded the next hop.

A more detailed look at these functional elements follows in the next subsections.

3.3.1 Packet Classification: Classifier

The packet classification policy identifies the subset of traffic which may receive a differentiated level of service by being conditioned and/or mapped to a specific DSCP value within the DiffServ domain.

Classification may be based on the content of the IP header. There are two types of classiers: 1. Behaviour Aggregate Classifiers, which classify packets based on the DSCP value; and 2. Multi-Field Classifiers, selecting packets based on the value of a combination of IP header fields, e.g. source and destination address.

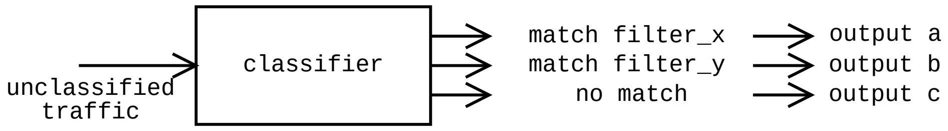


Figure 3.4: An example Classifier element

It will be clear that a Classifier takes a stream of traffic as input and steers them to the correct output, as is depicted in figure 3.4. In most cases these output channels are connected to the input channels of other functional elements of the DiffServ architecture.

3.3.2 Traffic Profiles: Meter

If the properties of a stream of packets are not exceeding certain predefined parameters, the packets are called in-profile (or in the opposite case: out-of-profile). So traffic profiles specify the (temporal) properties of a traffic stream (which is selected by a classifier), and provide rules for determining whether a particular packet is in-profile or not.

An example traffic profile is “maximum rate is 1Mbit/s, but allow bursts of 2Mbit/s with a duration of at the most 20 seconds, if they’re not within 1 minute of each other”. Different traffic

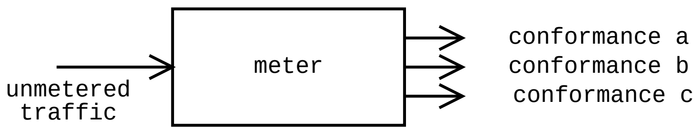


Figure 3.5: A generic Meter element

conditioning actions may apply depending on the result of applying such a traffic profile to a particular packet.

Figure 3.5 shows a very generic meter element. It takes a stream of traffic as input, and decides whether that stream is at that moment in time (temporal property) conforming, partially conforming, or not conforming at all to a certain traffic profile. The next functional elements in the DiffServ router for these three conformance levels might be different. A possible usage scenario is that non-conforming traffic is sent through a Counter element for out-of-band purposes like a billing application (to send a bill to a customer who is exceeding the traffic profile that was agreed upon)

3.3.3 Actions on packets: Marker, Counter, Multiplexer, Absolute Dropper

A group of elements that is performing actions on traffic streams, Action elements, is mostly called after the classification and/or metering phase.

- A *Marker* is an element that marks packets with a certain (preconfigured) DiffServ Code Point (DSCP) value. This is usually done at the edge of a DiffServ domain. Classification of packets within the core of that domain is done entirely and solely based on the value of this DSCP field in the IP header.
- The *Counter* element does nothing more than updating internal registers with the number of packets traversing through this building block. A manager may use this in order to get information about the number of packets belonging to a certain traffic class, which might be important for billing purposes.
- *Multiplexer* are used to multiplex or de-multiplex the (logical) stream of traffic, e.g. to combine the output of multiple functional elements as the input of a single counter.
- When it is necessary to drop packets regardless of their content e.g. because of traffic that is not conforming to a certain profile, the DiffServ architecture provides the user with an *Absolute Dropper* element that just discards any packets that arrive at its input.

The functions performed by each element is clear from the descriptions above. A complete overview can be found in the DiffServ Architecture document[6].

3.3.4 Traffic Conditioning: Queueing element

Traffic conditioning has to do with “shaping” the stream of network traffic according to a predefined set of rules, e.g. based on previous classification. This is often called policing. The functionality of the queueing element is split into sub-components:

- *FIFO queue*

This is the most simplistic form of a queue. Packets get sent in order of arrival at the queue: "first in, first out". This queueing technique is probably the most well-known and deployed of all possible algorithms.

Packets that leave this queue typically go to a Scheduler element in order to get sent over the network.

- *Scheduler*

  A scheduler uses an algorithm to determine in what order and at what time packets that arrive at its input are forwarded to the network (using the underlying operating system's network stack). Such algorithms are called service disciplines. Parameters that affect the operation of a scheduler include (but are not limited to): static parameters such as relative priority associated with the input channels of a scheduler, and dynamic parameters such as the DSCP value of packets currently at the input of a scheduler.

  Various categories of service discplines are documented, like "first come, first served", "rate based" and "weighted fair bandwidth sharing".

- *Algorithmic Dropper*

  Like the name says, this element selectively drops packets arriving at its input, using some selection algorithm. Note that the same functionality is achieved by selectively removing packets that are already in a FIFO queue.

  An algorithmic dropper may be put in front of or just after a (FIFO) queue, the former one known is known as a "tail dropper" (because it drops packets that are about to be added to the tail of a queue).

  Selection of packets that are to be dropped (or to be forwarded) is based on the result of some algorithm, that internally triggers the dropping of a packet. A lot of research is done in this area. A well-known algorithm is *RED* (Random Early Detection)[14]: the size of a queue is taken as input parameter to a function that calculates the probability whether a packet is dropped or not. The larger the queue, the higher this probability will be.

In figure 3.6 these three elements of the Queueing functional building block are working together.

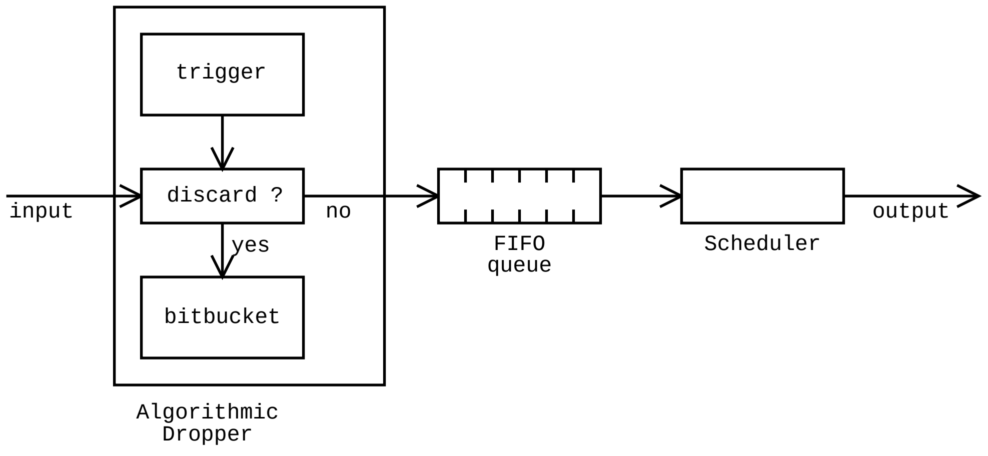


Figure 3.6: Queueing elements

It is probably clear now that a DiffServ Router is a complex technological artifact. Therefore the Service Level Agreement, describing what the service a user should get from the operator, is quite complex as well, unfortunately. If the SLA is ambiguous, it may lead to unpredictable performance etcetera. Hence the obvious need for juridically and technically "good" and "clear" agreements.

3.4 Two example PHBs

In the previous sections various properties and elements of the DiffServ architure have been discussed. The combination of building blocks in DiffServ capable nodes, is called a *Traffic Control Block*. Using TCBs, it's possible to implement a lot of different PHBs (Per-Hop Behaviours).

Two well-known PHBs are *Expedited Forwarding* and *Assured Forwarding*, that are described by the IETF diffserv WG in [10] and [9].

The Expedited Forwarding PHB (referred to as EF) gives the user a guaranteed minimal percentage of the link for usage by that user. It is suggested for (mission critical) applications that require a guarantee on the delay and jitter. This might be an expensive service to use, comparable to "premium service".

The Assured Forwarding PHB (AF) is somewhat more complex. It specifies four classes of traffic, and each of those classes is guaranteed a minimum amount of bandwidth and buffering, better than the common best-effort service. A typical use is that at the ingress of a DiffServ domain, classification and policing is performed, according to a Service Level Agreement with the customer. If the traffic is in-profile, packets are delivered with a relatively high probability of arrival and correct order. This is useful for companies that have e.g. a number of interconnected sites and want to have a reliable and relatively high performance link between these sites, up to a certain volume of aggregated traffic ("subscription rate").

Chapter 4

Differentiated Services Management

4.1 Introduction

Together with the need for service differentiation comes the need for management thereof. The network service providers delivering DiffServ services to their customers have to configure and monitor their routers in order to be able to satisfy the SLAs.

The IETF diffserv WG produced an informal management model[5] as well as a MIB[2] to help operators managing their DiffServ products. These documents are still “work in progress”.

It is likely that in practice DiffServ will be offered to customers as a way to make a distinction between various levels of service, e.g. Premium, Gold, Silver and Bronze. The IETF snmpconf WG[26] is chartered to write a Best Current Practices document describing the configuration management of network devices using SNMP. This includes Policy-Based Management. This chapter will discuss this important part of DiffServ configuration as well, although an implementation is outside the scope of this assignment.

4.2 DiffServ Informal Management Model

The DiffServ Informal Management Model[5] is based on the DiffServ Architecture[6] but focuses on configuration and management. In the architectural model, numerous functional elements have been defined. A combination of one or more of these building blocks results in a *datapath*. The Informal Management Model specifies the possible configuration parameters of these elements.

The conceptual model of a DiffServ router, including management elements, is depicted in figure 4.1. It can be concluded from this model that the main configurational aspects of DiffServ management are related to the configuration of the ingress and egress interfaces. Note that there are more aspects related to the configuration of a DiffServ domain. One might think of snooping other Quality-of-Service related packets from the network in order to gain knowledge used for configuration, e.g. RSVP packets. These might be useful to know what kind of traffic should be classified in some way. Such agents might be an input to the configuration as well, but this is outside the scope of this document.

For network interfaces, the DiffServ related parameters are divided in the parameters with information useful to the various building blocks: classification, metering, action and queueing elements (descriptions of these elements can be found in the previous chapter)

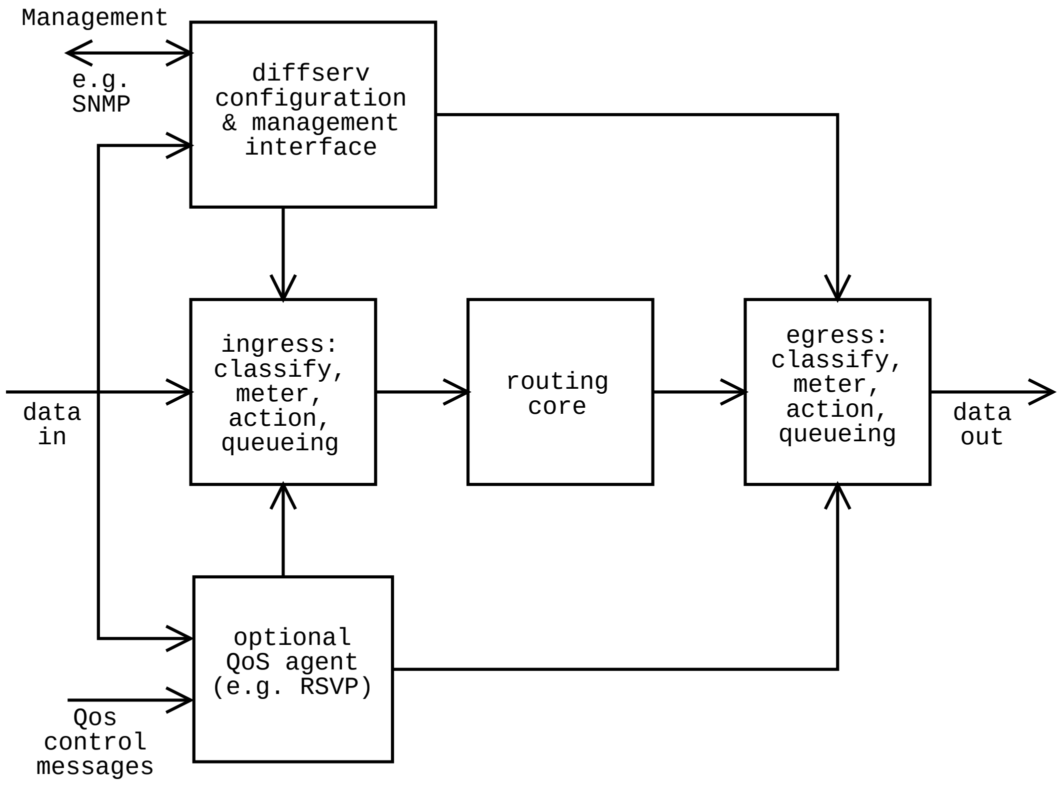


Figure 4.1: DiffServ Router Functional Blocks

- The *Classifier* element is based on filters to select matching and non-,matching packets. Based on this selection, packets are forwarded along the corresponding datapath.

