Network Working Group
Request for Comments: 3290
Category: Informational

Y. Bernet
Microsoft
S. Blake
Ericsson
D. Grossman
Motorola
A. Smith
Harbour Networks

May 2002

An Informal Management Model for Diffserv Routers

Status of this Memo

This memo provides information for the Internet community. It does not specify an Internet standard of any kind. Distribution of this memo is unlimited.

Copyright Notice

Abstract

This document proposes an informal management model of Differentiated Services (Diffserv) routers for use in their management and configuration. This model defines functional datapath elements (e.g., classifiers, meters, actions, marking, absolute dropping, counting, multiplexing), algorithmic droppers, queues and schedulers. It describes possible configuration parameters for these elements and how they might be interconnected to realize the range of traffic conditioning and per-hop behavior (PHB) functionalities described in the Diffserv Architecture.

Table of Contents

1 Introduction 3
2 Glossary 4
3 Conceptual Model 7
3.1 Components of a Diffserv Router 7
3.1.1 Datapath 7
3.1.2 Configuration and Management Interface 9
3.1.3 Optional QoS Agent Module 10
3.2 Diffserv Functions at Ingress and Egress 10
3.3 Shaping and Policing 12
3.4 Hierarchical View of the Model 12
4 Classifiers 13

4.1 Definition 13
4.1.1 Filters 15
4.1.2 Overlapping Filters 15
4.2 Examples 16
4.2.1 Behavior Aggregate (BA) Classifier 16
4.2.2 Multi-Field (MF) Classifier 17
4.2.3 Free-form Classifier 17
4.2.4 Other Possible Classifiers 18
5 Meters 19
5.1 Examples 20
5.1.1 Average Rate Meter 20
5.1.2 Exponential Weighted Moving Average (EWMA) Meter 21
5.1.3 Two-Parameter Token Bucket Meter 21
5.1.4 Multi-Stage Token Bucket Meter 22
5.1.5 Null Meter 23
6 Action Elements 23
6.1 DSCP Marker 24
6.2 Absolute Dropper 24
6.3 Multiplexor 25
6.4 Counter 25
6.5 Null Action 25
7 Queuing Elements 25
7.1 Queuing Model 26
7.1.1 FIFO Queue 27
7.1.2 Scheduler 28
7.1.3 Algorithmic Dropper 30
7.2 Sharing load among traffic streams using queuing 33
7.2.1 Load Sharing 34
7.2.2 Traffic Priority 35
8 Traffic Conditioning Blocks (TCBs) 35
8.1 TCB 36
8.1.1 Building blocks for Queuing 37
8.2 An Example TCB 37
8.3 An Example TCB to Support Multiple Customers 42
8.4 TCBs Supporting Microflow-based Services 44
8.5 Cascaded TCBs 47
9 Security Considerations 47
10 Acknowledgments 47
11 References 47
Appendix A. Discussion of Token Buckets and Leaky Buckets 50
Authors' Addresses 55
Full Copyright Statement....................................... 56

1. Introduction

Differentiated Services (Diffserv) [DSARCH] is a set of technologies which allow network service providers to offer services with different kinds of network quality-of-service (QoS) objectives to different customers and their traffic streams. This document uses terminology defined in [DSARCH] and [NEWTERMS] (some of these definitions are included here in Section 2 for completeness).

The premise of Diffserv networks is that routers within the core of the network handle packets in different traffic streams by forwarding them using different per-hop behaviors (PHBs). The PHB to be applied is indicated by a Diffserv codepoint (DSCP) in the IP header of each packet [DSFIELD]. The DSCP markings are applied either by a trusted upstream node, e.g., a customer, or by the edge routers on entry to the Diffserv network.

The advantage of such a scheme is that many traffic streams can be aggregated to one of a small number of behavior aggregates (BA), which are each forwarded using the same PHB at the router, thereby simplifying the processing and associated storage. In addition, there is no signaling other than what is carried in the DSCP of each packet, and no other related processing that is required in the core of the Diffserv network since QoS is invoked on a packet-by-packet basis.

The Diffserv architecture enables a variety of possible services which could be deployed in a network. These services are reflected to customers at the edges of the Diffserv network in the form of a Service Level Specification (SLS - see [NEWTERMS]). Whilst further discussion of such services is outside the scope of this document (see [PDBDEF]), the ability to provide these services depends on the availability of cohesive management and configuration tools that can be used to provision and monitor a set of Diffserv routers in a coordinated manner. To facilitate the development of such configuration and management tools it is helpful to define a conceptual model of a Diffserv router that abstracts away implementation details of particular Diffserv routers from the parameters of interest for configuration and management. The purpose of this document is to define such a model.

The basic forwarding functionality of a Diffserv router is defined in other specifications; e.g., [DSARCH, DSFIELD, AF-PHB, EF-PHB].

This document is not intended in any way to constrain or to dictate the implementation alternatives of Diffserv routers. It is expected that router implementers will demonstrate a great deal of variability in their implementations. To the extent that implementers are able to model their implementations using the abstractions described in this document, configuration and management tools will more readily be able to configure and manage networks incorporating Diffserv routers of assorted origins.

This model is intended to be abstract and capable of representing the configuration parameters important to Diffserv functionality for a variety of specific router implementations. It is not intended as a guide to system implementation nor as a formal modeling description. This model serves as the rationale for the design of an SNMP MIB [DSMIB] and for other configuration interfaces (e.g., other policy-management protocols) and, possibly, more detailed formal models (e.g., [QOSDEVMOD]): these should all be consistent with this model.

- o Section 3 starts by describing the basic high-level blocks of a Diffserv router. It explains the concepts used in the model, including the hierarchical management model for these blocks which uses low-level functional datapath elements such as Classifiers, Actions, Queues.

- o Section 4 describes Classifier elements.

- o Section 5 discusses Meter elements.

- o Section 6 discusses Action elements.

- o Section 7 discusses the basic queuing elements of Algorithmic Droppers, Queues, and Schedulers and their functional behaviors (e.g., traffic shaping).

- o Section 8 shows how the low-level elements can be combined to build modules called Traffic Conditioning Blocks (TCBs) which are useful for management purposes.

- o Section 9 discusses security concerns.

- o Appendix A contains a brief discussion of the token bucket and leaky bucket algorithms used in this model and some of the practical effects of the use of token buckets within the Diffserv architecture.

2. Glossary

This document uses terminology which is defined in [DSARCH]. There is also current work-in-progress on this terminology in the IETF and some of the definitions provided here are taken from that work. Some of the terms from these other references are defined again here in order to provide additional detail, along with some new terms specific to this document.

Absolute Dropper
A functional datapath element which simply discards all packets arriving at its input.

Algorithmic Dropper
A functional datapath element which selectively discards packets that arrive at its input, based on a discarding algorithm. It has one data input and one output.

Classifier
A functional datapath element which consists of filters that select matching and non-matching packets. Based on this selection, packets are forwarded along the appropriate datapath within the router. A classifier, therefore, splits a single incoming traffic stream into multiple outgoing streams.

Counter
A functional datapath element which updates a packet counter and also an octet counter for every packet that passes through it.

Datapath
A conceptual path taken by packets with particular characteristics through a Diffserv router. Decisions as to the path taken by a packet are made by functional datapath elements such as Classifiers and Meters.

Filter
A set of wildcard, prefix, masked, range and/or exact match conditions on the content of a packet’s headers or other data, and/or on implicit or derived attributes associated with the packet. A filter is said to match only if each condition is satisfied.

Functional Datapath Element
A basic building block of the conceptual router. Typical elements are Classifiers, Meters, Actions, Algorithmic Droppers, Queues and Schedulers.

Multiplexer (Mux)
A multiplexor.

Multiplexor (Mux)
A functional datapath element that merges multiple traffic streams (datapaths) into a single traffic stream (datapath).

Non-work-conserving
A property of a scheduling algorithm such that it services packets no sooner than a scheduled departure time, even if this means leaving packets queued while the output (e.g., a network link or connection to the next element) is idle.

Policing
The process of comparing the arrival of data packets against a temporal profile and forwarding, delaying or dropping them so as to make the output stream conformant to the profile.

Queuing Block
A combination of functional datapath elements that modulates the transmission of packets belonging to a traffic streams and determines their ordering, possibly storing them temporarily or discarding them.

Scheduling algorithm
An algorithm which determines which queue of a set of queues to service next. This may be based on the relative priority of the queues, on a weighted fair bandwidth sharing policy or some other policy. Such an algorithm may be either work-conserving or non-work-conserving.

Service-Level Specification (SLS)
A set of parameters and their values which together define the treatment offered to a traffic stream by a Diffserv domain.

Shaping
The process of delaying packets within a traffic stream to cause it to conform to some defined temporal profile. Shaping can be implemented using a queue serviced by a non-work-conserving scheduling algorithm.

Traffic Conditioning Block (TCB)
A logical datapath entity consisting of a number of functional datapath elements interconnected in such a way as to perform a specific set of traffic conditioning functions on an incoming traffic stream. A TCB can be thought of as an entity with one input and one or more outputs and a set of control parameters.

Traffic Conditioning Specification (TCS)
A set of parameters and their values which together specify a set of classifier rules and a traffic profile. A TCS is an integral element of a SLS.

Work-conserving   A property of a scheduling algorithm such that it services a packet, if one is available, at every transmission opportunity.