  A filter is a set of conditions that take various parts of the packet (possibly header values, content, attributes relevant for classification) as input and determine the next element in the datapath by the output of this operation. Examples are using the packet's source and destination IP numbers and TCP port numbers on the edge (a so-called MF- or Multi-Field Classifier) of a DiffServ domain, and its DSCP value (which is put in the packet by an edge router) on the core routers (known as a BA- Behaviour Aggregate Classifier). Note that the use of transport protocol port numbers implies some other requisites like the prevention of IP fragmentation within the DiffServ domain.

  The primary issues for configuration of the Classification element are obvious: the parameters mentioned above, as well as the output (i.e. next functional element in the datapath) tied to traffic conforming to that particular filter. Note that it is, according to the archirural model, possible to combine filters for selection of packets by sequential application.

  An example configuration that makes a distinction between packets coming from the network 10.0.0.0/24 and other packets is given below:

  ```
  Filter Matched       Output Stream
  --------------       -------------
  Filter1              A
  no match             B

  Filter1:
  ```

```
    IPv4 Source Address:   10.0.0.0
    IPv4 Source Mask:      255.255.255.0
```

- *Metering* is a function that may be part of the datapath and is used to detect whether a traffic stream is in- or out-of-profile.

  Meters are parameterized by a temporal profile and by conformance levels. A common example is a meter with three conformance levels. The result is “green” when a stream is conforming, “yellow” when it’s partially conforming and “red” representing non-conformation. The various output results may be used to select different further treatment in the datapath (e.g. queueing or dropping).

  Thus the management interface has to provide means to specify these parameters. The key parameters are “rate” and “burstiness”. For example a configuration that measures the current rate (i.e. profile), which in fact takes the average rate during a certain time interval, of a traffic stream and selects the next functional element. In this example, burstiness is not taken into account.

```
  Meter1:
    Type:                AverageRate
    Profile:             Profile1
    ConformingOutput:    Queue1
    NonConformingOutput: Counter1

  Profile1:
    Type:                AverageRate
    AverageRate:         120 kbps
    Delta:               100 msec
```

- *Action* elements are operating on packets, i.e. they may mark, drop, or count packets, or simply do nothing.

  The configuration of these elements is straightforward. For example, the DSCP Marker should be configured with a certain DSCP value that it should put in packets it receives on its input, and the Absolute Dropper element just drops all packets it receives, so it doesn’t need any configuration at all. Note that this particular functional element has no successor in the datapath.

  The Counter element is useful for monitoring purposes. When part of a certain datapath, it may be used to count the number of packets that are treated by that particular datapath. Regular polling of this element gives information that might be useful for accounting purposes.

  An example configuration of the DSCP Marker Action element:

```
  Marker1:
    Type:                DSCPMarker
    Mark:                001100
```

- The *Queueing* elements are usually the last functional element in the datapath a packet traverses. They take care of modulating the transmission of packets that are part of different traffic streams, by prioritizing, storing and/or discarding and sending them.

  An example configuration for an algorithmic dropper that drops packets when a certain queue is too long is given below:

```
AlgorithmicDropper1:
  Type:                 AlgorithmicDropper
  Discipline:           drop-on-threshold
  Trigger:              Fifo1.depth > 10kbyte
  Output:               Fifo1
```

In figure 4.2 an example Traffic Control Block is depicted. From the picture it is clear that DiffServ configuration is a matter of combining the various functional elements in a datapath.

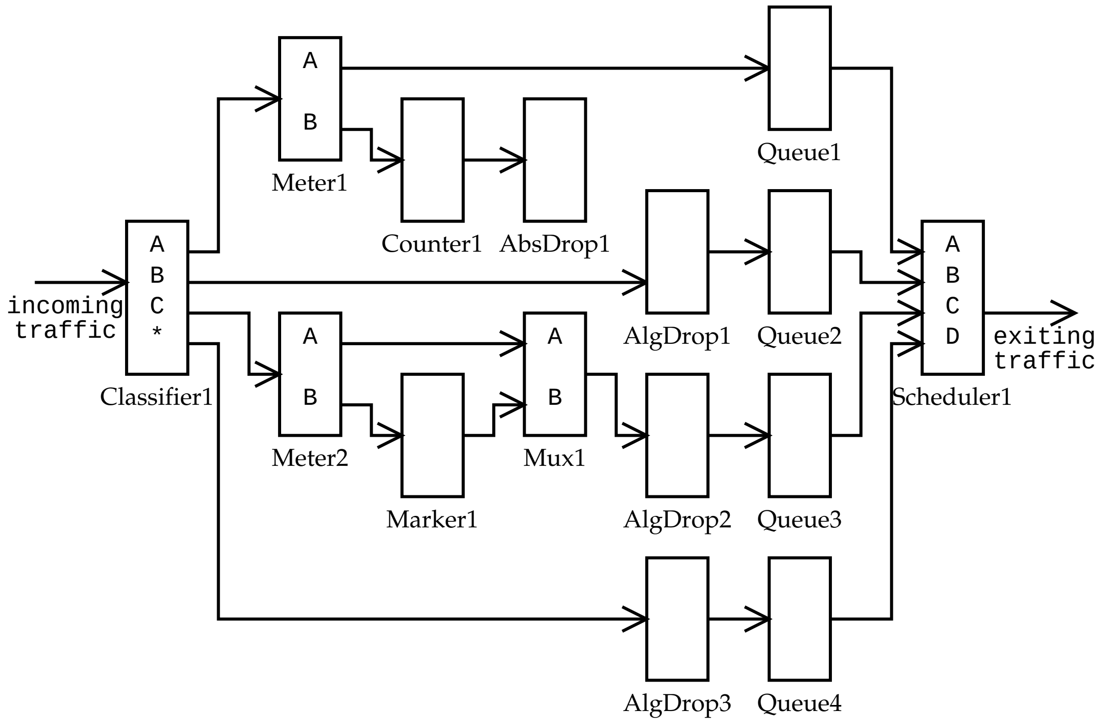


Figure 4.2: Example Traffic Control Block

One can imagine that the configuration of a DiffServ router might be very complex, due to the nature of the architecture. A lot of parameters affect the operations of a router. This implies that the configuration and management interface is not trivial. The overview of management and configuration parameters given above yields a more technical and formal specification, using the Structure of Management Information language: the DiffServ MIB.

4.3 DiffServ MIB

The DiffServ MIB describes the configuration and management aspects of devices implementing the DiffServ Architecture. Specifically, it talks about the configuration parameters of the various elements that are defined by that architecture. The current status of the MIB is Internet Draft, although it is expected to go onto the IETF Standards Track. The current (draft) version of the MIB[2] can be found in appendix B of this report in tree-format.

The DiffServ MIB contains the functional elements of the datapath, using various tables. The idea is that *RowPointers* are used to combine the various functional elements into one datapath. The elements of the MIB are described below (note that the ingress and egress portions of a DiffServ device are modeled identically).

4.3.1 Data Path Table

The Data Path Table contains the starting point of the DiffServ datapaths. This is realized by making seperate datapaths using the network interface traffic is received from or sent on, as well as the direction of the traffic stream.

This part of the MIB is displayed in figure 4.3, in which case the datapath starts with a classifier element.

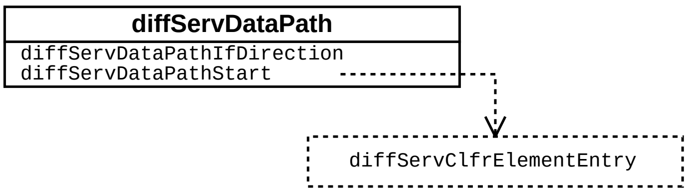


Figure 4.3: diffServDataPath

4.3.2 Classifier and Filter tables

The classifier table contains a framework that is extensible with multiple filters, by containing pointers to the tables with the specific filters (i.c. an IP Six-Tuple Multi-Field Classification Table). Figure 4.4 shows this framework.

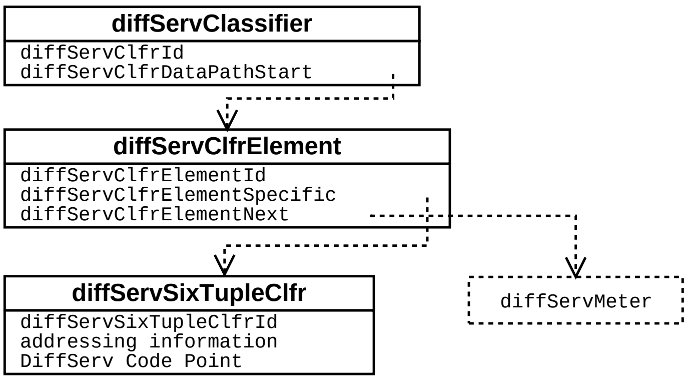


Figure 4.4: diffServClassifier

A classifier functional element may contain multiple elements, e.g. two MF-filters in a row. In that case the "next" element would be another instance of the classifier element object. In the picture there is only one filter, and the next functional element in the datapath is a Meter. The addressing information columns in the Six-Tuple Classifier table contain information about IP versions and addresses and network masks, as well as transport layer port-numbers. It is also possible with this filter to classify traffic based on the DiffServ Code Point (e.g. in the core routers of a DiffServ domain).

4.3.3 Meter Tables

The meter tables contain an extensible framework for the meter functional element, as well as an example parameterization table: the token bucket meter. In figure 4.5 part of the Meter table is shown.

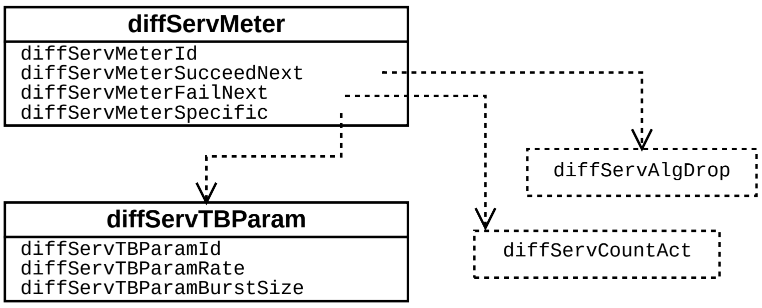


Figure 4.5: diffServMeter

From the picture it will be clear how the policing decision whether the traffic is in-profile or not is represented in the MIB. In this example, out-of-profile traffic is counted by a Counter element, whereas in-profile traffic is sent on to the Algorithmic Dropper element. This is accomplished by the use of RowPointers.

Metering is parameterized in the Token Bucket Parameter table (which in fact contains more parameters than only the columns shown in the picture).

4.3.4 Action Tables

The absolute dropper, DSCP marker and counter are functional elements that fall into this category. The “do-nothing” and ”(de)multiplexing” elements are also part of the Action building block, but are not represented in the MIB. Their behaviour can be expressed in the MIB using RowPointers however.

An outline is given of a Counter element in figure 4.6.

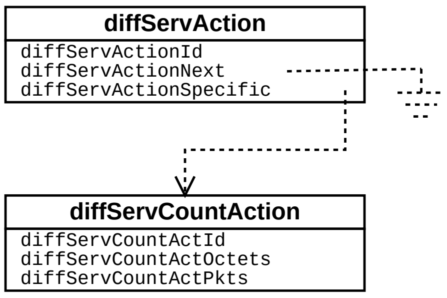


Figure 4.6: diffServAction

Such an element might be used just before an absolute dropper that follows a out-of-profile metering decision. An absolute dropper is represented by a `zeroDotZero` RowPointer (which points to nothing).

4.3.5 Queue, Scheduler and Algorithmic Dropper Tables

The queueing elements of the DiffServ architecture and management model include algorithmic droppers, queues and schedulers. These functional elements are represented in the MIB using various tables.

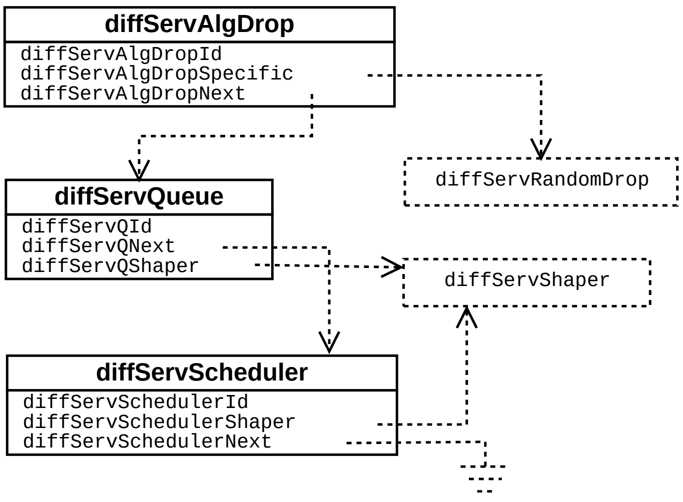


Figure 4.7: diffServAlgDrop, diffServQueue and diffServScheduler

Figure 4.7 gives an impression of how the Queueing element of the DiffServ architecture (see figure 3.6) is represented in the DiffServ MIB for management purposes. An algorithmic dropper uses a "random" trigger algorithm to drop packets, parameterized by the Random Dropper. Packets that are not dropped are sent on to a Queue that, together with the Scheduler, uses a Shaper for paramaterization of the queueing behaviour. The end of DiffServ treatment, like in this case the end of the functional datapath, is indicated by a `zeroDotZero` RowPointer (the "ground" symbol in the picture).

Note that not every single object in the DiffServ MIB has been discussed in this section. However it will be clear now how the MIB works, using RowPointers and various complex tables. For a complete (reference) description of the MIB, the document produced by the IETF diffserv WG should be used.

4.4 DiffServ Policy MIB

In the previous section the DiffServ MIB tree has been discussed. The MIB offers loads of configuration options, indicating and maybe adding to the complexity of deploying Differentiated Services (this is called instance-level configuration).

Within the IETF, the snmpconf WG[26] is chartered with writing a document about the most effective method of doing configuration management with SNMP, including Policy Based Management (for network-wide configuration). As proof of concept, the snmpconf WG has written a (draft version of a) DiffServ Policy MIB.

The DiffServ Policy MIB module provides a conceptual layer between high-level policy definitions that affect configuration of the DiffServ (sub)system and the instance-specific information that would include such details as the parameters for all the queues associated with each interface in a router. This gives an interface for DiffServ configuration at a conceptually higher layer.

A commonly used example is to make a distinction between various levels of service, e.g. Premium, Gold, Silver and Bronze. These are "templates" for the configuration of the instance-level DiffServ MIB, i.e. a Silver service could be defined as a datapath that implements the following requirements:

1. Match inbound traffic from IP number `10.0.0.1`, and limit this to a rate of 500kb/s and a maximum burst of 800kb, while marking with the DSCP value corresponding to the AF11 Per-Hop Behaviour.

2. Match all other inbound IP traffic, limit this to a rate of 64kb/s and a maximum burst of 80kb, while marking it with AF11 if the traffic in in-profile, otherwise drop the traffic.

Application of this service definition, which in fact is a policy decision, throughout the network is easy for the network manager, when using the DiffServ Policy MIB. The MIB takes care of the instance-specific (and complicated) configuration of the DiffServ nodes. This is achieved by having pointers to the correct datapath starting points in the DiffServ MIB from within the DiffServ Policy MIB, as outlined in figure 4.8.

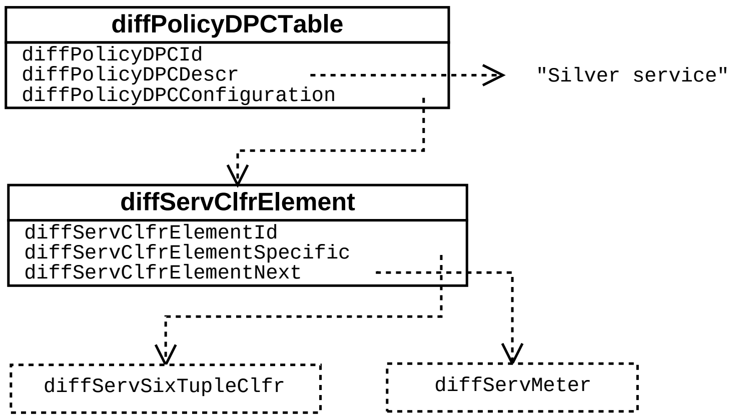


Figure 4.8: DiffServ Policy MIB and DiffServ MIB implementing the Silver service

The definition of the DiffServ Policy MIB can be found in [15]. It is expected that the DiffServ Policy MIB will become an important instrument to operators as it makes the job of DiffServ configuration a lot easier.

Chapter 5

Traffic Control in the Linux Kernel

5.1 Introduction

The GNU/Linux operating system offers a wide range of network traffic control functions. Alexey Kuznetsov designed and implemented an architecture called *Linux Network Traffic Control*, which contains mechanisms serving as a framework for supporting e.g. intserv and diffserv functionality[1].

The Swiss Federal Institute of Technology (École Polytechnique Fédérale de Lausanne, EPFL) has contributed code to the Linux kernel that implements Differentiated Services[4] using the Linux Network Traffic Control framework. Previously the code was provided as patches, but in the 2.4 series of the kernel it is incorporated in the mainstream kernel source code.

This chapter gives an overview of the architecture of the Network Traffic Control code and describes its structure, as well as the DiffServ specific parts. It is necessary to look into this as the structure of the DiffServ MIB is different from the DiffServ implementation in Linux. This conflict leads to problems that need to be solved in order to implement the DiffServ MIB. More information about these issues is to be found in section 5.3 and in chapter 6.

Note that at time of writing, the EPFL DiffServ support is the only open source implementation of Differentiated Services on Linux, though it is known that there is at least one other vendor that has done its own implementation of DiffServ on Linux.

5.2 Overview

Network traffic that is received by a host from the network is either destined for that host, or should be sent on to the next hop in case of a router (or is discarded if neither of these cases are applicable). This distinction is made clear in picture 5.1. If packets are forwarded, which is often a kernel-level process, numerous actions have to be carried out: selection of the output interface, selection of the next hop, encapsulation, etcetera. When that's done, packets are queued on the respective output interface. That's the point where Linux Network Traffic Control, *TC*, comes into play.

TC can, among other things, decide whether a packet should be queued or dropped (the latter for example in case where the traffic exceeds certain thresholds), in which order packets are sent (hence giving priority to certain network traffic flows) and it can delay the sending of packets (e.g. to limit the rate of outbound traffic). Once TC has released a packet for sending, the device driver picks it up and emits the packet to the network.

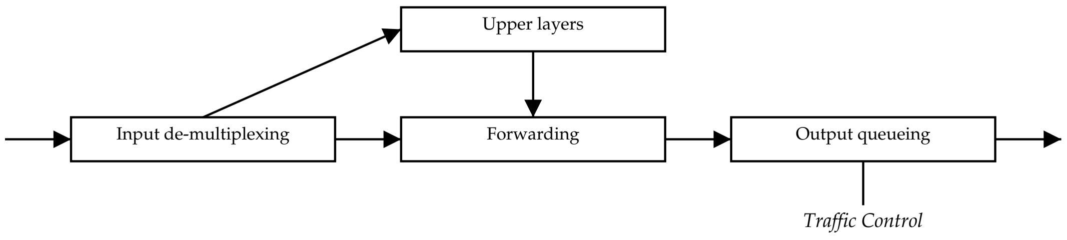


Figure 5.1: Processing of Network Traffic in Linux

The TC framework consists of four major conceptual components that are discusses in the following subsections:

- queueing disciplines
- classes
- filters
- policing

5.2.1 Queueing Disciplines

In a system with one or more network interface cards, each of these devices has a queueuing discipline attached to it. A queueing discipline determines in what manner packets enqueued for that device are treated, e.g. using the FIFO algorithm and sending packets as fast as the respective device can send.

A more sophisticated queueing discipline might use a filter to determine whether to forward the packet as fast as the interface permits or to enforce a specific maximum traffic rate, depending on the originating IP address of the packet (see subsection 5.2.3), hence possibly giving priority to one packet over another. An example of such a queueing discpline (qdisc) is given in figure 5.2.

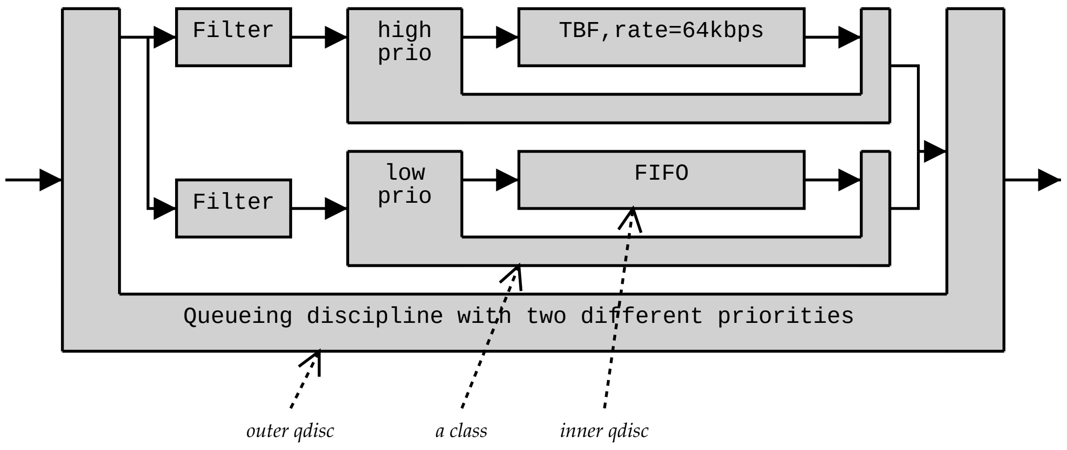


Figure 5.2: Combination of various queueing disciplines and a filter

5.2.2 Classes

Every queueing discpline has one ore more classes attached to it. The very existence of classes, and their semantics, are fundamental properties of a queueing discpline (qdisc). A qdisc uses classes to treat various classes of traffic in different ways. Note that this disctinction is made using filters. Classes are not storage places, they can use other queueing disciplines for that. So within a queuing discipline attached to a network device, other queuing disciplines may reside. And to these qdisc's, other filters and classes may be attached, hence giving enourmous flexibility (and complexity in configuration) to the user of the TC framework.

In figure 5.2 two classes of traffic are distinguished. The token-bucket-filter and the FIFO qdiscs are examples of inner queuing disciplines within the outer qdisc attached to the device. This leads to the conclusion that a class ″owns″exactly one queue, but in theory it is possible for classes to share a single queue. Though this adds a complexity factor: within a queue it is not clear which packets belong to which classes, as they are not carrying additional information with an explicit indication.

5.2.3 Filters

Filters are used by a queueing discpline to assign incoming packets to one of its classes, at enqueuing time. Filters are kept in filter lists that can be maintained either per qdisc or per traffic class, depending on the design of the queueing discipline, and are ordered by priority. The structure of filters is displayed in figure 5.3.

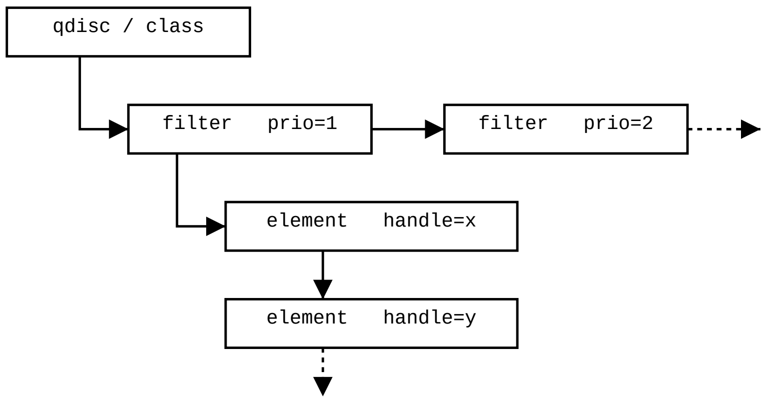


Figure 5.3: Combination of various queueing disciplines and a filter

As can be seen in the picture, filters may or may not have an internal structure. When they have, those internal elements are used to classify the traffic according to any field from the packet headers (e.g. from and destination IP addresses, or the DiffServ Code Point). It is possible to do multiple selections in a row, in order to start classifing traffic on one criterium and then on another.

5.2.4 Policing

To prevent network traffic from exceeding certain bounds, policing is used. In the context of Linux Network Traffic Control, policing affects all traffic control actions that depend in some way on the traffic volume. This includes but is not limited to decisions about whether to drop or to enqueue packets in both the inner and outer queueing disciplines.

Possible critera that are the parameters to this decision are maximum packet size, average rate of the traffic, the peak rate and the burstiness of the traffic. But it is certainly possible to extend this list with other properties, if an implementor would want that.

5.3 DiffServ in the Linux kernel

In figure 5.4 the relation of elements of the Differentiated Services architecture and the Linux Network Traffic Control elements is outlined. The three main functions (classification, metering and queueing/scheduling) are performed by different elements in the two architectures, as is highlighted by the grey boxes. An imminent conclusion is that the DiffServ architecture has not been designed specifically for Linux, nor has the Network Traffic Control code been tailored for DiffServ.