3. Conceptual Model

This section introduces a block diagram of a Diffserv router and describes the various components illustrated in Figure 1. Note that a Diffserv core router is likely to require only a subset of these components: the model presented here is intended to cover the case of both Diffserv edge and core routers.

3.1. Components of a Diffserv Router

The conceptual model includes abstract definitions for the following:

- o Traffic Classification elements.
- o Metering functions.
- o Actions of Marking, Absolute Dropping, Counting, and Multiplexing.
- o Queuing elements, including capabilities of algorithmic dropping and scheduling.
- o Certain combinations of the above functional datapath elements into higher-level blocks known as Traffic Conditioning Blocks (TCBs).

The components and combinations of components described in this document form building blocks that need to be manageable by Diffserv configuration and management tools. One of the goals of this document is to show how a model of a Diffserv device can be built using these component blocks. This model is in the form of a connected directed acyclic graph (DAG) of functional datapath elements that describes the traffic conditioning and queuing behaviors that any particular packet will experience when forwarded to the Diffserv router. Figure 1 illustrates the major functional blocks of a Diffserv router.

3.1.1. Datapath

An ingress interface, routing core, and egress interface are illustrated at the center of the diagram. In actual router implementations, there may be an arbitrary number of ingress and egress interfaces interconnected by the routing core. The routing core element serves as an abstraction of a router’s normal routing and switching functionality. The routing core moves packets between interfaces according to policies outside the scope of Diffserv (note: it is possible that such policies for output-interface selection might involve use of packet fields such as the DSCP but this is outside the scope of this model). The actual queuing delay and packet loss behavior of a specific router’s switching fabric/backplane is not modeled by the routing core; these should be modeled using the functional datapath elements described later. The routing core of this model can be thought of as an infinite bandwidth, zero-delay interconnect between interfaces - properties like the behavior of the core when overloaded need to be reflected back into the queuing elements that are modeled around it (e.g., when too much traffic is directed across the core at an egress interface), the excess must either be dropped or queued somewhere: the elements performing these functions must be modeled on one of the interfaces involved.

The components of interest at the ingress to and egress from interfaces are the functional datapath elements (e.g., Classifiers, Queuing elements) that support Diffserv traffic conditioning and per-hop behaviors [DSARCH]. These are the fundamental components comprising a Diffserv router and are the focal point of this model.

```
               +---------------+
               | Diffserv      |
        Mgmt   | configuration |
      <----+-->| & management  |------------------+
      SNMP,|   | interface     |                  |
      COPS |   +---------------+                  |
      etc. |        |                             |
           |        |                             |
           |        v                             v
           |   +-------------+                 +-------------+
           |   | ingress i/f |   +---------+   | egress i/f  |
      -------->|  classify,  |-->| routing |-->|  classify,  |---->
      data |   |  meter,     |   |  core   |   |  meter      |data out
      in   |   |  action,    |   +---------+   |  action,    |
           |   |  queuing    |                 |  queuing    |
           |   +-------------+                 +-------------+
           |        ^                             ^
           |        |                             |
           |        |                             |
           |   +------------+                     |
           +-->| QOS agent  |                     |
      -------->| (optional) |---------------------+
        QOS    |(e.g., RSVP)|
        cntl   +------------+
        msgs
```

Figure 1: Diffserv Router Major Functional Blocks

3.1.2. Configuration and Management Interface

Diffserv operating parameters are monitored and provisioned through this interface. Monitored parameters include statistics regarding traffic carried at various Diffserv service levels. These statistics may be important for accounting purposes and/or for tracking compliance to Traffic Conditioning Specifications (TCSs) negotiated with customers. Provisioned parameters are primarily the TCS parameters for Classifiers and Meters and the associated PHB configuration parameters for Actions and Queuing elements. The network administrator interacts with the Diffserv configuration and management interface via one or more management protocols, such as SNMP or COPS, or through other router configuration tools such as serial terminal or telnet consoles.

Specific policy rules and goals governing the Diffserv behavior of a router are presumed to be installed by policy management mechanisms. However, Diffserv routers are always subject to implementation limits which scope the kinds of policies which can be successfully implemented by the router. External reporting of such implementation capabilities is considered out of scope for this document.

3.1.3. Optional QoS Agent Module

Diffserv routers may snoop or participate in either per-microflow or per-flow-aggregate signaling of QoS requirements [E2E] (e.g., using the RSVP protocol). Snooping of RSVP messages may be used, for example, to learn how to classify traffic without actually participating as a RSVP protocol peer. Diffserv routers may reject or admit RSVP reservation requests to provide a means of admission control to Diffserv-based services or they may use these requests to trigger provisioning changes for a flow-aggregation in the Diffserv network. A flow-aggregation in this context might be equivalent to a Diffserv BA or it may be more fine-grained, relying on a multi-field (MF) classifier [DSARCH]. Note that the conceptual model of such a router implements the Integrated Services Model as described in [INTSERV], applying the control plane controls to the data classified and conditioned in the data plane, as described in [E2E].

Note that a QoS Agent component of a Diffserv router, if present, might be active only in the control plane and not in the data plane. In this scenario, RSVP could be used merely to signal reservation state without installing any actual reservations in the data plane of the Diffserv router: the data plane could still act purely on Diffserv DSCPs and provide PHBs for handling data traffic without the normal per-microflow handling expected to support some Intserv services.

3.2. Diffserv Functions at Ingress and Egress

This document focuses on the Diffserv-specific components of the router. Figure 2 shows a high-level view of ingress and egress interfaces of a router. The diagram illustrates two Diffserv router interfaces, each having a set of ingress and a set of egress elements. It shows classification, metering, action and queuing functions which might be instantiated at each interface’s ingress and egress.

The simple diagram of Figure 2 assumes that the set of Diffserv functions to be carried out on traffic on a given interface are independent of those functions on all other interfaces. There are some architectures where Diffserv functions may be shared amongst multiple interfaces (e.g., processor and buffering resources that handle multiple interfaces on the same line card before forwarding across a routing core). The model presented in this document may be easily extended to handle such cases; however, this topic is not treated further here as it leads to excessive complexity in the explanation of the concepts.

```
         Interface A                        Interface B
       +-------------+     +---------+     +-------------+
       | ingress:    |     |         |     | egress:     |
       |   classify, |     |         |     |   classify, |
    --->|   meter,    |---->|         |---->|   meter,    |--->
       |   action,   |     |         |     |   action,   |
       |   queuing   |     | routing |     |   queuing   |
       +-------------+     |  core   |     +-------------+
       | egress:     |     |         |     | ingress:    |
       |   classify, |     |         |     |   classify, |
    <---|   meter,    |<----|         |<----|   meter,    |<---
       |   action,   |     |         |     |   action,   |
       |   queuing   |     +---------+     |   queuing   |
       +-------------+                     +-------------+
```

Figure 2. Traffic Conditioning and Queuing Elements

In principle, if one were to construct a network entirely out of two-port routers (connected by LANs or similar media), then it might be necessary for each router to perform four QoS control functions in the datapath on traffic in each direction:

- Classify each message according to some set of rules, possibly just a "match everything" rule.

- If necessary, determine whether the data stream the message is part of is within or outside its rate by metering the stream.

- Perform a set of resulting actions, including applying a drop policy appropriate to the classification and queue in question and perhaps additionally marking the traffic with a Differentiated Services Code Point (DSCP) [DSFIELD].

- Enqueue the traffic for output in the appropriate queue. The scheduling of output from this queue may lead to shaping of the traffic or may simply cause it to be forwarded with some minimum rate or maximum latency assurance.

If the network is now built out of N-port routers, the expected behavior of the network should be identical. Therefore, this model must provide for essentially the same set of functions at the ingress as on the egress of a router’s interfaces. The one point of difference in the model between ingress and the egress is that all traffic at the egress of an interface is queued, while traffic at the ingress to an interface is likely to be queued only for shaping purposes, if at all. Therefore, equivalent functional datapath elements may be modeled at both the ingress to and egress from an interface.

Note that it is not mandatory that each of these functional datapath elements be implemented at both ingress and egress; equally, the model allows that multiple sets of these elements may be placed in series and/or in parallel at ingress or at egress. The arrangement of elements is dependent on the service requirements on a particular interface on a particular router. By modeling these elements at both ingress and egress, it is not implied that they must be implemented in this way in a specific router. For example, a router may implement all shaping and PHB queuing at the interface egress or may instead implement it only at the ingress. Furthermore, the classification needed to map a packet to an egress queue (if present) need not be implemented at the egress but instead might be implemented at the ingress, with the packet passed through the routing core with in-band control information to allow for egress queue selection.

Specifically, some interfaces will be at the outer "edge" and some will be towards the "core" of the Diffserv domain. It is to be expected (from the general principles guiding the motivation of Diffserv) that "edge" interfaces, or at least the routers that contain them, will implement more complexity and require more configuration than those in the core although this is obviously not a requirement.

3.3. Shaping and Policing

Diffserv nodes may apply shaping, policing and/or marking to traffic streams that exceed the bounds of their TCS in order to prevent one traffic stream from seizing more than its share of resources from a Diffserv network. In this model, Shaping, sometimes considered as a TC action, is treated as a function of queuing elements - see section 7. Algorithmic Dropping techniques (e.g., RED) are similarly treated since they are often closely associated with queues. Policing is modeled as either a concatenation of a Meter with an Absolute Dropper or as a concatenation of an Algorithmic Dropper with a Scheduler. These elements will discard packets which exceed the TCS.