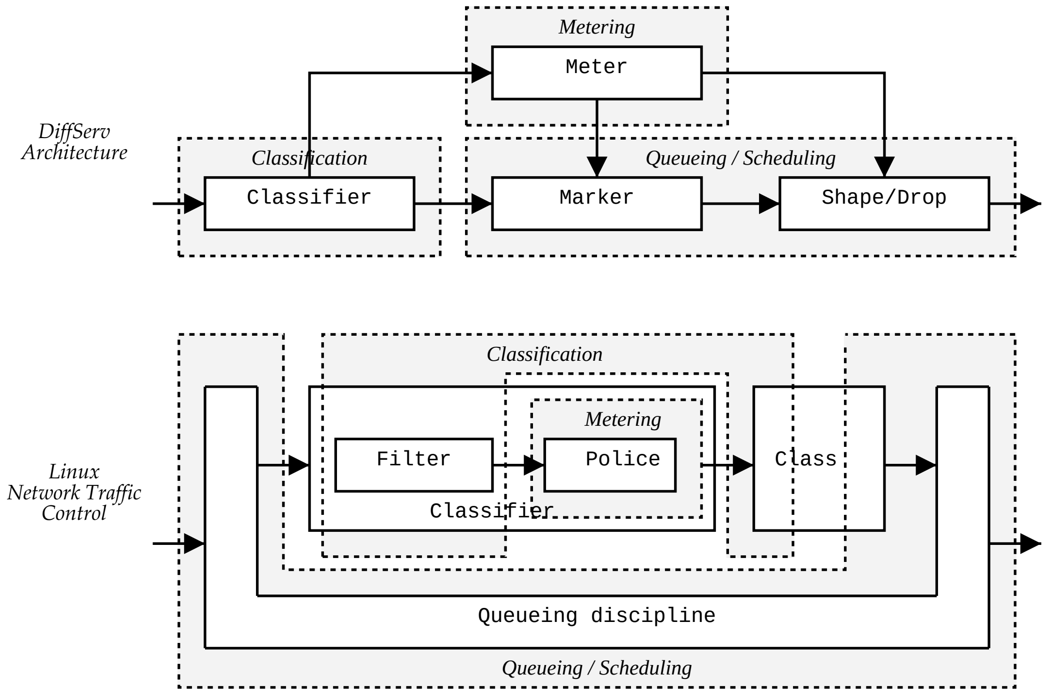


Figure 5.4: DiffServ versus TC architecture

The good news is that the TC framework nevertheless offers most of the functionality required for implementing DiffServ support. EPFL used this framework to extend the Linux kernel with DiffServ support. Three new elements have been added to the original implementation:

- To be able to support the Per-Hop Behaviours defined by the IETF diffserv WG (expedited and assured forwarding), a queueing discipline implementing the RED algorithm (in particular GRED) has been added (*sch_gred*)
- The use of the DiffServ Code Point, necessary for the scalable classification of packets throughout the network, leads to the introduction of yet another queueing discipline (*sch_dsmark*)
- A new classifier that uses this information is needed as well (*cls_tcindex*)

In the Linux kernel, traffic control is focused on the egress part of a router. This is in contrast with the DiffServ architecture, that heavily uses the ingress interface. In order to support this, it is necessary for routers running the Linux operating system to be able to distinguish packets using the inbound network interface. This is done using the `netfilter` infrastructure and its `iptables` commandline utility.

Figure 5.5 outlines the usage of the various new elements in order to implement Differentiated Services on a core router (not an edge router).

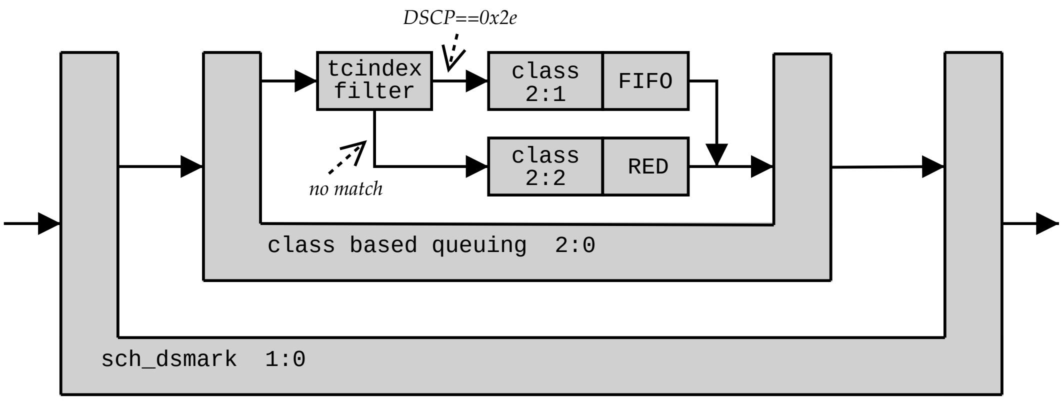


Figure 5.5: Expedited Forwarding in a core router using TC

A more detailed description of this implementation can be found in [4].

5.4 Configuring

Even without the use of the DiffServ MIB, it is possible to configure the traffic control functions in the Linux kernel. This is achieved by using the `tc` commandline utility. This tool uses the `netlink` interface for configuration as outlined in figure 5.6.

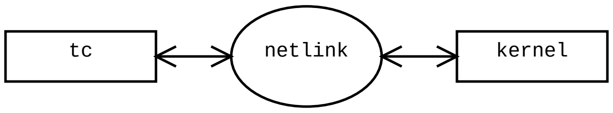


Figure 5.6: Communication between tc and the kernel

A full description of this tool is outside the scope of this document. The author is not aware of a reference guide, but an overview of the various options and commands in Portugese is available[22].

In order to test the prototype of the DiffServ MIB on Linux, the router has been setup using this `tc` utility. The script that was used can be found in appendix C, and sets up the router using a configuration like the Expedited Forwarding PHB from figure 5.5.

Chapter 6

The DiffServ MIB and the Linux Kernel

6.1 Introduction

A DiffServ implementation for the Linux kernel doesn't know about the DiffServ MIB specification. Therefore, it is necessary to map the various management functions offered by the MIB to functions provided by the DiffServ implementation. From figure 5.4 it is clear that the DiffServ and the Linux Network Traffic Control architectures are quite different. The management of both designs is tailored after the respective architectures: the MIB is modelled after the DiffServ Architecture, but the kernel is to be configured using handles that are "pointers" to the various elements by sending `rtnetlink` messages.

This leads to the conclusion that conversion needs to be done before the MIB can be filled with values from the kernel (like counters), and that configuration information that is written into the MIB by a manager application need to be translated to the correct `rtnetlink` messages.

It is clear that various protocols have to be combined. Figure 6.1 gives an overview of the communication between the respective entities in the model.

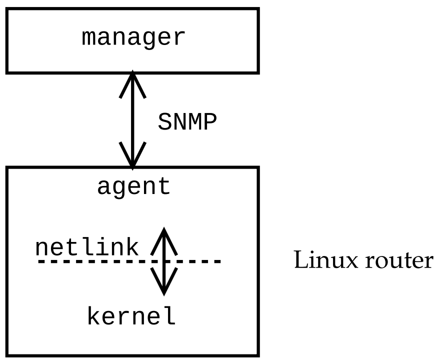


Figure 6.1: manager, agent and kernel: communication protocols

6.2 Representing information from the kernel

The prototyping in this assignment is focused on the "monitoring" part of the manager-agent paradigm, i.e. the manager retrieves some values from the agent. Therefore the implementation of the agent needs to gather information when it is requested to do so (or on its own). In the case of the DiffServ MIB agent, this means that the agent has to get information from the kernel using the `rtnetlink` messages. An extensive discussion of this type of messages can be found in appendix D.

This sections covers the various tables from the DiffServ MIB and explains what needs to be done in order to retrieve the requested information.

6.2.1 DataPath Table

This tables merely is a starting point for the DiffServ management information. The network interfaces are numbered according to the `ifTable` numbering, which happens to be the same as the internal numbering used in the Linux kernel. This information is retrieved using a `RTM_GETLINK` message. No conversion needs to be done at all, though parsing of the resulting message remains necessary of course.

6.2.2 Classifier Tables

In the MIB there is the "SixTuple Classifier" which makes it possible to represent any part of the IP and transport layer headers in the MIB, like IP addresses, DSCPs and port numbers.

Linux Network Traffic Control makes use of two types of filters that together give the information necessary to fill these tables in the MIB: the `u32` filter which looks at addressing information, and the `tcindex` filter which uses the TOS field from the IP header, thus giving information about the DiffServ Code Point. To retrieve information about these filters (i.e. their parameters but at least as important information about their place in the chain of elements that are part of the network traffic control engine), `RTM_TFILTER` messages are used.

The Classifier tables are indexed by two separate identifiers, enumerating the classifier entries and the classifier element entries. In Linux, there is no unique identification for a filter. Elements within a single filter are identified with a handle though. Multiple filters belonging to the same queueing discipline or class are ordered by a numerical priority value. Thus a DiffServ MIB implementation must take care of assigning unique identifiers to the filters and their elements itself.

6.2.3 Meter Tables

The Meter tables can be found in the `diffServMeter` and `diffServTBParam` subtrees in the MIB. From figure 5.4 it is clear that Metering is the same as Policing in the TC environment. Linux uses only three "primitives"when dealing with traffic control: `qdisc`, `filter` and `class`. None of them is exactly the same as a Meter in the DiffServ architecture.

However, the "Classifier"element in that picture, which is not a real element as it not formally defined in the TC architecture, is what the `filter` primitive can do in terms of functionality. A conclusion and solution to this problem can easily be drawn: the `RTM_FILTER` messages can be used to gain knowledge about policing in the kernel as well. This is something that typically only occurs at edge routers at the boundary of a diffserv domain; core routers use filters only for determining the DSCP value.

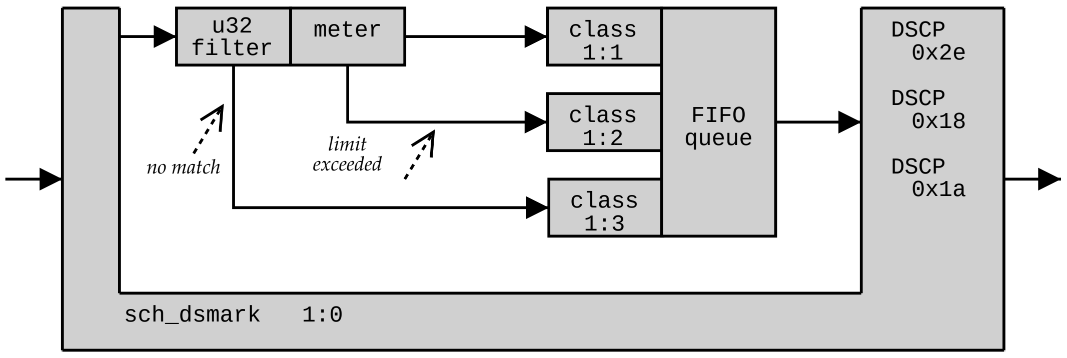


Figure 6.2: Classifying, Policing and Marking at the edge of a DiffServ domain

Figure 6.2 makes clear how this works, and also solves the open issue of the "succeed" and "fail" operations that follow a policing decision.

The problem with indexing of the tables, which is done with identifiers of the meters, is the same as with the Classifier tables. Hence the implementation of the DiffServ MIB must take care of this administrative work.

6.2.4 Action Tables

There are basically two operations that belong to these tables: marking with some DSCP value, and counting the packets.

From picture 5.4 it is known that the Marking operation in the DiffSer architecture is part of the Class element in the TC architecture. The corresponding primitive, `class`, provides an implementor with the necessary operations, using the `RTM_TCLASS` messages.

Counting is achieved in the TC architecture using a packet counting FIFO that can be used as an inner queueing discipline. This is controlled with the `qdisc` primitive. In the Linux kernel this special FIFO queue is enhanced with a limit on its size, but the DiffServ MIB does not support this for the Counting operation.

Indexing these tables with the right values is once again up to the agent, though the handles and classid's that are used internally by the Linux TC engine might be helpfull.

6.2.5 Queuing Tables

In the MIB queueing falls apart in three subcategories: algorithmic dropper management, and queueing and scheduling management. They are all part of both the `class` and `qdisc` primitives in the Linux TC architecture (figure 5.4). Class-Based Queueing (CBQ) is often used in Linux to get this functionality.

The TC implementation provides a so-called Weighted Round Robin (WRR) scheduling method as part of CBQ, and parameters (like traffic rate) to this algorithm are accessed using the `RTM_QDISC` messages. WRR can be represented in the MIB in the Scheduler table, using an `diffServAssuredRateEntry` to store the parameters. The `diffServShapingRate` table is not used, as there is no corresponding implementation in the Linux Network Traffic Control engine.

Administration like indexing values for these tables is up to the DiffServ MIB agent. Theoretically it is possible to share queues in TC, but this is very complicated and introduces other problems, e.g. when actions need to be performed on only a specific part of the queue. This implicates that assigning unique identifiers to the parameter sets suffices.

The conclusion that can be drawn from the previous paragraphs is that it proves to be impossible to come up with sensible values for each and every object in the MIB. Most tables can be filled however, so it is feasible to implement (at least the monitoring part of) the DiffServ MIB on a Linux router.

6.3 Setting configuration information in the kernel

No extensive research has been done in this area, but the problems discovered in the previous section apply here as well: there is no exact match between the Linux Network Traffic Control architecture and the DiffServ architecture, so an implementor has to come up with conversions.

It seems possible to configure TC using the DiffServ MIB, i.e. to implement the configuration part of the manager-agent paradigm, but this has not been done in this project. The usage of complex tables in the DiffServ MIB makes things very complicated, but not impossible.

Chapter 7

DiffServ MIB Implementation Overview

7.1 Introduction

In the previous chapters a description of the DiffServ protocol and management of DiffServ networks was given. These form the foundation of the implementation work, part of the assignment this report is about.

This chapter gives an overview of a prototype implementation of the DiffServ MIB, on a router running the Linux operating system. It starts with a description of the `net-snmp` SNMP implementation, from an implementors point of view. Because of the amount of "reusable" code in SNMP agents, program code is often automatically generated using code-generators. In this project, `libsmi` was used. The report continues with a (global) description of the protocol that is used to retrieve information that is needed in the MIB from the kernel: `rtnetlink`. Finally a small example, part of the DiffServ MIB implementation, is discussed.

7.2 Support routines: net-snmp

`Net-snmp` is a suite that implements the Simple Network Management Protocol and comes with an extensible (master-) agent, a library with SNMP related support routines and various tools to perform `Get` and `Set` operations on information stored in a MIB. This software package may be downloaded from [19]. It can be run on various operating systems, such as Solaris and Linux, while it supports SNMPv1, SNMPv2c and SNMPv3. It also has support for the AgentX (Agent Extensibility) protocol, but at time of writing this is not yet really stable and complete, unfortunately.

Apart from the SNMP Agent, that comes with support for numerous MIBs in the `mib-2`, the `net-snmp` suite contains a number of (shared) libraries that independent applications may use to act as an SNMP agent. The library provides functions for almost all general SNMP related routines, like receiving and sending SNMP PDUs.

One can imagine that it is not really efficient to have one piece of code implementing every MIB; a modular approach, with different modules implementing different parts of the MIB-tree, is far more attractive. There are various possibilities for extending the `net-snmp` agent with support for other MIBs, e.g. the DiffServ MIB.

- *Agent Extensibility Protocol* (AgentX)[8]

  This model uses real master- and sub-agents, communicating over a channel like TCP/IP or UNIX Domain Sockets. The master-agent, i.e. `net-snmp`, dispatches requests that should be handled by a sub-agent to that particular sub-agent over that channel. The sub-agent returns whatever needs to be returned, and the master-agent sends it back to the manager entity.

  The problem with AgentX in `net-snmp` is that the support is not complete yet. This is especially the case when dealing with complex tables in MIBs, like the DiffServ MIB has. In a future version of the suite, AgentX will likely be fully supported as it is regarded as the standard agent extensibility protocol now.

- *SNMP Multiplexing Protocol* (SMUX)[23]

  This protocol allows a user-process, termed a SMUX peer, to register itself with the running SNMP agent when it wants to export a MIB module. This protocol is the predecessor of AgentX, and is regarded as historic by the IETF.

- *dlmod* (`net-snmp` proprietary interface)

  With `net-snmp` it is possible to put directives in the configuration file of the SNMP agent that make it load "shared object files", i.e. compiled sub-agents that implement MIBs outside of the `net-snmp` code base. These sub-agents use the `net-snmp` shared libraries for the "non-instrumentation" part of their code. They get loaded by the master-agent at startup. The idea of the master-agent dispatching requests for particular parts of the MIB-tree to those sub-agents remains intact.

In this project, this *dlmod* method has been used. AgentX is promising, and it is recommended for future applications to use it as it is an open and standard protocol, but it cannot be used in this project because of stability and support problems, especially when dealing with complex tables (like the tables in de DiffServ MIB).

To summerize what happens when running the `net-snmp` suite with a DiffServ MIB implementation in pseudo-code (see also figure 2.7 and 6.1):

```
start and initialize master agent
load DiffServ MIB sub agent
while true
   wait for request from manager
   if request is for DiffServ subtree
      dispatch to DiffServ MIB sub agent
         (in sub agent)
            call function that handles this OID
            create netlink message if necessary
            send and receive netlink messages
            parse netlink message
            send result back to master agent
         (end of sub agent processing)
   else /* request is for another sub agent */
      ...
   endif
   construct SNMP response PDU, send to manager
endwhile
```

7.3 Code-generation: libsmi

Libsmi[20] is a library to access SMI MIB information. Applications can use it to access SMI information that is stored in various repositories (plain text, but the model allows for network access as well) containing SMIv1 and SMIv2 as well es SMIng MIB module files. Its purpose is to separate the parsing and handling of these MIB module files from the manager- or agent-application itself.

On top of this library, various tools are available, e.g. a MIB syntax checker and a tool to dump and convert SMI files in various formats. One of these formats is C program code. This gives the implementor a framework that needs to be filled in, containing most of the reusable code that is present in almost every SNMP agent. This "driver" was used in this DiffServ MIB prototype to generate the C program code. An (partial) example of this output can be found in section 7.5.

Other drivers provide for example a conversion from SMI to Java code to use with the JAX AgentX program. The decision was made in this assignment to use C code instead of Java because the environment the agent has to run in: a Linux machine. Linux is quite C-oriented, and especially the netlink part for communication with the kernel is difficult to implement using Java (in fact it's not possible at all without the use of the Java Native Interface). Also the big advantage of Java, portability between various operating systems and platforms, is not an issue here as other platforms have very different interfaces to their DiffServ implementation. A DiffServ MIB agent developed for the Linux operating system will most likely not run on other platforms.

Libsmi is also capable of dumping MIB module files in a tree format (like the DiffServ MIB in appendix B) and reverse engineering MIBs to UML diagrams (see section 4.3 for some figures that are made with help of libsmi).

7.4 RTNetlink Protocol

Linux has support for a lot of advanced networking features, including policy based routing and Quality of Service. These features are configured and controlled using a `netlink` socket interface. Routing messages, called `rtnetlink`, are special netlink messages to control the routing behaviour of Linux, which includes Network Traffic Control and DiffServ functionality.

Netlink is used to transfer information between kernel and user-space processes, over its bidirectional communications links. It consists of a standard socket-based interface for user-processes and an internal application program interface (API) for the kernel. Netlink sockets are "raw", but it is a datagram oriented service.

This means that a user can get information from the kernel, as outlined in the following piece of pseudo-code:

```
   (user process)
1  open netlink socket
2  construct request message
3  send message over netlink socket
      (kernel)
4     receive and parse message
5     find necessary information in internal data
6     construct response message
7     send message over netlink socket
   (user process)
8  wait for messages on the open netlink socket
9  receive and parse message
   /* do whatever the user space program needs to do */
```

Such code is typically part of an application that needs to provide information about routing or traffic control information. In this assignment, support for netlink sockets is part of the DiffServ MIB subagent.

Appendix D contains a more detailed description of the `rtnetlink` protocol.

7.5 Implementation Example

In section 7.2 an outline of the SNMP agent process has been given, and in the previous section the netlink communication, part of the agent, has been described. This section concludes this chapter with an example in (pseudo-)code. It is taken from the code that implements the `diffServClfrElementTable` table.

Upon receipt of an SNMP `GetNext` message, while in the DiffServ MIB subtree, the master agent sends the request to a function that has earlier being registered with the master agent as taking care of the subtree of the DiffServ MIB that contains this table.

```
1  master agent receives request, and finds out which function
   to call: read_diffServClfrElementEntry_stub(...)
```

The arguments to this function include the requested OID, which has a prefix that corresponds with the OIDs in the `diffServClfrElementTable` table. When this function gets called, it starts with a sanity check on the requested object:

```
2  if (header_diffServClfrElementEntry(...))
     return NULL;
```

This function checks whether the requested OID is a possible instance in the table, and whether te instance exists. This has to be implemented manually and contains some complicated logic regarding the indices of the table, especially in the case of complex tables. If the requested index is not found, and there is no possible instance in this table that comes after the requested row, NULL is returned. This indicates to the master agent that it should continue with the next subtree, if any.

The sub agent then continues with a procedure to copy the information regarding the requested object into a given piece of memory:

```
3  read_diffServClfrElementEntry (&diffServClfrElementEntry, ...);
```

This procedure contains the real implementation: it has to find out which value from the kernel (or another data source) it has to provide to the master agent, that subsequently constructs the responding SNMP message. There are various possibilities here: either renew the already known information from the kernel in real time, or provide the master agent with possible outdated information. This choice is a trade-off between doing the complicated message interaction and all the administrative burden related with it, or simply copy the data from a data structure that is already present in the agent. For some objects, like counters, the information should be as uptodate as possible. But configuration information is not likely to change very often, so in such cases the simple copy-operation is probably good enough.

When the requested information is somehow put into the piece of memory pointed to by `&diffServClfrElementEntry`, the agent returns it to the master agent. For ease of use, all types of data are returned as "char-pointers". This is done one row at a time:

```
4  switch(vp->magic) {
     ...
     case DIFFSERVCLFRELEMENTSPECIFIC:
       ...
       return (unsigned char *) diffServClfrElementEntry.
                         diffServClfrElementSpecific;
     ...
   }
```

In this piece of code, `vp->magic` is the number of the requested row, in this case the `diffServClfrElementSpecific` row.

The master agent is now able to construct an SNMP message to send to the manager. The manager application receives a variable binding containing the requested OID and a RowPointer (the value) containing the OID of a Multi-Field Classifier, for instance.

The prototype containing all the details is available for download (see section 1.3).

Chapter 8

Conclusions and Recommendations

The research and programming efforts that have been put into this assignment have resulted in a number of conclusions and recommendations for possible future work.

8.1 Conclusions

In section 1.2 two goals for this assignment have been defined. Answers to those questions have been found by prototyping the DiffServ MIB on a router running Linux.

The first question:

> Does the DiffServ MIB, with this prototype implementation, solve the management issues it is intended to address?

The DiffServ MIB is supposed to help network service providers operating their network, managing the DiffServ functionality in routers. The MIB is closely modeled after the Differentiated Services Architecture, as can be concluded from the description and analyses in chapter 3 and 4. An operator can manage every aspect of DiffServ, when the DiffServ functionality in the router is modelled after the Architecture defined by the IETF diffserv WG.

However in systems where this is not the case the DiffServ functionality may or may not be fully manageble using the DiffServ MIB. The MIB solves this issue by providing functionality to extend the MIB in such cases, by means of using RowPointers to other MIBs. Vendors can provide MIBs that comply with their own implementation of DiffServ. These can be linked to the DiffServ MIB, so that a network manager is able to fully utilise the router's capabilities.

The prototyping environment in this assignment is DiffServ on a router running the Linux operating system. Linux comes with a generic framework for Network Traffic Control, and DiffServ support is implemented on top of that framework. The underlying architecture is different from the DiffServ Architecture, and it turns out that the problem that is outlined above is present. Most of the router's functionality, with regard to DiffServ, is managable with the DiffServ MIB, and it is possible to fill most of the DiffServ MIB with sensible values retrieved from the DiffServ implementation in the Linux kernel. But to achieve this result, it is necessary to map functionality from the DiffServ Architecture to the elements of the Linux Network Traffic Control (see figure 5.4), which is non-trivial.

The second issue addressed in this assignment:

> Instrumenting the MIB might cause discovery of some "problems" with the current draft version, like under- or over-specification. The IETF DiffServ Working Group will be interested, so giving them feedback about the results of this work is important.

The DiffServ MIB seems well equipped: it is perfectly possible to manage every building block mentioned in the Architecture, and it is extendible for most situations when those building blocks do not apply to the managed router. This has been reported to the IETF community via email, as listed in appendix E. During the prototyping process, the author has closely followed the IETF diffserv WG mailinglist, as well as other DiffServ-related lists such as the Linux specific DiffServ list. Suggestions and questions about the DiffServ MIB have been sent to various people in private email a number of times.

The MIB is still under development, but the IETF diffserv WG is working towards a final version. Major changes are not likely to happen anymore, and the current version seems fine. New revisions will probably focus on minor things like exact parameterization of properties.

The prototype that has been developed during this assignment only implements the monitoring of DiffServ on a Linux router. Configuration aspects have been looked at, but the time available for this assignment was not sufficient to implement it. It seems feasible to implement DiffServ configuration using this MIB on the Linux platform, however. Possible future work may lead to a full implementation of the DiffServ MIB, which is necessary to get the MIB promoted along the IETF Standards Track, and to act in a Policy-Based Management framework together with the DiffServ Policy MIB.