3.4. Hierarchical View of the Model

From a device-level configuration management perspective, the following hierarchy exists:

At the lowest level considered here, there are individual functional datapath elements, each with their own configuration parameters and management counters and flags.

At the next level, the network administrator manages groupings of these functional datapath elements interconnected in a DAG. These functional datapath elements are organized in self-contained TCBs which are used to implement some desired network policy (see Section 8). One or more TCBs may be instantiated at each interface’s ingress or egress; they may be connected in series and/or in parallel configurations on the multiple outputs of a preceding TCB. A TCB can be thought of as a "black box" with one input and one or more outputs (in the data path). Each interface may have a different TCB configuration and each direction (ingress or egress) may too.

At the topmost level considered here, the network administrator manages interfaces. Each interface has ingress and egress functionality, with each of these expressed as one or more TCBs. This level of the hierarchy is what was illustrated in Figure 2.

Further levels may be built on top of this hierarchy, in particular ones for aiding in the repetitive configuration tasks likely for routers with many interfaces: some such "template" tools for Diffserv routers are outside the scope of this model but are under study by other working groups within IETF.

4. Classifiers

4.1. Definition

Classification is performed by a classifier element. Classifiers are 1:N (fan-out) devices: they take a single traffic stream as input and generate N logically separate traffic streams as output. Classifiers are parameterized by filters and output streams. Packets from the input stream are sorted into various output streams by filters which match the contents of the packet or possibly match other attributes associated with the packet. Various types of classifiers using different filters are described in the following sections. Figure 3 illustrates a classifier, where the outputs connect to succeeding functional datapath elements.

The simplest possible Classifier element is one that matches all packets that are applied at its input. In this case, the Classifier element is just a no-op and may be omitted.

Note that we allow a Multiplexor (see Section 6.5) before the Classifier to allow input from multiple traffic streams. For example, if traffic streams originating from multiple ingress interfaces feed through a single Classifier then the interface number could be one of the packet classification keys used by the Classifier. This optimization may be important for scalability in the management plane. Classifiers may also be cascaded in sequence to perform more complex lookup operations whilst still maintaining such scalability.

Another example of a packet attribute could be an integer representing the BGP community string associated with the packet’s best-matching route. Other contextual information may also be used by a Classifier (e.g., knowledge that a particular interface faces a Diffserv domain or a legacy IP TOS domain [DSARCH] could be used when determining whether a DSCP is present or not).

```
   unclassified              classified
   traffic                   traffic
           +------------+
           |            |--> match Filter1 --> OutputA
   ------->| classifier |--> match Filter2 --> OutputB
           |            |--> no match      --> OutputC
           +------------+
```

Figure 3. An Example Classifier

The following BA classifier separates traffic into one of three output streams based on matching filters:

Filter Matched	Output Stream
Filter1	A
Filter2	B
no match	C

Where the filters are defined to be the following BA filters ([DSARCH], Section 4.2.1):

Filter	DSCP
Filter1	101010
Filter2	111111
Filter3	****** (wildcard)

4.1.1. Filters

A filter consists of a set of conditions on the component values of a packet’s classification key (the header values, contents, and attributes relevant for classification). In the BA classifier example above, the classification key consists of one packet header field, the DSCP, and both Filter1 and Filter2 specify exact-match conditions on the value of the DSCP. Filter3 is a wildcard default filter which matches every packet, but which is only selected in the event that no other more specific filter matches.

In general there are a set of possible component conditions including exact, prefix, range, masked and wildcard matches. Note that ranges can be represented (with less efficiency) as a set of prefixes and that prefix matches are just a special case of both masked and range matches.

In the case of a MF classifier, the classification key consists of a number of packet header fields. The filter may specify a different condition for each key component, as illustrated in the example below for a IPv4/TCP classifier:

Filter	IPv4 Src Addr	IPv4 Dest Addr	TCP SrcPort	TCP DestPort
Filter4	172.31.8.1/32	172.31.3.X/24	X	5003

In this example, the fourth octet of the destination IPv4 address and the source TCP port are wildcard or "don’t care".

MF classification of IP-fragmented packets is impossible if the filter uses transport-layer port numbers (e.g., TCP port numbers). MTU-discovery is therefore a prerequisite for proper operation of a Diffserv network that uses such classifiers.

4.1.2. Overlapping Filters

Note that it is easy to define sets of overlapping filters in a classifier. For example:

Filter	IPv4 Src Addr	IPv4 Dest Addr
Filter5	172.31.8.X/24	X/0
Filter6	X/0	172.30.10.1/32

A packet containing {IP Dest Addr 172.31.8.1, IP Src Addr 172.30.10.1} cannot be uniquely classified by this pair of filters and so a precedence must be established between Filter5 and Filter6 in order to break the tie. This precedence must be established either (a) by a manager which knows that the router can accomplish this particular ordering (e.g., by means of reported capabilities), or (b) by the router along with a mechanism to report to a manager which precedence is being used. Such precedence mechanisms must be supported in any translation of this model into specific syntax for configuration and management protocols.

As another example, one might want first to disallow certain applications from using the network at all, or to classify some individual traffic streams that are not Diffserv-marked. Traffic that is not classified by those tests might then be inspected for a DSCP. The word "then" implies sequence and this must be specified by means of precedence.

An unambiguous classifier requires that every possible classification key match at least one filter (possibly the wildcard default) and that any ambiguity between overlapping filters be resolved by precedence. Therefore, the classifiers on any given interface must be "complete" and will often include an "everything else" filter as the lowest precedence element in order for the result of classification to be deterministic. Note that this completeness is only required of the first classifier that incoming traffic will meet as it enters an interface - subsequent classifiers on an interface only need to handle the traffic that it is known that they will receive.

This model of classifier operation makes the assumption that all filters of the same precedence be applied simultaneously. Whilst convenient from a modeling point-of-view, this may or may not be how the classifier is actually implemented - this assumption is not intended to dictate how the implementation actually handles this, merely to clearly define the required end result.

4.2. Examples

4.2.1. Behavior Aggregate (BA) Classifier

The simplest Diffserv classifier is a behavior aggregate (BA) classifier [DSARCH]. A BA classifier uses only the Diffserv codepoint (DSCP) in a packet’s IP header to determine the logical output stream to which the packet should be directed. We allow only an exact-match condition on this field because the assigned DSCP values have no structure, and therefore no subset of DSCP bits are significant.

The following defines a possible BA filter:

```
Filter8:
Type:   BA
Value:  111000
```

4.2.2. Multi-Field (MF) Classifier

Another type of classifier is a multi-field (MF) classifier [DSARCH]. This classifies packets based on one or more fields in the packet (possibly including the DSCP). A common type of MF classifier is a 6-tuple classifier that classifies based on six fields from the IP and TCP or UDP headers (destination address, source address, IP protocol, source port, destination port, and DSCP). MF classifiers may classify on other fields such as MAC addresses, VLAN tags, link-layer traffic class fields, or other higher-layer protocol fields.

The following defines a possible MF filter:

```
Filter9:
Type:               IPv4-6-tuple
IPv4DestAddrValue: 0.0.0.0
IPv4DestAddrMask:  0.0.0.0
IPv4SrcAddrValue:  172.31.8.0
IPv4SrcAddrMask:   255.255.255.0
IPv4DSCP:          28
IPv4Protocol:      6
IPv4DestL4PortMin: 0
IPv4DestL4PortMax: 65535
IPv4SrcL4PortMin:  20
IPv4SrcL4PortMax:  20
```

A similar type of classifier can be defined for IPv6.

4.2.3. Free-form Classifier

A Free-form classifier is made up of a set of user definable arbitrary filters each made up of {bit-field size, offset (from head of packet), mask}:

```
Classifier2:
Filter12:    OutputA
Filter13:    OutputB
Default:     OutputC
```

```
      Filter12:
      Type:         FreeForm
      SizeBits:     3 (bits)
      Offset:       16 (bytes)
      Value:        100 (binary)
      Mask:         101 (binary)

      Filter13:
      Type:         FreeForm
      SizeBits:     12 (bits)
      Offset:       16 (bytes)
      Value:        100100000000 (binary)
      Mask:         111111111111 (binary)
```

Free-form filters can be combined into filter groups to form very powerful filters.

4.2.4. Other Possible Classifiers

Classification may also be performed based on information at the datalink layer below IP (e.g., VLAN or datalink-layer priority) or perhaps on the ingress or egress IP, logical or physical interface identifier (e.g., the incoming channel number on a channelized interface). A classifier that filters based on IEEE 802.1p Priority and on 802.1Q VLAN-ID might be represented as:

```
      Classifier3:
      Filter14 AND Filter15:  OutputA
      Default:                OutputB

      Filter14:                        -- priority 4 or 5
      Type:         Ieee8021pPriority
      Value:        100 (binary)
      Mask:         110 (binary)

      Filter15:                        -- VLAN 2304
      Type:         Ieee8021QVlan
      Value:        100100000000 (binary)
      Mask:         111111111111 (binary)
```

Such classifiers may be the subject of other standards or may be proprietary to a router vendor but they are not discussed further here.

5. Meters

Metering is defined in [DSARCH]. Diffserv network providers may choose to offer services to customers based on a temporal (i.e., rate) profile within which the customer submits traffic for the service. In this event, a meter might be used to trigger real-time traffic conditioning actions (e.g., marking) by routing a non-conforming packet through an appropriate next-stage action element. Alternatively, by counting conforming and/or non-conforming traffic using a Counter element downstream of the Meter, it might also be used to help in collecting data for out-of-band management functions such as billing applications.