8.2 Recommendations

A clear recommendation for future work is implementing the configuration part of the DiffServ MIB.This is not an easy task, and given the complexity and learning curve associated with it, the author doesn't recommend it as an M.Sc assignment for one single person.

The differences between the DiffServ Architecture and the Linux Network Traffic Control are very fundemental. It is important to note that (full) DiffServ functionality can be achieved without conformation to the IETF diffserv WG's architectural document, but the DiffServ MIB might be less suited in such cases. TC is only a relatively small part of Linux' network code. The Classification functionality of DiffServ is implemented in an entirely different part of the kernel: `netfilter`. Netfilter gives a functional framework for filtering, network address translation and packet mangling functions. Marking packets with a particular DSCP value on the ingress side of a router is much easier to do with netfilter then it is with TC. But the disavantage is that netfilter can't be configured using the netlink interface to the kernel. It is probably possible to create a MIB that configures netfilter; the DiffServ MIB may point to that MIB as the parameterization of classifier elements.

The author doesn't think that either the DiffServ MIB or the TC architecture should be changed to be "more compatible". They are conceptually different, and there is little reason to change that. It is true that it might not be possible to fully manage the DiffServ functionality in the kernel using the MIB, but most of it is. MIBs are instruments to operators, but they should not dictate the underlying infrastructure: it wouldn't be wise to drop part of the capabilities of Linux Network Traffic Control to fully comply with the DiffServ Architecture. Also, the MIB has to be generic: it has to "run" on multiple architectures. It is very difficult to find a "common denominator" between all those architecture without loosing too much of the MIB's capabilities. The recommendation in this respect hence is not to change the MIB or TC architecture.

Bibliography

[1] Werner Almesbergers. Linux network traffic control: Implementation overview. April 1999. ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.

[2] Baker, Chan, and Smith. Management information base for the differentiated services architecture.
http://www.ietf.org/internet-drafts/draft-ietf-diffserv-mib-10.txt, June 2001 (work in progress).

[3] IETF diffserv WG. Differentiated services working group charter page.
http://www.ietf.org/html.charters/diffserv-charter.html.

[4] Almesberger et al. Differentiated services on linux.
ftp://icaftp.epfl.ch/pub/linux/diffserv/misc/dsid-01.ps.gz, June 1999.

[5] Bernet et. al. Diffserv informal management model.
http://www.ietf.org/internet-drafts/
draft-ietf-diffserv-model-06.txt, February 2001 (work in progress).

[6] Blake et al. Architecture for differentiated services. (RFC 2475), December 1998.
http://www.ietf.org/rfc/rfc2475.txt.

[7] Crawley et al. A framework for qos-based routing in the internet. (RFC 2386), August 1998.
http://www.ietf.org/rfc/rfc2386.txt.

[8] Daniele et al. Agent extensibility (agentx) protocol version 1. (RFC 2741), January 2000.
http://www.ietf.org/rfc/rfc2741.txt.

[9] Heinanen et al. Assured forwarding phb group. (RFC 2597), June 1999.
http://www.ietf.org/rfc/rfc2597.txt.

[10] Jacobson et al. An expedited forwarding phb. (RFC 2598), June 1999.
http://www.ietf.org/rfc/rfc2598.txt.

[11] McCloghrie et al. Structure of management information version 2. (RFC 2578), April 1999.
http://www.ietf.org/rfc/rfc2578.txt.

[12] N. Semret et al. Peering and provisioning of differentiated internet services. *Proceedings of the IEEE INFOCOM'2000*, March 2000.

[13] Ethereal. Ethereal network analyzer.
http://www.ethereal.com/.

[14] S. Floyd and V. Jacobson. Random early detection gateways for congestion avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[15] Hazewinkel and Partain. The diffserv policy mib.
`http://www.ietf.org/internet-drafts/`
`draft-ietf-snmpconf-diffpolicy-04.txt`, March 2001 (work in progress).

[16] IETF intserv WG. Integrated services working group charter page.
`http://www.ietf.org/html.charters/intserv-charter.html`.

[17] Alexey Kuznetsov. Network traffic control configuration tools.
`ftp://ftp.inr.ac.ru/ip-routing/`.

[18] IETF mpls WG. Multiprotocol label switching working group charter page.
`http://www.ietf.org/html.charters/mpls-charter.html`.

[19] NET-SNMP. Project home page.
`http://www.net-snmp.org/`.

[20] Instute of Operating Systems and Computer Networks TU Braunschweig. libsmi home page.
`http://www.ibr.cs.tu-bs.de/projects/libsmi/`.

[21] DPNM Lab Postech. Diffserv webpage.
`http://dpnm.postech.ac.kr/research/01/ipqos/dsmib/index.html`.

[22] Rui Prior. Qualidade de servico. 2001.
`http://telecom.inescn.pt/doc/msc/rprior2001.pdf`.

[23] M. Rose. Snmp mux protocol and mib. (RFC 1227), May 1991.
`http://www.ietf.org/rfc/rfc1227.txt`.

[24] Oscar Sanz. *Feasibility study of implementing the DiffServ MIB*. University of Twente, 2000.

[25] Jürgen Schönwälder. Internet management standards: Quo vadis? February 1999.
`http://www.ibr.cs.tu-bs.de/ schoenw/papers/mvs-99.ps.gz`.

[26] IETF snmpconf WG. Configuration management with snmp working group charter page.
`http://www.ietf.org/html.charters/snmpconf-charter.html`.

[27] William Stallings. *SNMP, SNMPv2, SNMPv3, and RMON 1 and 2*. Addison-Wesley, third edition, 1999.

[28] W. Richard Stevens. *TCP/IP Illustrated, Volume1: The Protocols*. Addison-Wesley, 1994.

[29] UTWENTE/TSS-MGT and TUBS/IBR. Simpleweb homepage.
`http://www.simpleweb.org/`.

List of Acronyms

AF	Assured Forwarding
ASN.1	Abstract Syntax Notation One
BGP	Border Gateway Protocol
CBQ	Class Based Queueing
DiffServ,DS	Differentiated Services
DSCP	DiffServ Code Point
EF	Expedited Forwarding
FIFO	First In First Out
IETF	Internet Engineering Task Force
IP	Internet Protocol
ISP	Internet Service Provider
MIB	Management Information Base
MPLS	Multiprotocol Label Switching
PDU	Protocol Data Unit
RED	Random Early Detection
RFC	Request For Comments
RSVP	Resource reSerVation Protocol
SLA	Service Level Agreement
SMI	Structure of Management Information
SMUX	SNMP MUltipleXing protocol
SNMP	Simple Network Management Protocol
TC	Linux Network Traffic Control architecture
TCB	Traffic Control Block
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UML	Unified Modelling Language

Appendix A

SMIv2 Example

An example of Management Information in SMIv2 notation: a counter that keeps track of the number of octets sent through some network interface.

```
ifOutOctets OBJECT-TYPE
    SYNTAX      Counter32
    MAX-ACCESS  read-only
    STATUS      current
    DESCRIPTION
            "The total number of octets transmitted out of
            the interface, including framing characters.

            Discontinuities in the value of this counter can
            occur at re-initialization of the management
            system, and at other times as indicated by the
            value of ifCounterDiscontinuityTime."
    ::= { ifEntry 16 }
```

Appendix B

Differentiated Services MIB tree

In this appendix the current (as of July 2001) DiffServ MIB[2] is displayed in a tree format. It has been generated using the LIBSMI toolkit with the module definitions from the Internet Draft:

```
smidump -f tree DIFFSERV-MIB
```