Meters are logically 1:N (fan-out) devices (although a multiplexor can be used in front of a meter). Meters are parameterized by a temporal profile and by conformance levels, each of which is associated with a meter’s output. Each output can be connected to another functional element.

Note that this model of a meter differs slightly from that described in [DSARCH]. In that description the meter is not a datapath element but is instead used to monitor the traffic stream and send control signals to action elements to dynamically modulate their behavior based on the conformance of the packet. This difference in the description does not change the function of a meter. Figure 4 illustrates a meter with 3 levels of conformance.

In some Diffserv examples (e.g., [AF-PHB]), three levels of conformance are discussed in terms of colors, with green representing conforming, yellow representing partially conforming and red representing non-conforming. These different conformance levels may be used to trigger different queuing, marking or dropping treatment later on in the processing. Other example meters use a binary notion of conformance; in the general case N levels of conformance can be supported. In general there is no constraint on the type of functional datapath element following a meter output, but care must be taken not to inadvertently configure a datapath that results in packet reordering that is not consistent with the requirements of the relevant PHB specification.

```
      unmetered              metered
      traffic                traffic
                +---------+
                |         |--------> conformance A
      --------->|  meter  |--------> conformance B
                |         |--------> conformance C
                +---------+
```

Figure 4. A Generic Meter

A meter, according to this model, measures the rate at which packets making up a stream of traffic pass it, compares the rate to some set of thresholds, and produces some number of potential results (two or more): a given packet is said to be "conformant" to a level of the meter if, at the time that the packet is being examined, the stream appears to be within the rate limit for the profile associated with that level. A fuller discussion of conformance to meter profiles (and the associated requirements that this places on the schedulers upstream) is provided in Appendix A.

5.1. Examples

The following are some examples of possible meters.

5.1.1. Average Rate Meter

An example of a very simple meter is an average rate meter. This type of meter measures the average rate at which packets are submitted to it over a specified averaging time.

An average rate profile may take the following form:

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

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

A Meter measuring against this profile would continually maintain a count that indicates the total number and/or cumulative byte-count of packets arriving between time T (now) and time T - 100 msecs. So long as an arriving packet does not push the count over 12 kbits in the last 100 msec, the packet would be deemed conforming. Any packet that pushes the count over 12 kbits would be deemed non-conforming. Thus, this Meter deems packets to correspond to one of two conformance levels: conforming or non-conforming, and sends them on for the appropriate subsequent treatment.

5.1.2. Exponential Weighted Moving Average (EWMA) Meter

The EWMA form of Meter is easy to implement in hardware and can be parameterized as follows:

```
avg_rate(t) = (1 - Gain) * avg_rate(t’) +  Gain * rate(t)
t = t’ + Delta
```

For a packet arriving at time t:

```
if (avg_rate(t) > AverageRate)
   non-conforming
else
   conforming
```

"Gain" controls the time constant (e.g., frequency response) of what is essentially a simple IIR low-pass filter. "Rate(t)" measures the number of incoming bytes in a small fixed sampling interval, Delta. Any packet that arrives and pushes the average rate over a predefined rate AverageRate is deemed non-conforming. An EWMA Meter profile might look something like the following:

```
Meter2:
Type:                 ExpWeightedMovingAvg
Profile:              Profile2
ConformingOutput:     Queue1
NonConformingOutput: AbsoluteDropper1

Profile2:
Type:                 ExpWeightedMovingAvg
AverageRate:          25 kbps
Delta:                10 usec
Gain:                 1/16
```

5.1.3. Two-Parameter Token Bucket Meter

A more sophisticated Meter might measure conformance to a token bucket (TB) profile. A TB profile generally has two parameters, an average token rate, R, and a burst size, B. TB Meters compare the arrival rate of packets to the average rate specified by the TB profile. Logically, tokens accumulate in a bucket at the average rate, R, up to a maximum credit which is the burst size, B. When a packet of length L arrives, a conformance test is applied. There are at least two such tests in widespread use:

Strict conformance
Packets of length L bytes are considered conforming only if there are sufficient tokens available in the bucket at the time of packet arrival for the complete packet (i.e., the current depth is greater than or equal to L): no tokens may be borrowed from future token allocations. For examples of this approach, see [SRTCM] and [TRTCM].

Loose conformance
Packets of length L bytes are considered conforming if any tokens are available in the bucket at the time of packet arrival: up to L bytes may then be borrowed from future token allocations.

Packets are allowed to exceed the average rate in bursts up to the burst size. For further discussion of loose and strict conformance to token bucket profiles, as well as system and implementation issues, see Appendix A.

A two-parameter TB meter has exactly two possible conformance levels (conforming, non-conforming). Such a meter might appear as follows:

```
Meter3:
Type:                SimpleTokenBucket
Profile:             Profile3
ConformanceType:     loose
ConformingOutput:    Queue1
NonConformingOutput: AbsoluteDropper1

Profile3:
Type:                SimpleTokenBucket
AverageRate:         200 kbps
BurstSize:           100 kbytes
```

5.1.4. Multi-Stage Token Bucket Meter

More complicated TB meters might define multiple burst sizes and more conformance levels. Packets found to exceed the larger burst size are deemed non-conforming. Packets found to exceed the smaller burst size are deemed partially-conforming. Packets exceeding neither are deemed conforming. Some token bucket meters designed for Diffserv networks are described in more detail in [SRTCM, TRTCM]; in some of these references, three levels of conformance are discussed in terms of colors with green representing conforming, yellow representing partially conforming, and red representing non-conforming. Note that these multiple-conformance-level meters can sometimes be implemented using an appropriate sequence of multiple two-parameter TB meters.

A profile for a multi-stage TB meter with three levels of conformance might look as follows:

```
Meter4:
Type:                TwoRateTokenBucket
ProfileA:            Profile4
ConformanceTypeA:    strict
ConformingOutputA:   Queue1

ProfileB:            Profile5
ConformanceTypeB:    strict
ConformingOutputB:   Marker1
NonConformingOutput: AbsoluteDropper1

Profile4:
Type:                SimpleTokenBucket
AverageRate:         100 kbps
BurstSize:           20 kbytes

Profile5:
Type:                SimpleTokenBucket
AverageRate:         100 kbps
BurstSize:           100 kbytes
```

5.1.5. Null Meter

A null meter has only one output: always conforming, and no associated temporal profile. Such a meter is useful to define in the event that the configuration or management interface does not have the flexibility to omit a meter in a datapath segment.

```
Meter5:
Type:                NullMeter
Output:              Queue1
```

6. Action Elements

The classifiers and meters described up to this point are fan-out elements which are generally used to determine the appropriate action to apply to a packet. The set of possible actions that can then be applied include:

- Marking

- Absolute Dropping

- Multiplexing

- Counting

- Null action - do nothing

The corresponding action elements are described in the following sections.

6.1. DSCP Marker

DSCP Markers are 1:1 elements which set a codepoint (e.g., the DSCP in an IP header). DSCP Markers may also act on unmarked packets (e.g., those submitted with DSCP of zero) or may re-mark previously marked packets. In particular, the model supports the application of marking based on a preceding classifier match. The mark set in a packet will determine its subsequent PHB treatment in downstream nodes of a network and possibly also in subsequent processing stages within this router.

DSCP Markers for Diffserv are normally parameterized by a single parameter: the 6-bit DSCP to be marked in the packet header.

```
Marker1:
Type:                DSCPMarker
Mark:                010010
```

6.2. Absolute Dropper

Absolute Droppers simply discard packets. There are no parameters for these droppers. Because this Absolute Dropper is a terminating point of the datapath and has no outputs, it is probably desirable to forward the packet through a Counter Action first for instrumentation purposes.

```
AbsoluteDropper1:
Type:                AbsoluteDropper
```

Absolute Droppers are not the only elements than can cause a packet to be discarded: another element is an Algorithmic Dropper element (see Section 7.1.3). However, since this element’s behavior is closely tied the state of one or more queues, we choose to distinguish it as a separate functional datapath element.

6.3. Multiplexor

It is occasionally necessary to multiplex traffic streams into a functional datapath element with a single input. A M:1 (fan-in) multiplexor is a simple logical device for merging traffic streams. It is parameterized by its number of incoming ports.

```
Mux1:
Type:                 Multiplexor
Output:               Queue2
```

6.4. Counter

One passive action is to account for the fact that a data packet was processed. The statistics that result might be used later for customer billing, service verification or network engineering purposes. Counters are 1:1 functional datapath elements which update a counter by L and a packet counter by 1 every time a L-byte sized packet passes through them. Counters can be used to count packets about to be dropped by an Absolute Dropper or to count packets arriving at or departing from some other functional element.

```
Counter1:
Type:                 Counter
Output:               Queue1
```

6.5. Null Action

A null action has one input and one output. The element performs no action on the packet. Such an element is useful to define in the event that the configuration or management interface does not have the flexibility to omit an action element in a datapath segment.

```
Null1:
Type:                 Null
Output:               Queue1
```

7. Queuing Elements

Queuing elements modulate the transmission of packets belonging to the different traffic streams and determine their ordering, possibly storing them temporarily or discarding them. Packets are usually stored either because there is a resource constraint (e.g., available bandwidth) which prevents immediate forwarding, or because the queuing block is being used to alter the temporal properties of a traffic stream (i.e., shaping). Packets are discarded for one of the following reasons:

- because of buffering limitations.
- because a buffer threshold is exceeded (including when shaping is performed).
- as a feedback control signal to reactive control protocols such as TCP.
- because a meter exceeds a configured profile (i.e., policing).

The queuing elements in this model represent a logical abstraction of a queuing system which is used to configure PHB-related parameters. The model can be used to represent a broad variety of possible implementations. However, it need not necessarily map one-to-one with physical queuing systems in a specific router implementation. Implementors should map the configurable parameters of the implementation’s queuing systems to these queuing element parameters as appropriate to achieve equivalent behaviors.

7.1. Queuing Model

Queuing is a function which lends itself to innovation. It must be modeled to allow a broad range of possible implementations to be represented using common structures and parameters. This model uses functional decomposition as a tool to permit the needed latitude.

Queuing systems perform three distinct, but related, functions: they store packets, they modulate the departure of packets belonging to various traffic streams and they selectively discard packets. This model decomposes queuing into the component elements that perform each of these functions: Queues, Schedulers, and Algorithmic Droppers, respectively. These elements may be connected together as part of a TCB, as described in section 8.

The remainder of this section discusses FIFO Queues: typically, the Queue element of this model will be implemented as a FIFO data structure. However, this does not preclude implementations which are not strictly FIFO, in that they also support operations that remove or examine packets (e.g., for use by discarders) other than at the head or tail. However, such operations must not have the effect of reordering packets belonging to the same microflow.

Note that the term FIFO has multiple different common usages: it is sometimes taken to mean, among other things, a data structure that permits items to be removed only in the order in which they were inserted or a service discipline which is non-reordering.

7.1.1. FIFO Queue

In this model, a FIFO Queue element is a data structure which at any time may contain zero or more packets. It may have one or more thresholds associated with it. A FIFO has one or more inputs and exactly one output. It must support an enqueue operation to add a packet to the tail of the queue and a dequeue operation to remove a packet from the head of the queue. Packets must be dequeued in the order in which they were enqueued. A FIFO has a current depth, which indicates the number of packets and/or bytes that it contains at a particular time. FIFOs in this model are modeled without inherent limits on their depth - obviously this does not reflect the reality of implementations: FIFO size limits are modeled here by an algorithmic dropper associated with the FIFO, typically at its input. It is quite likely that every FIFO will be preceded by an algorithmic dropper. One exception might be the case where the packet stream has already been policed to a profile that can never exceed the scheduler bandwidth available at the FIFO’s output - this would not need an algorithmic dropper at the input to the FIFO.

This representation of a FIFO allows for one common type of depth limit, one that results from a FIFO supplied from a limited pool of buffers, shared between multiple FIFOs.

In an implementation, packets are presumably stored in one or more buffers. Buffers are allocated from one or more free buffer pools. If there are multiple instances of a FIFO, their packet buffers may or may not be allocated out of the same free buffer pool. Free buffer pools may also have one or more thresholds associated with them, which may affect discarding and/or scheduling. Other than this, buffering mechanisms are implementation specific and not part of this model.

A FIFO might be represented using the following parameters:

```
Queue1:
Type:       FIFO
Output:     Scheduler1
```

Note that a FIFO must provide triggers and/or current state information to other elements upstream and downstream from it: in particular, it is likely that the current depth will need to be used by Algorithmic Dropper elements placed before or after the FIFO. It will also likely need to provide an implicit "I have packets for you" signal to downstream Scheduler elements.

7.1.2. Scheduler

A scheduler is an element which gates the departure of each packet that arrives at one of its inputs, based on a service discipline. It has one or more inputs and exactly one output. Each input has an upstream element to which it is connected, and a set of parameters that affects the scheduling of packets received at that input.

The service discipline (also known as a scheduling algorithm) is an algorithm which might take any of the following as its input(s):

a) static parameters such as relative priority associated with each of the scheduler’s inputs.

b) absolute token bucket parameters for maximum or minimum rates associated with each of the scheduler’s inputs.

c) parameters, such as packet length or DSCP, associated with the packet currently present at its input.

d) absolute time and/or local state.

Possible service disciplines fall into a number of categories, including (but not limited to) first come, first served (FCFS), strict priority, weighted fair bandwidth sharing (e.g., WFQ), rate-limited strict priority, and rate-based. Service disciplines can be further distinguished by whether they are work-conserving or non-work-conserving (see Glossary). Non-work-conserving schedulers can be used to shape traffic streams to match some profile by delaying packets that might be deemed non-conforming by some downstream node: a packet is delayed until such time as it would conform to a downstream meter using the same profile.

[DSARCH] defines PHBs without specifying required scheduling algorithms. However, PHBs such as the class selectors [DSFIELD], EF [EF-PHB] and AF [AF-PHB] have descriptions or configuration parameters which strongly suggest the sort of scheduling discipline needed to implement them. This document discusses a minimal set of queue parameters to enable realization of these PHBs. It does not attempt to specify an all-embracing set of parameters to cover all possible implementation models. A minimal set includes:

a) a minimum service rate profile which allows rate guarantees for each traffic stream as required by EF and AF without specifying the details of how excess bandwidth between these traffic streams is shared. Additional parameters to control this behavior should be made available, but are dependent on the particular scheduling algorithm implemented.

b) a service priority, used only after the minimum rate profiles of all inputs have been satisfied, to decide how to allocate any remaining bandwidth.

c) a maximum service rate profile, for use only with a non-work-conserving service discipline.

Any one of these profiles is composed, for the purposes of this model, of both a rate (in suitable units of bits, bytes or larger chunks in some unit of time) and a burst size, as discussed further in Appendix A.

By way of example, for an implementation of the EF PHB using a strict priority scheduling algorithm that assumes that the aggregate EF rate has been appropriately bounded by upstream policing to avoid starvation of other BAs, the service rate profiles are not used: the minimum service rate profile would be defaulted to zero and the maximum service rate profile would effectively be the "line rate". Such an implementation, with multiple priority classes, could also be used for the Diffserv class selectors [DSFIELD].

Alternatively, setting the service priority values for each input to the scheduler to the same value enables the scheduler to satisfy the minimum service rates for each input, so long as the sum of all minimum service rates is less than or equal to the line rate.

For example, a non-work-conserving scheduler, allocating spare bandwidth equally between all its inputs, might be represented using the following parameters:

```
Scheduler1:
Type:           Scheduler2Input

Input1:
MaxRateProfile: Profile1
MinRateProfile: Profile2
Priority:       none

Input2:
MaxRateProfile: Profile3
MinRateProfile: Profile4
Priority:       none
```

A work-conserving scheduler might be represented using the following parameters:

```
Scheduler2:
Type:           Scheduler3Input
Input1:
MaxRateProfile: WorkConserving
MinRateProfile: Profile5
Priority:       1

Input2:
MaxRateProfile: WorkConserving
MinRateProfile: Profile6
Priority:       2

Input3:
MaxRateProfile: WorkConserving
MinRateProfile: none
Priority:       3
```

7.1.3. Algorithmic Dropper

An Algorithmic Dropper is an element which selectively discards packets that arrive at its input, based on a discarding algorithm. It has one data input and one output. In this model (but not necessarily in a real implementation), a packet enters the dropper at its input and either its buffer is returned to a free buffer pool or the packet exits the dropper at the output.

Alternatively, an Algorithmic Dropper can be thought of as invoking operations on a FIFO Queue which selectively remove a packet and return its buffer to the free buffer pool based on a discarding algorithm. In this case, the operation could be modeled as being a side-effect on the FIFO upon which it operated, rather than as having a discrete input and output. This treatment is equivalent and we choose the one described in the previous paragraph for this model.

One of the primary characteristics of an Algorithmic Dropper is the choice of which packet (if any) is to be dropped: for the purposes of this model, we restrict the packet selection choices to one of the following and we indicate the choice by the relative positions of Algorithmic Dropper and FIFO Queue elements in the model:

a) selection of a packet that is about to be added to the tail of a queue (a "Tail Dropper"): the output of the Algorithmic Dropper element is connected to the input of the relevant FIFO Queue element.

b) a packet that is currently at the head of a queue (a "Head Dropper"): the output of the FIFO Queue element is connected to the input of the Algorithmic Dropper element.

Other packet selection methods could be added to this model in the form of a different type of datapath element.

The Algorithmic Dropper is modeled as having a single input. It is possible that packets which were classified differently by a Classifier in this TCB will end up passing through the same dropper. The dropper's algorithm may need to apply different calculations based on characteristics of the incoming packet (e.g., its DSCP). So there is a need, in implementations of this model, to be able to relate information about which classifier element was matched by a packet from a Classifier to an Algorithmic Dropper. In the rare cases where this is required, the chosen model is to insert another Classifier element at this point in the flow and for it to feed into multiple Algorithmic Dropper elements, each one implementing a drop calculation that is independent of any classification keys of the packet: this will likely require the creation of a new TCB to contain the Classifier and the Algorithmic Dropper elements.

> NOTE: There are many other formulations of a model that could represent this linkage that are different from the one described above: one formulation would have been to have a pointer from one of the drop probability calculation algorithms inside the dropper to the original Classifier element that selects this algorithm. Another way would have been to have multiple "inputs" to the Algorithmic Dropper element fed from the preceding elements, leading eventually back to the Classifier elements that matched the packet. Yet another formulation might have been for the Classifier to (logically) include some sort of "classification identifier" along with the packet along its path, for use by any subsequent element. And yet another could have been to include a classifier inside the dropper, in order for it to pick out the drop algorithm to be applied. These other approaches could be used by implementations but were deemed to be less clear than the approach taken here.