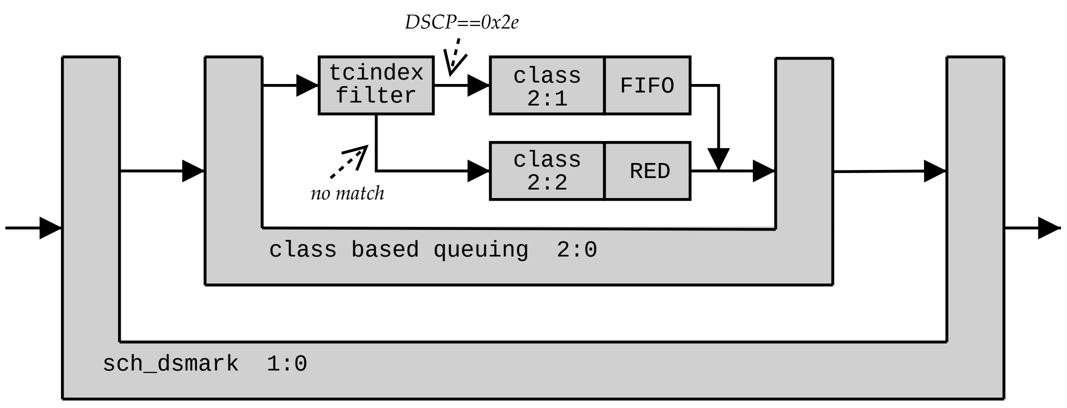
```
diffServMib(1.3.6.1.2.1.12345)
|
+--diffServMIBObjects(1)
|  |
|  +--diffServDataPath(1)
|  |  |
|  |  +--diffServDataPathTable(1)
|  |     |
|  |     +--diffServDataPathEntry(1) [ifIndex,diffServDataPathIfDirection]
|  |        |
|  |        +-- --- IfDirection diffServDataPathIfDirection(1)
|  |        +-- rwn RowPointer  diffServDataPathStart(2)
|  |        +-- rwn RowStatus   diffServDataPathStatus(3)
|  |
|  +--diffServClassifier(2)
|  |  |
|  |  +-- r-n Integer32 diffServClfrNextFree(1)
|  |  |
|  |  +--diffServClfrTable(2)
|  |  |  |
|  |  |  +--diffServClfrEntry(1) [diffServClfrId]
|  |  |     |
|  |  |     +-- --- Unsigned32 diffServClfrId(1)
|  |  |     +-- rwn RowPointer diffServClfrDataPathStart(2)
|  |  |     +-- rwn RowStatus  diffServClfrStatus(3)
|  |  |
|  |  +-- r-n Integer32 diffServClfrElementNextFree(3)
|  |  |
|  |  +--diffServClfrElementTable(4)
|  |  |  |
|  |  |  +--diffServClfrElementEntry(1) [diffServClfrElementClfrId,
|  |  |                                   diffServClfrElementId]
|  |  |     |
|  |  |     +-- --- Integer32  diffServClfrElementClfrId(1)
|  |  |     +-- --- Integer32  diffServClfrElementId(2)
|  |  |     +-- rwn Unsigned32 diffServClfrElementPrecedence(3)
|  |  |     +-- rwn RowPointer diffServClfrElementNext(4)
|  |  |     +-- rwn RowPointer diffServClfrElementSpecific(5)
|  |  |     +-- rwn RowStatus  diffServClfrElementStatus(6)
|  |  |
|  |  +-- r-n Integer32 diffServSixTupleClfrNextFree(5)
|  |  |
|  |  +--diffServSixTupleClfrTable(6)
|  |     |
|  |     +--diffServSixTupleClfrEntry(1) [diffServSixTupleClfrId]
|  |        |
|  |        +-- --- Integer32               diffServSixTupleClfrId(1)
|  |        +-- rwn InetAddressType         diffServSixTupleClfrDstAddrType(2)
|  |        +-- rwn InetAddress             diffServSixTupleClfrDstAddr(3)
|  |        +-- rwn InetAddressPrefixLength diffServSixTupleClfrDstPrefixLength(4)
|  |        +-- rwn InetAddressType         diffServSixTupleClfrSrcAddrType(5)
|  |        +-- rwn InetAddress             diffServSixTupleClfrSrcAddr(6)
|  |        +-- rwn InetAddressPrefixLength diffServSixTupleClfrSrcPrefixLength(7)
|  |        +-- rwn DscpOrAny               diffServSixTupleClfrDscp(8)
|  |        +-- rwn Unsigned32              diffServSixTupleClfrProtocol(9)
|  |        +-- rwn InetPortNumber          diffServSixTupleClfrDstL4PortMin(10)
|  |        +-- rwn InetPortNumber          diffServSixTupleClfrDstL4PortMax(11)
|  |        +-- rwn InetPortNumber          diffServSixTupleClfrSrcL4PortMin(12)
|  |        +-- rwn InetPortNumber          diffServSixTupleClfrSrcL4PortMax(13)
|  |        +-- rwn RowStatus               diffServSixTupleClfrStatus(14)
|  |
|  +--diffServMeter(3)
|  |  |
|  |  +-- r-n Integer32 diffServMeterNextFree(1)
|  |  |
|  |  +--diffServMeterTable(2)
|  |     |
|  |     +--diffServMeterEntry(1) [diffServMeterId]
|  |        |
|  |        +-- --- Integer32  diffServMeterId(1)
|  |        +-- rwn RowPointer diffServMeterSucceedNext(2)
|  |        +-- rwn RowPointer diffServMeterFailNext(3)
|  |        +-- rwn RowPointer diffServMeterSpecific(4)
|  |        +-- rwn RowStatus  diffServMeterStatus(5)
|  |
|  +--diffServTBParam(4)
|  |  |
|  |  +-- r-n Integer32 diffServTBParamNextFree(1)
|  |  |
|  |  +--diffServTBParamTable(2)
|  |  |  |
|  |  |  +--diffServTBParamEntry(1) [diffServTBParamId]
|  |  |     |
|  |  |     +-- --- Integer32        diffServTBParamId(1)
|  |  |     +-- rwn ObjectIdentifier diffServTBParamType(2)
|  |  |     +-- rwn Unsigned32       diffServTBParamRate(3)
|  |  |     +-- rwn BurstSize        diffServTBParamBurstSize(4)
|  |  |     +-- rwn Unsigned32       diffServTBParamInterval(5)
|  |  |     +-- rwn RowStatus        diffServTBParamStatus(6)
|  |  |
|  |  +--diffServTBParamSimpleTokenBucket(3)
|  |  |
|  |  +--diffServTBParamAvgRate(4)
|  |  |
|  |  +--diffServTBParamSrTCMBlind(5)
|  |  |
|  |  +--diffServTBParamSrTCMAware(6)
|  |  |
|  |  +--diffServTBParamTrTCMBlind(7)
|  |  |
|  |  +--diffServTBParamTrTCMAware(8)
|  |  |
|  |  +--diffServTBParamTswTCM(9)
|   |
|   +--diffServAction(5)
|   |   |
|   |   +-- r-n Integer32 diffServActionNextFree(1)
|   |   |
|   |   +--diffServActionTable(2)
|   |   |   |
|   |   |   +--diffServActionEntry(1) [diffServActionId]
|   |   |       |
|   |   |       +-- --- Integer32  diffServActionId(1)
|   |   |       +-- rwn RowPointer diffServActionNext(2)
|   |   |       +-- rwn RowPointer diffServActionSpecific(3)
|   |   |       +-- rwn RowStatus  diffServActionStatus(4)
|   |   |
|   |   +--diffServDscpMarkActTable(3)
|   |   |   |
|   |   |   +--diffServDscpMarkActEntry(1) [diffServDscpMarkActDscp]
|   |   |       |
|   |   |       +-- r-n Dscp      diffServDscpMarkActDscp(1)
|   |   |
|   |   +-- r-n Integer32 diffServCountActNextFree(4)
|   |   |
|   |   +--diffServCountActTable(5)
|   |       |
|   |       +--diffServCountActEntry(1) [diffServCountActId]
|   |           |
|   |           +-- --- Integer32 diffServCountActId(1)
|   |           +-- r-n Counter32 diffServCountActOctets(2)
|   |           +-- r-n Counter64 diffServCountActHCOctets(3)
|   |           +-- r-n Counter32 diffServCountActPkts(4)
|   |           +-- r-n Counter64 diffServCountActHCPkts(5)
|   |           +-- r-n TimeStamp diffServCountActDiscontTime(6)
|   |           +-- rwn RowStatus diffServCountActStatus(7)
|   |
|   +--diffServAlgDrop(6)
|   |   |
|   |   +-- r-n Integer32 diffServAlgDropNextFree(1)
|   |   |
|   |   +--diffServAlgDropTable(2)
|   |   |   |
|   |   |   +--diffServAlgDropEntry(1) [diffServAlgDropId]
|   |   |       |
|   |   |       +-- --- Integer32   diffServAlgDropId(1)
|   |   |       +-- rwn Enumeration diffServAlgDropType(2)
|   |   |       +-- rwn RowPointer  diffServAlgDropNext(3)
|   |   |       +-- rwn RowPointer  diffServAlgDropQMeasure(4)
|   |   |       +-- rwn Unsigned32  diffServAlgDropQThreshold(5)
|   |   |       +-- rwn RowPointer  diffServAlgDropSpecific(6)
|   |   |       +-- r-n Counter32   diffServAlgDropOctets(7)
|   |   |       +-- r-n Counter64   diffServAlgDropHCOctets(8)
|   |   |       +-- r-n Counter32   diffServAlgDropPkts(9)
|   |   |       +-- r-n Counter64   diffServAlgDropHCPkts(10)
|   |   |       +-- r-n TimeStamp   diffServAlgDropDiscontinuityTime(11)
|   |   |       +-- rwn RowStatus   diffServAlgDropStatus(12)
|   |   |
|   |   +-- r-n Integer32 diffServRandomDropNextFree(3)
|   |   |
|   |   +--diffServRandomDropTable(4)
|   |       |
|   |       +--diffServRandomDropEntry(1) [diffServRandomDropId]
|   |           |
|   |           +-- --- Integer32  diffServRandomDropId(1)
|   |           +-- rwn Unsigned32 diffServRandomDropMinThreshBytes(2)
|   |           +-- rwn Unsigned32 diffServRandomDropMinThreshPkts(3)
|  |        +-- rwn Unsigned32 diffServRandomDropMaxThreshBytes(4)
|  |        +-- rwn Unsigned32 diffServRandomDropMaxThreshPkts(5)
|  |        +-- rwn Integer32  diffServRandomDropProbMax(6)
|  |        +-- rwn Integer32  diffServRandomDropWeight(7)
|  |        +-- rwn Integer32  diffServRandomDropSamplingRate(8)
|  |        +-- rwn RowStatus  diffServRandomDropStatus(9)
|  |
|  +--diffServQueue(7)
|  |  |
|  |  +-- r-n Integer32 diffServQNextFree(1)
|  |  |
|  |  +--diffServQTable(2)
|  |     |
|  |     +--diffServQEntry(1) [diffServQId]
|  |        |
|  |        +-- --- Integer32  diffServQId(1)
|  |        +-- rwn RowPointer diffServQNext(2)
|  |        +-- rwn RowPointer diffServQRate(3)
|  |        +-- rwn RowPointer diffServQShaper(4)
|  |        +-- rwn RowStatus  diffServQStatus(5)
|  |
|  +--diffServScheduler(8)
|     |
|     +-- r-n Integer32 diffServSchedulerNextFree(1)
|     |
|     +--diffServSchedulerTable(2)
|     |  |
|     |  +--diffServSchedulerEntry(1) [diffServSchedulerId]
|     |     |
|     |     +-- --- Integer32        diffServSchedulerId(1)
|     |     +-- rwn RowPointer       diffServSchedulerNext(2)
|     |     +-- rwn ObjectIdentifier diffServSchedulerMethod(3)
|     |     +-- rwn RowPointer       diffServSchedulerRate(4)
|     |     +-- rwn RowPointer       diffServSchedulerShaper(5)
|     |     +-- rwn RowStatus        diffServSchedulerStatus(6)
|     |
|     +--diffServSchedulerPriority(3)
|     |
|     +--diffServSchedulerWRR(4)
|     |
|     +--diffServSchedulerWFQ(5)
|     |
|     +-- r-n Integer32 diffServAssuredRateNextFree(6)
|     |
|     +--diffServAssuredRateTable(7)
|     |  |
|     |  +--diffServAssuredRateEntry(1) [diffServAssuredRateId]
|     |     |
|     |     +-- --- Integer32  diffServAssuredRateId(1)
|     |     +-- rwn Unsigned32 diffServAssuredRatePriority(2)
|     |     +-- rwn Unsigned32 diffServAssuredRateAbs(3)
|     |     +-- rwn Unsigned32 diffServAssuredRateRel(4)
|     |     +-- rwn RowStatus  diffServAssuredRateStatus(5)
|     |
|     +-- r-n Integer32 diffServShapingRateNextFree(8)
|     |
|     +--diffServShapingRateTable(9)
|        |
|        +--diffServShapingRateEntry(1) [diffServShapingRateId,diffServShapingRateLevel]
|           |
|           +-- --- Integer32  diffServShapingRateId(1)
|           +-- --- Integer32  diffServShapingRateLevel(2)
|           +-- rwn Unsigned32 diffServShapingRateAbs(3)
|           +-- rwn Unsigned32 diffServShapingRateRel(4)
|           +-- rwn BurstSize  diffServShapingRateThreshold(5)
|           +-- rwn RowStatus  diffServShapingRateStatus(6)
|
+--diffServMIBConformance(2)
   |
   +--diffServMIBCompliances(1)
   |  |
   |  +--diffServMIBCompliance(1)
   |
   +--diffServMIBGroups(2)
      |
      +--diffServMIBDataPathGroup(1)
      |
      +--diffServMIBClfrGroup(2)
      |
      +--diffServMIBClfrElementGroup(3)
      |
      +--diffServMIBSixTupleClfrGroup(4)
      |
      +--diffServMIBMeterGroup(5)
      |
      +--diffServMIBTBParamGroup(6)
      |
      +--diffServMIBActionGroup(7)
      |
      +--diffServMIBDscpMarkActGroup(8)
      |
      +--diffServMIBCounterGroup(9)
      |
      +--diffServMIBHCCounterGroup(10)
      |
      +--diffServMIBVHCCounterGroup(11)
      |
      +--diffServMIBAlgDropGroup(12)
      |
      +--diffServMIBRandomDropGroup(13)
      |
      +--diffServMIBQGroup(14)
      |
      +--diffServMIBSchedulerGroup(15)
      |
      +--diffServMIBAssuredRateGroup(16)
      |
      +--diffServMIBShapingRateGroup(17)
      |
      +--diffServMIBStaticGroup(18)
```

Appendix C

Configuration of a DiffServ router

This appendix lists the DiffServ configuration that is used in this project. It is a typical configuration for a "core router" implementing a expedited forwarding per-hop behaviour. It includes the output of the DiffServ MIB for this configuration.

C.1 Logical view of the configuration



C.2 Shell script used for configuration

```
#!/bin/sh

TC=/path/to/tc

$TC qdisc add dev dummy0 handle 1:0 root dsmark indices 64 \
        set_tc_index

$TC filter add dev dummy0 parent 1:0 protocol ip prio 1 \
        tcindex mask 0xfc shift 2

$TC qdisc add dev dummy0 parent 1:0 handle 2:0 cbq bandwidth \
        10Mbit allot 1514 cell 8 avpkt 1000 mpu 64

$TC class add dev dummy0 parent 2:0 classid 2:1 cbq bandwidth \
        10Mbit rate 1500Kbit avpkt 1000 prio 1 bounded \
        isolated allot 1514 weight 1 maxburst 10 defmap 1

$TC qdisc add dev dummy0 parent 2:1 pfifo limit 5

$TC filter add dev dummy0 parent 2:0 protocol ip prio 1 \
        handle 0x2e tcindex classid 2:1 pass_on

$TC class add dev dummy0 parent 2:0 classid 2:2 cbq bandwidth 10Mbit \
        rate 5Mbit avpkt 1000 prio 7 allot 1514 weight 1 maxburst 21 \
        borrow

$TC qdisc add dev dummy0 parent 2:2 red limit 60KB min 15KB max 56KB \
        burst 20 avpkt 1000 bandwidth 10Mbit probability 0.4

$TC filter add dev dummy0 parent 2:0 protocol ip prio 2 handle 0 \
        tcindex mask 0 classid 2:2 pass_on
```