An Algorithmic Dropper, an example of which is illustrated in Figure 5, has one or more triggers that cause it to make a decision whether or not to drop one (or possibly more than one) packet. A trigger may be internal (the arrival of a packet at the input to the dropper) or it may be external (resulting from one or more state changes at another element, such as a FIFO Queue depth crossing a threshold or a scheduling event). It is likely that an instantaneous FIFO depth will need to be smoothed over some averaging interval before being used as a useful trigger. Some dropping algorithms may require several trigger inputs feeding back from events elsewhere in the system (e.g., depth-smoothing functions that calculate averages over more than one time interval).

```
              +------------------+      +-----------+
              | +-------+        |  n   |smoothing  |
              | |trigger|<----------/---|function(s)|
              | |calc.  |        |      |(optional) |
              | +-------+        |      +-----------+
              |     |            |          ^
              |     v            |          |Depth
     Input    | +-------+ no     |      ------------+   to Scheduler
     ---------->|discard|--------------->    |x|x|x|x|------->
              | |   ?   |        |      ------------+
              | +-------+        |           FIFO
              |    |yes          |
              |  | | |           |
              |  | v | count +   |
              |  +---+ bit-bucket|
              +------------------+
              Algorithmic
              Dropper
```

Figure 5. Example of Algorithmic Dropper from Tail of a Queue

A trigger may be a boolean combination of events (e.g., a FIFO depth exceeding a threshold OR a buffer pool depth falling below a threshold). It takes as its input some set of dynamic parameters (e.g., smoothed or instantaneous FIFO depth), and some set of static parameters (e.g., thresholds), and possibly other parameters associated with the packet. It may also have internal state (e.g., history of its past actions). Note that, although an Algorithmic Dropper may require knowledge of data fields in a packet, as discovered by a Classifier in the same TCB, it may not modify the packet (i.e., it is not a marker).

The result of the trigger calculation is that the dropping algorithm makes a decision on whether to forward or to discard a packet. The discarding function is likely to keep counters regarding the discarded packets (there is no appropriate place here to include a Counter Action element).

The example in Figure 5 also shows a FIFO Queue element from whose tail the dropping is to take place and whose depth characteristics are used by this Algorithmic Dropper. It also shows where a depth-smoothing function might be included: smoothing functions are outside the scope of this document and are not modeled explicitly here, we merely indicate where they might be added.

RED, RED-on-In-and-Out (RIO) and Drop-on-threshold are examples of dropping algorithms. Tail-dropping and head-dropping are effected by the location of the Algorithmic Dropper element relative to the FIFO

Queue element. As an example, a dropper using a RIO algorithm might be represented using 2 Algorithmic Droppers with the following parameters:

```
AlgorithmicDropper1: (for in-profile traffic)
Type:                   AlgorithmicDropper
Discipline:             RED
Trigger:                Internal
Output:                 Fifo1
MinThresh:              Fifo1.Depth > 20 kbyte
MaxThresh:              Fifo1.Depth > 30 kbyte
SampleWeight            .002
MaxDropProb             1%

AlgorithmicDropper2: (for out-of-profile traffic)
Type:                   AlgorithmicDropper
Discipline:             RED
Trigger:                Internal
Output:                 Fifo1
MinThresh:              Fifo1.Depth > 10 kbyte
MaxThresh:              Fifo1.Depth > 20 kbyte
SampleWeight            .002
MaxDropProb             2%
```

Another form of Algorithmic Dropper, a threshold-dropper, might be represented using the following parameters:

```
AlgorithmicDropper3:
Type:                   AlgorithmicDropper
Discipline:             Drop-on-threshold
Trigger:                Fifo2.Depth > 20 kbyte
Output:                 Fifo1
```

7.2. Sharing load among traffic streams using queuing

Queues are used, in Differentiated Services, for a number of purposes. In essence, they are simply places to store traffic until it is transmitted. However, when several queues are used together in a queuing system, they can also achieve effects beyond that for given traffic streams. They can be used to limit variation in delay or impose a maximum rate (shaping), to permit several streams to share a link in a semi-predictable fashion (load sharing), or to move variation in delay from some streams to other streams.

Traffic shaping is often used to condition traffic, such that packets arriving in a burst will be "smoothed" and deemed conforming by subsequent downstream meters in this or other nodes. In [DSARCH] a shaper is described as a queuing element controlled by a meter which defines its temporal profile. However, this representation of a shaper differs substantially from typical shaper implementations.

In the model described here, a shaper is realized by using a non-work-conserving Scheduler. Some implementations may elect to have queues whose sole purpose is shaping, while others may integrate the shaping function with other buffering, discarding, and scheduling associated with access to a resource. Shapers operate by delaying the departure of packets that would be deemed non-conforming by a meter configured to the shaper's maximum service rate profile. The packet is scheduled to depart no sooner than such time that it would become conforming.

7.2.1. Load Sharing

Load sharing is the traditional use of queues and was theoretically explored by Floyd & Jacobson [FJ95], although it has been in use in communications systems since the 1970's.

[DSARCH] discusses load sharing as dividing an interface among traffic classes predictably, or applying a minimum rate to each of a set of traffic classes, which might be measured as an absolute lower bound on the rate a traffic stream achieves or a fraction of the rate an interface offers. It is generally implemented as some form of weighted queuing algorithm among a set of FIFO queues i.e., a WFQ scheme. This has interesting side-effects.

A key effect sought is to ensure that the mean rate the traffic in a stream experiences is never lower than some threshold when there is at least that much traffic to send. When there is less traffic than this, the queue tends to be starved of traffic, meaning that the queuing system will not delay its traffic by very much. When there is significantly more traffic and the queue starts filling, packets in this class will be delayed significantly more than traffic in other classes that are under-using their available capacity. This form of queuing system therefore tends to move delay and variation in delay from under-used classes of traffic to heavier users, as well as managing the rates of the traffic streams.

A side-effect of a WRR or WFQ implementation is that between any two packets in a given traffic class, the scheduler may emit one or more packets from each of the other classes in the queuing system. In cases where average behavior is in view, this is perfectly acceptable. In cases where traffic is very intolerant of jitter and there are a number of competing classes, this may have undesirable consequences.

7.2.2. Traffic Priority

Traffic Prioritization is a special case of load sharing, wherein a certain traffic class is deemed so jitter-intolerant that if it has traffic present, that traffic must be sent at the earliest possible time. By extension, several priorities might be defined, such that traffic in each of several classes is given preferential service over any traffic of a lower class. It is the obvious implementation of IP Precedence as described in [RFC 791], of 802.1p traffic classes [802.1D], and other similar technologies.

Priority is often abused in real networks; people tend to think that traffic which has a high business priority deserves this treatment and talk more about the business imperatives than the actual application requirements. This can have severe consequences; networks have been configured which placed business-critical traffic at a higher priority than routing-protocol traffic, resulting in collapse of the network’s management or control systems. However, it may have a legitimate use for services based on an Expedited Forwarding (EF) PHB, where it is absolutely sure, thanks to policing at all possible traffic entry points, that a traffic stream does not abuse its rate and that the application is indeed jitter-intolerant enough to merit this type of handling. Note that, even in cases with well-policed ingress points, there is still the possibility of unexpected traffic loops within an un-policed core part of the network causing such collapse.

8. Traffic Conditioning Blocks (TCBs)

The Classifier, Meter, Action, Algorithmic Dropper, Queue and Scheduler functional datapath elements described above can be combined into Traffic Conditioning Blocks (TCBs). A TCB is an abstraction of a set of functional datapath elements that may be used to facilitate the definition of specific traffic conditioning functionality (e.g., it might be likened to a template which can be replicated multiple times for different traffic streams or different customers). It has no likely physical representation in the implementation of the data path: it is invented purely as an abstraction for use by management tools.

This model describes the configuration and management of a Diffserv interface in terms of a TCB that contains, by definition, zero or more Classifier, Meter, Action, Algorithmic Dropper, Queue and Scheduler elements. These elements are arranged arbitrarily according to the policy being expressed, but always in the order here. Traffic may be classified; classified traffic may be metered; each stream of traffic identified by a combination of classifiers and meters may have some set of actions performed on it, followed by drop algorithms; packets of the traffic stream may ultimately be stored into a queue and then be scheduled out to the next TCB or physical interface. It is permissible to omit elements or include null elements of any type, or to concatenate multiple functional datapath elements of the same type.

When the Diffserv treatment for a given packet needs to have such building blocks repeated, this is performed by cascading multiple TCBs: an output of one TCB may drive the input of a succeeding one. For example, consider the case where traffic of a set of classes is shaped to a set of rates, but the total output rate of the group of classes must also be limited to a rate. One might imagine a set of network news feeds, each with a certain maximum rate, and a policy that their aggregate may not exceed some figure. This may be simply accomplished by cascading two TCBs. The first classifies the traffic into its separate feeds and queues each feed separately. The feeds (or a subset of them) are now fed into a second TCB, which places all input (these news feeds) into a single queue with a certain maximum rate. In implementation, one could imagine this as the several literal queues, a CBQ or WFQ system with an appropriate (and complex) weighting scheme, or a number of other approaches. But they would have the same externally measurable effect on the traffic as if they had been literally implemented with separate TCBs.

8.1. TCB

A generalized TCB might consist of the following stages:

- Classification stage
- Metering stage
- Action stage (involving Markers, Absolute Droppers, Counters, and Multiplexors)
- Queuing stage (involving Algorithmic Droppers, Queues, and Schedulers)

where each stage may consist of a set of parallel datapaths consisting of pipelined elements.

A Classifier or a Meter is typically a 1:N element, an Action, Algorithmic Dropper, or Queue is typically a 1:1 element and a Scheduler is a N:1 element. A complete TCB should, however, result in a 1:1 or 1:N abstract element. Note that the fan-in or fan-out of an element is not an important defining characteristic of this taxonomy.

8.1.1. Building blocks for Queuing

Some particular rules are applied to the ordering of elements within a Queuing stage within a TCB: elements of the same type may appear more than once, either in parallel or in series. Typically, a queuing stage will have relatively many elements in parallel and few in series. Iteration and recursion are not supported constructs (the elements are arranged in an acyclic graph). The following inter-connections of elements are allowed:

- The input of a Queue may be the input of the queuing block, or it may be connected to the output of an Algorithmic Dropper, or to an output of a Scheduler.

- Each input of a Scheduler may be connected to the output of a Queue, to the output of an Algorithmic Dropper, or to the output of another Scheduler.

- The input of an Algorithmic Dropper may be the first element of the queuing stage, the output of another Algorithmic Dropper, or it may be connected to the output of a Queue (to indicate head-dropping).

- The output of the queuing block may be the output of a Queue, an Algorithmic Dropper, or a Scheduler.

Note, in particular, that Schedulers may operate in series such so that a packet at the head of a Queue feeding the concatenated Schedulers is serviced only after all of the scheduling criteria are met. For example, a Queue which carries EF traffic streams may be served first by a non-work-conserving Scheduler to shape the stream to a maximum rate, then by a work-conserving Scheduler to mix EF traffic streams with other traffic streams. Alternatively, there might be a Queue and/or a dropper between the two Schedulers.

Note also that some non-sensical scenarios (e.g., a Queue preceding an Algorithmic Dropper, directly feeding into another Queue), are prohibited.

8.2. An Example TCB

A SLS is presumed to have been negotiated between the customer and the provider which specifies the handling of the customer’s traffic, as defined by a TCS) by the provider’s network. The agreement might be of the following form:

DSCP	PHB	Profile	Treatment
001001	EF	Profile4	Discard non-conforming.
001100	AF11	Profile5	Shape to profile, tail-drop when full.
001101	AF21	Profile3	Re-mark non-conforming to DSCP 001000, tail-drop when full.
other	BE	none	Apply RED-like dropping.

This SLS specifies that the customer may submit packets marked for DSCP 001001 which will get EF treatment so long as they remain conforming to Profile4, which will be discarded if they exceed this profile. The discarded packets are counted in this example, perhaps for use by the provider’s sales department in convincing the customer to buy a larger SLS. Packets marked for DSCP 001100 will be shaped to Profile5 before forwarding. Packets marked for DSCP 001101 will be metered to Profile3 with non-conforming packets "downgraded" by being re-marked with a DSCP of 001000. It is implicit in this agreement that conforming packets are given the PHB originally indicated by the packets’ DSCP field.

Figures 6 and 7 illustrates a TCB that might be used to handle this SLS at an ingress interface at the customer/provider boundary.

The Classification stage of this example consists of a single BA classifier. The BA classifier is used to separate traffic based on the Diffserv service level requested by the customer (as indicated by the DSCP in each submitted packet’s IP header). We illustrate three DSCP filter values: A, B, and C. The ’X’ in the BA classifier is a wildcard filter that matches every packet not otherwise matched.

The path for DSCP 001100 proceeds directly to Dropper1 whilst the paths for DSCP 001001 and 001101 include a metering stage. All other traffic is passed directly on to Dropper3. There is a separate meter for each set of packets corresponding to classifier outputs A and C. Each meter uses a specific profile, as specified in the TCS, for the corresponding Diffserv service level. The meters in this example each indicate one of two conformance levels: conforming or non-conforming.

Following the Metering stage is an Action stage in some of the branches. Packets submitted for DSCP 001001 (Classifier output A) that are deemed non-conforming by Meter1 are counted and discarded while packets that are conforming are passed on to Queue1. Packets submitted for DSCP 001101 (Classifier output C) that are deemed non-conforming by Meter2 are re-marked and then both conforming and non-conforming packets are multiplexed together before being passed on to Dropper2/Queue3.

The Algorithmic Dropping, Queuing and Scheduling stages are realized as follows, illustrated in figure 7. Note that the figure does not show any of the implicit control linkages between elements that allow e.g., an Algorithmic Dropper to sense the current state of a succeeding Queue.

```
                         +-----+
                         |    A|---------------------------> to Queue1
                      +->|     |
                      |  |    B|--+  +-----+    +-----+
                      |  +-----+  |  |     |    |     |
                      |  Meter1   +->|     |--->|     |
                      |              |     |    |     |
                      |              +-----+    +-----+
                      |              Counter1   Absolute
submitted +-----+     |                         Dropper1
traffic   |    A|-----+
--------->|    B|--------------------------------------> to AlgDropper1
          |    C|-----+
          |    X|--+  |
          +-----+  |  |  +-----+                +-----+
        Classifier1|  |  |    A|--------------->|A    |
           (BA)    |  +->|     |                |     |--> to AlgDrop2
                   |     |    B|--+  +-----+ +->|B    |
                   |     +-----+  |  |     | |  +-----+
                   |     Meter2   +->|     |-+    Mux1
                   |                 |     |
                   |                 +-----+
                   |                 Marker1
                   +-----------------------------------> to AlgDropper3
```

Figure 6: An Example Traffic Conditioning Block (Part 1)

Conforming DSCP 001001 packets from Meter1 are passed directly to Queue1: there is no way, with configuration of the following Scheduler to match the metering, for these packets to overflow the depth of Queue1, so there is no requirement for dropping at this point. Packets marked for DSCP 001100 must be passed through a tail-dropper, AlgDropper1, which serves to limit the depth of the following queue, Queue2: packets that arrive to a full queue will be discarded. This is likely to be an error case: the customer is obviously not sticking to its agreed profile. Similarly, all packets from the original DSCP 001101 stream (some may have been re-marked by this stage) are passed to AlgDropper2 and Queue3. Packets marked for all other DSCPs are passed to AlgDropper3 which is a RED-like Algorithmic Dropper: based on feedback of the current depth of Queue4, this dropper is supposed to discard enough packets from its input stream to keep the queue depth under control.

These four Queue elements are then serviced by a Scheduler element Scheduler1: this must be configured to give each of its inputs an appropriate priority and/or bandwidth share. Inputs A and C are given guarantees of bandwidth, as appropriate for the contracted profiles. Input B is given a limit on the bandwidth it can use (i.e., a non-work-conserving discipline) in order to achieve the desired shaping of this stream. Input D is given no limits or guarantees but a lower priority than the other queues, appropriate for its best-effort status. Traffic then exits the Scheduler in a single orderly stream.

The interconnections of the TCB elements illustrated in Figures 6 and 7 can be represented textually as follows:

```
      TCB1:

      Classifier1:
      FilterA:             Meter1
      FilterB:             Dropper1
      FilterC:             Meter2
      Default:             Dropper3
```

```
      from Meter1                     +-----+
      ------------------------------->|     |----+
                                      |     |    |
                                      +-----+    |
                                      Queue1     |
                                                 |  +-----+
      from Classifier1 +-----+        +-----+    +->|A    |
      ---------------->|     |------->|     |------>|B    |------->
                       |     |        |     |  +--->|C    |  exiting
                       +-----+        +-----+  | +->|D    |  traffic
                       AlgDropper1    Queue2   | |  +-----+
                                               | |  Scheduler1
      from Mux1        +-----+        +-----+  | |
      ---------------->|     |------->|     |--+ |
                       |     |        |     |    |
                       +-----+        +-----+    |
                       AlgDropper2    Queue3     |
                                                 |
      from Classifier1 +-----+        +-----+    |
      ---------------->|     |------->|     |----+
                       |     |        |     |
                       +-----+        +-----+
                       AlgDropper3    Queue4
```

Figure 7: An Example Traffic Conditioning Block (Part 2)

```
        Meter1:
        Type:                 AverageRate
        Profile:              Profile4
        ConformingOutput:     Queue1
        NonConformingOutput:  Counter1

        Counter1:
        Output:               AbsoluteDropper1

        Meter2:
        Type:                 AverageRate
        Profile:              Profile3
        ConformingOutput:     Mux1.InputA
        NonConformingOutput:  Marker1

        Marker1:
        Type:                 DSCPMarker
        Mark:                 001000
        Output:               Mux1.InputB

        Mux1:
        Output:               Dropper2

        AlgDropper1:
        Type:                 AlgorithmicDropper
        Discipline:           Drop-on-threshold
        Trigger:              Queue2.Depth > 10kbyte
        Output:               Queue2

        AlgDropper2:
        Type:                 AlgorithmicDropper
        Discipline:           Drop-on-threshold
        Trigger:              Queue3.Depth > 20kbyte
        Output:               Queue3

        AlgDropper3:
        Type:                 AlgorithmicDropper
        Discipline:           RED93
        Trigger:              Internal
        Output:               Queue3
        MinThresh:            Queue3.Depth > 20 kbyte
        MaxThresh:            Queue3.Depth > 40 kbyte
           <other RED parms too>
        Queue1:
        Type:                    FIFO
        Output:                  Scheduler1.InputA

        Queue2:
        Type:                    FIFO
        Output:                  Scheduler1.InputB

        Queue3:
        Type:                    FIFO
        Output:                  Scheduler1.InputC

        Queue4:
        Type:                    FIFO
        Output:                  Scheduler1.InputD

        Scheduler1:
        Type:                    Scheduler4Input
        InputA:
        MaxRateProfile:          none
        MinRateProfile:          Profile4
        Priority:                20
        InputB:
        MaxRateProfile:          Profile5
        MinRateProfile:          none
        Priority:                40
        InputC:
        MaxRateProfile:          none
        MinRateProfile:          Profile3
        Priority:                20
        InputD:
        MaxRateProfile:          none
        MinRateProfile:          none
        Priority:                10
```

8.3. An Example TCB to Support Multiple Customers

The TCB described above can be installed on an ingress interface to implement a provider/customer TCS if the interface is dedicated to the customer. However, if a single interface is shared between multiple customers, then the TCB above will not suffice, since it does not differentiate among traffic from different customers. Its classification stage uses only BA classifiers.