C.3 DiffServ MIB: snmpwalk output

The output of a `snmpwalk` command, on a Linux router with DiffServ functionality, running the agent with the DiffServ MIB module loaded (edited for readability):

```
diffServDataPathStart[1][inbound] = OID: zeroDotZero.0
diffServDataPathStart[1][outbound] = OID: zeroDotZero.0
diffServDataPathStart[2][inbound] = OID: zeroDotZero.0
diffServDataPathStart[2][outbound] = OID: zeroDotZero.0
diffServDataPathStart[3][inbound] = OID: zeroDotZero.0
diffServDataPathStart[3][outbound] = OID: zeroDotZero.0
diffServDataPathStart[4][inbound] = OID: zeroDotZero.0
diffServDataPathStart[4][outbound] = OID: diffServClfrElementId[1][1]
diffServDataPathStatus[1][inbound] = notInService(2)
diffServDataPathStatus[1][outbound] = notInService(2)
diffServDataPathStatus[2][inbound] = notInService(2)
diffServDataPathStatus[2][outbound] = notInService(2)
diffServDataPathStatus[3][inbound] = notInService(2)
diffServDataPathStatus[3][outbound] = notInService(2)
diffServDataPathStatus[4][inbound] = notInService(2)
diffServDataPathStatus[4][outbound] = active(1)
diffServClfrNextFree.0 = 3
diffServClfrDataPathStart[1] = OID: zeroDotZero.0
diffServClfrDataPathStart[2] = OID: zeroDotZero.0
diffServClfrStatus[1] = active(1)
diffServClfrStatus[2] = active(2)
diffServClfrElementNextFree.0 = 4
diffServClfrElementPrecedence[1][1] = 1
diffServClfrElementPrecedence[2][2] = 1
diffServClfrElementPrecedence[2][3] = 1
diffServClfrElementNext[1][1] = OID: diffServClfrElementId[2]
diffServClfrElementNext[2][2] = OID: diffServAlgDropId[1]
diffServClfrElementNext[2][3] = OID: diffServQId[2]
diffServClfrElementSpecific[1][1] = OID: diffServSixTupleClfrId[1]
diffServClfrElementSpecific[2][2] = OID: diffServSixTupleClfrId[2]
diffServClfrElementSpecific[2][3] = OID: diffServSixTupleClfrId[3]
```

```
diffServClfrElementStatus[1][1] = active(2)
diffServClfrElementStatus[2][2] = active(2)
diffServClfrElementStatus[2][3] = active(2)
diffServSixTupleClfrNextFree.0 = 4
diffServSixTupleClfrDstAddrType[1] = unknown(0)
diffServSixTupleClfrDstAddrType[2] = unknown(0)
diffServSixTupleClfrDstAddrType[3] = unknown(0)
diffServSixTupleClfrDstAddr[1] = ""
diffServSixTupleClfrDstAddr[2] = ""
diffServSixTupleClfrDstAddr[3] = ""
diffServSixTupleClfrSrcAddrType[1] = unknown(0)
diffServSixTupleClfrSrcAddrType[2] = unknown(0)
diffServSixTupleClfrSrcAddrType[3] = unknown(0)
diffServSixTupleClfrSrcAddr[1] = ""
diffServSixTupleClfrSrcAddr[2] = ""
diffServSixTupleClfrSrcAddr[3] = ""
diffServSixTupleClfrDscp[1] = -1
diffServSixTupleClfrDscp[2] = 46
diffServSixTupleClfrDscp[3] = 0
diffServSixTupleClfrProtocol[1] = 255
diffServSixTupleClfrProtocol[2] = 255
diffServSixTupleClfrProtocol[3] = 255
diffServSixTupleClfrStatus[1] = active(1)
diffServSixTupleClfrStatus[2] = active(1)
diffServSixTupleClfrStatus[3] = active(1)
diffServMeterNextFree.0 = 1
diffServTBParamNextFree.0 = 1
diffServActionNextFree.0 = 1
diffServCountActNextFree.0 = 1
diffServAlgDropNextFree.0 = 2
diffServAlgDropType = randomDrop(4)
diffServAlgDropNext = OID: diffServQId[1]
diffServAlgDropQMeasure = diffServQId[1]
diffServAlgDropQThreshold = 0
diffServAlgDropSpecific = OID: diffServRandomDropId[1]
diffServAlgDropOctets = 0
diffServAlgDropHCOctets = 0
diffServAlgDropPkts = 0
diffServAlgDropHCPkts = 0
diffServAlgDropDiscontinuityTime = Timeticks: (0) 0:00:00.00
diffServAlgDropStatus[1] = active(1)
diffServRandomDropNextFree.0 = 2
diffServRandomDropMinThreshBytes = 15000
diffServRandomDropMinThreshPkts = 0
diffServRandomDropMaxThreshBytes = 45000
diffServRandomDropMaxThreshPkts = 0
diffServRandomDropProbMax = 0.4
diffServRandomDropWeight = 0
diffServRandomDropSamplingRate = 0
diffServRandomDropStatus = active(1)
diffServQNextFree.0 = 3
diffServQNext[1] = OID: diffServSchedulerId[1]
diffServQNext[2] = OID: diffServSchedulerId[1]
diffServQRate[1] = OID: zeroDotZero.0
diffServQRate[2] = OID: diffServAssuredRateId[1]
diffServQShaper[1] = OID: zeroDotZero.0
diffServQShaper[2] = OID: zeroDotZero.0
diffServQStatus[1] = active(1)
diffServQStatus[2] = active(1)
diffServSchedulerNextFree.0 = 1
diffServSchedulerNext = OID: zeroDotZero.0
diffServSchedulerMethod = diffServSchedulerWRR.0
diffServSchedulerRate = OID: zeroDotZero.0
diffServSchedulerShaper = OID: zeroDotZero.0
```

```
diffServSchedulerStatus = active(1)
diffServAssuredRateNextFree.0 = 2
diffServAssuredRatePriority[1] = 1
diffServAssuredRateAbs[1] = 1500
diffServAssuredRateRel[1] = 15
diffServAssuredRateStatus[1] = active(1)
diffServShapingRateNextFree.0 = 1
```

Appendix D

RTNetlink Communication

This appendix gives some insight into the rtnetlink communication, used for exchanging routing and network traffic control information between kernel and user-space processes.

All rtnetlink messages consist of a netlink message header and appended attributes. The header gives information about the rest of the header, which is necessary for message parsing (decoding) purposes:

```
struct nlmsghdr
{
  __u32 nlmsg_len;  /* Length of message including header */
  __u16 nlmsg_type; /* Message content */
  __u16 nlmsg_flags;/* Additional flags */
  __u32 nlmsg_seq;  /* Sequence number */
  __u32 nlmsg_pid;  /* PID of the process that opened the socket */
};
```

Traffic control messages (`RTM_QDISC`, `RTM_TCLASS` and `RTM_TFILTER`) contain a `tcmsg` after the initial header, structured as follows:

```
struct tcmsg
{
  unsigned char tcm_family;  /* NETLINK or UNSPEC */
  int           tcm_ifindex; /* Interface index */
  __u32         tcm_handle;  /* Handle */
  __u32         tcm_parent;  /* Handle of parent */
  __u32         tcm_info;    /* Additional information */
};
```

When information is returned from the kernel to the user-space process, the remainder of the message contains "attributes" with the actual information. This information is accessed by typecasting the data starting at memory address `NLMSG_DATA(msg)` to a `struct rtattr`:

```
struct rtattr
{
  unsigned short rta_len;   /* length of this attribute */
  unsigned short rta_type;  /* attribute type */
};
```

The contents of this attribute are not structured, and follow immediately (although aligned) after this “header”. It is now possible to access all attributes in a message by looking at the respective `rta_len` values combined with the value of `nlmsg_len`.

For more information, please refer to the `netlink(7)` and `rtnetlink(7)` manual pages and `<linux/netlink.h>` and `<linux/rtnetlink.h>` header files on a Linux system. Also the source code of the `iproute` package[17] might help in understanding rtnetlink.

Appendix E

Report sent to the IETF and Linux DiffServ implementors

Included below is the short report that has been sent to the `diffserv@ietf.org` and `diffserv-general@lists.sourceforge.net` mailing lists. These are the IETF diffserv WG and Linux DiffServ lists.

```
                         Prototyping the DiffServ MIB
                              on a Linux router

                                 August 2001
                              Remco van de Meent
                            <meent@cs.utwente.nl>

1. Introduction

This is an abstract of the result of the M.Sc assignment the author of this
report did at the University of Twente. The IETF diffserv WG[1] is working on
an SMIv2 MIB[2] for devices implementing the Differentiated Services
Architecture[3].

In this assignment, the 10th revision of the DiffServ MIB Internet Draft has
been prototyped. The prototyping environment is a router running the GNU/Linux
operating system, enhanced with DiffServ support from EPFL[4,5], running a
version 2.4 kernel. The full report as well as the C program code will be made
available from
http://www.simpleweb.org/nm/education/assignments/previous-assignments.html

The focus has been the monitoring part of management. Configurational aspects
have not been implemented, due to complexity and time constraints.


2. DiffServ Architecture and DiffServ MIB

The architecture defines a scalable and efficient way of service
differentiating on the Internet, by aggregating traffic classification state
information using the DSCP field in the IP-header.

The DiffServ architecture splits its functions up in various functional
elements, as outlined in the following figure:
```

```
                                +-------+
                                |       |--------------------+
                         +----->| Meter |                    |
                         |      |       |--+                 |
                         |      +-------+  |                 |
                         |                 V                 V
                   +------------+      +--------+      +---------+
                   |            |      |        |      | Shaper/ |
     packets =====>| Classifier |=====>| Marker |=====>| Dropper |=====>
                   |            |      |        |      |         |
                   +------------+      +--------+      +---------+
```

The DiffServ MIB is closely modelled after this architecture. RowPointers are used to ‘‘hop’’ fromone functional element to another. Parameterization of the various functions is achieved by the use of ‘‘specific’’ tables. This gives a flexible structure, where third parties can add their own tables with specific information.

3. Linux Network Traffic Control Architecture

The Linux Network Traffic Control Architecture (TC) consists of three main elements: queuing disciplines (qdisc), classes and filters. It is heavily focused on the egress part of a router. Support for DiffServ has been built on top of this generic framework.

Qdiscs determine how packets for certain interfaces or classes are treated, like FIFO or RED. Various classes of traffic may be distinguished using filters, which use IP header information (addressing, DSCP). Filters are also used to police (meter) traffic to make sure it matches a certain profile, with a fall-trough mechanism for traffic that is not in profile. The concept of Class Based Queueing (CBQ) is used to treat various classes of traffic (e.g. ‘‘low’’ and ‘‘high’’ priority) in different ways.

The picture below gives an example of a configuration that might be used to implement the Expedited Forwarding per-hop behaviour. Class 1 matches traffic with DSCP value 0x2e set, class 2 everything else.

```
          +--+   +--+               +-------+   +----+     +--+   +--+
          |  |   |  |   +------+ .->|class 1|-->|FIFO|-+   |  |   |  |
          |  |   |  |-->|filter|-'  +-------+   +----+ |   |  |   |  |
          |  |   |  |   +------+                       |   |  |   |  |
 packets  |  |-->|  |      |        +-------+   +---+  '.  |  |-->|  |
 ========>|  |   |  |      +------->|class 2|-->|RED|----->|  |   |  |====>
          |  |   |  |               +-------+   +---+      |  |   |  |
          |  |   |  +--------------------------------------+  |   |  |
          |  |   |    class based queueing                     |   |  |
          |  |   +--------------------------------------------+   |  |
          |  |                                                    |  |
          |  +----------------------------------------------------+  |
          |    sch_dsmark   queueing discipline                      |
          +----------------------------------------------------------+
```

In the TC, the DiffServ concepts of Queueing and Scheduling are taken care of by the qdiscs and classes, Classification is done by filters and classes and Metering is performed by filters. TC is not tailored for DiffServ, and these mappings are non-trivial. The architectures are conceptually different.

Monitoring and configuration of TC in the kernel is done over ‘‘netlink’’ sockets, which is a special socket for exchanging routing and traffic control information (among other things) between kernel and user-space processes.

4. DiffServ MIB and the Linux Kernel

As outlined in the previous paragraphs, both architectures are quite different from each other. As the MIB closely follows the DiffServ Architecture, problems arise when trying to monitor or configure DiffServ functionality in the Linux kernel with this MIB.

For instance, the Counter element is used in the DiffServ Architecture for counting packets that it receives at its input. Queueing disciplines however keep some statistical information, including the number of dropped packets, but it is not possible to attach a counter to a filter.

Not every table in the (current version of the) MIB can be filled with sensible information, because of such constraints. Some of the flexibility the MIB offers can’t be used when managing DiffServ on a Linux router.

5. Conclusions and Recommendations

Routers that are not strictly modelled after the DiffServ Architecture may or may not be fully managable using the DiffServ MIB. In the case of Linux using the DiffServ implementation from EPFL it is not. Also there are better ways in Linux to classifify and mark packets then using TC: the netfilter architecture with its iptables command line utility. It might be possible to create a link between a MIB implementation and netfilter, but netlink sockets cannot be used as they are not supported by netfilter.

This doesn’t mean that the MIB is not good: it seems well-equipped to manage DiffServ functionality on routers implementing the DiffServ Architecture. The use of RowPointers gives enormous flexibility. So the author doesn’t think the MIB should be changed in this respect. Linux might not just be the easiest platform to implement this MIB on.

Furthermore it doesn’t necessarily mean that the management interface to TC should be changed. It gives access to all the flexibility TC offers, and it wouldn’t be wise to drop part of that. Management of TC using MIBs is possible, but has not been implemented yet, as far as the author knows. Interaction with the DiffServ MIB can certainly be achieved using RowPointers.

6. References

[1] IETF diffserv WG, Differentiated Services Working Group Charter Page
http://www.ietf.org/html.charters/diffserv-charter.html

[2] Baker et al. Management Information Base for the Differentiated Services Architecture.
http://www.ietf.org/internet-drafts/draft-iets-diffserv-mib-10.txt
June 2001 (work in progress).

[3] Blake et al. Architecture for Differentiated Services (RFC 2475), December 1998.
http://www.ietf.org/rfc/rfc2475.txt

[4] Differentiated Services on Linux, Homepage
http://diffserv.sourceforge.net/

[5] Almesbergers et al. Differentiated Services on Linux.
ftp://icaftp.epfl.ch/pub/linux/diffserv/misc/dsid-01.ps.gz,
June 1999