The configuration is readily modified to support the case of multiple customers per interface, as follows. First, a TCB is defined for each customer to reflect the TCS with that customer: TCB1, defined above is the TCB for customer 1. Similar elements are created for

TCB2 and for TCB3 which reflect the agreements with customers 2 and 3 respectively. These 3 TCBs may or may not contain similar elements and parameters.

Finally, a classifier is added to the front end to separate the traffic from the three different customers. This forms a new TCB, TCB4, which is illustrated in Figure 8.

A representation of this multi-customer TCB might be:

```
TCB4:

Classifier4:
Filter1:     to TCB1
Filter2:     to TCB2
Filter3:     to TCB3
No Match:    AbsoluteDropper4

AbsoluteDropper4:
Type:                AbsoluteDropper

TCB1:
(as defined above)

TCB2:
(similar to TCB1, perhaps with different
 elements or numeric parameters)

TCB3:
(similar to TCB1, perhaps with different
 elements or numeric parameters)
```

and the filters, based on each customer’s source MAC address, could be defined as follows:

```
Filter1:

submitted +-----+
traffic   |    A|--------> TCB1
--------->|    B|--------> TCB2
          |    C|--------> TCB3
          |    X|------+   +-----+
          +-----+      +-->|     |
          Classifier4      +-----+
                           AbsoluteDrop4
```

Figure 8: An Example of a Multi-Customer TCB

```
      Type:         MacAddress
      SrcValue:     01-02-03-04-05-06 (source MAC address of customer 1)
      SrcMask:      FF-FF-FF-FF-FF-FF
      DestValue:    00-00-00-00-00-00
      DestMask:     00-00-00-00-00-00

      Filter2:
      (similar to Filter1 but with customer 2's source MAC address as
       SrcValue)

      Filter3:
      (similar to Filter1 but with customer 3's source MAC address as
       SrcValue)
```

In this example, Classifier4 separates traffic submitted from different customers based on the source MAC address in submitted packets. Those packets with recognized source MAC addresses are passed to the TCB implementing the TCS with the corresponding customer. Those packets with unrecognized source MAC addresses are passed to a dropper.

TCB4 has a Classifier stage and an Action element stage performing dropping of all unmatched traffic.

8.4. TCBs Supporting Microflow-based Services

The TCB illustrated above describes a configuration that might be suitable for enforcing a SLS at a router's ingress. It assumes that the customer marks its own traffic for the appropriate service level. It then limits the rate of aggregate traffic submitted at each service level, thereby protecting the resources of the Diffserv network. It does not provide any isolation between the customer's individual microflows.

A more complex example might be a TCB configuration that offers additional functionality to the customer. It recognizes individual customer microflows and marks each one independently. It also isolates the customer's individual microflows from each other in order to prevent a single microflow from seizing an unfair share of the resources available to the customer at a certain service level. This is illustrated in Figure 9.

Suppose that the customer has an SLS which specifies 2 service levels, to be identified to the provider by DSCP A and DSCP B. Traffic is first directed to a MF classifier which classifies traffic based on miscellaneous classification criteria, to a granularity sufficient to identify individual customer microflows. Each microflow can then be marked for a specific DSCP The metering elements limit the contribution of each of the customer’s microflows to the service level for which it was marked. Packets exceeding the allowable limit for the microflow are dropped.

```
                     +-----+   +-----+
    Classifier1      |     |   |     |---------------+
        (MF)      +->|     |-->|     |     +-----+   |
      +-----+     |  |     |   |     |---->|     |   |
      |    A|------  +-----+   +-----+     +-----+   |
   -->|    B|-----+  Marker1   Meter1      Absolute  |
      |    C|---+ |                        Dropper1  |   +-----+
      |    X|-+ | |  +-----+   +-----+               +-->|A    |
      +-----+ | | |  |     |   |     |------------------>|B    |--->
              | | +->|     |-->|     |     +-----+   +-->|C    | to TCB2
              | |    |     |   |     |---->|     |   |   +-----+
              | |    +-----+   +-----+     +-----+   |    Mux1
              | |    Marker2   Meter2      Absolute  |
              | |                          Dropper2  |
              | |    +-----+   +-----+               |
              | |    |     |   |     |---------------+
              | |--->|     |-->|     |     +-----+
              |      |     |   |     |---->|     |
              |      +-----+   +-----+     +-----+
              |      Marker3   Meter3      Absolute
              |                            Dropper3
              V etc.
```

Figure 9: An Example of a Marking and Traffic Isolation TCB

This TCB could be formally specified as follows:

```
TCB1:
Classifier1: (MF)
FilterA:             Marker1
FilterB:             Marker2
FilterC:             Marker3
etc.

Marker1:
Output:              Meter1

Marker2:
Output:              Meter2

Marker3:
Output:              Meter3
```

```
      Meter1:
      ConformingOutput:    Mux1.InputA
      NonConformingOutput: AbsoluteDropper1

      Meter2:
      ConformingOutput:    Mux1.InputB
      NonConformingOutput: AbsoluteDropper2

      Meter3:
      ConformingOutput:    Mux1.InputC
      NonConformingOutput: AbsoluteDropper3

      etc.

      Mux1:
      Output:              to TCB2
```

Note that the detailed traffic element declarations are not shown here. Traffic is either dropped by TCB1 or emerges marked for one of two DSCPs. This traffic is then passed to TCB2 which is illustrated in Figure 10.

TCB2 could then be specified as follows:

```
      Classifier2: (BA)
      FilterA:               Meter5
      FilterB:               Meter6
```

```
                     +-----+
                     |     |---------------> to Queue1
                  +->|     |     +-----+
        +-----+   |  |     |---->|     |
        |    A|---+  +-----+     +-----+
      ->|     |       Meter5     AbsoluteDropper4
        |    B|---+  +-----+
        +-----+   |  |     |---------------> to Queue2
      Classifier2 +->|     |     +-----+
         (BA)        |     |---->|     |
                     +-----+     +-----+
                      Meter6     AbsoluteDropper5
```

Figure 10: Additional Example: TCB2

```
      Meter5:
      ConformingOutput:      Queue1
      NonConformingOutput:   AbsoluteDropper4
      Meter6:
      ConformingOutput:      Queue2
      NonConformingOutput:   AbsoluteDropper5
```

8.5. Cascaded TCBs

Nothing in this model prevents more complex scenarios in which one microflow TCB precedes another (e.g., for TCBs implementing separate TCSs for the source and for a set of destinations).

9. Security Considerations

Security vulnerabilities of Diffserv network operation are discussed in [DSARCH]. This document describes an abstract functional model of Diffserv router elements. Certain denial-of-service attacks such as those resulting from resource starvation may be mitigated by appropriate configuration of these router elements; for example, by rate limiting certain traffic streams or by authenticating traffic marked for higher quality-of-service.

There may be theft-of-service scenarios where a malicious host can exploit a loose token bucket policer to obtain slightly better QoS than that committed in the TCS.

10. Acknowledgments

Concepts, terminology, and text have been borrowed liberally from [POLTERM], as well as from other IETF work on MIBs and policy-management. We wish to thank the authors of some of those documents: Fred Baker, Michael Fine, Keith McCloghrie, John Seligson, Kwok Chan, Scott Hahn, and Andrea Westerinen for their contributions.

This document has benefited from the comments and suggestions of several participants of the Diffserv working group, particularly Shahram Davari, John Strassner, and Walter Weiss. This document could never have reached this level of rough consensus without the relentless pressure of the co-chairs Brian Carpenter and Kathie Nichols, for which the authors are grateful.

11. References

[AF-PHB] Heinanen, J., Baker, F., Weiss, W. and J. Wroclawski, "Assured Forwarding PHB Group", RFC 2597, June 1999.

[DSARCH] Carlson, M., Weiss, W., Blake, S., Wang, Z., Black, D. and E. Davies, "An Architecture for Differentiated Services", RFC 2475, December 1998.

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

[DSMIB] Baker, F., Smith, A., and K. Chan, "Management Information Base for the Differentiated Services Architecture", RFC 3289, May 2002.

[E2E] Bernet, Y., Yavatkar, R., Ford, P., Baker, F., Zhang, L., Speer, M., Nichols, K., Braden, R., Davie, B., Wroclawski, J. and E. Felstaine, "A Framework for Integrated Services Operation over Diffserv Networks", RFC 2998, November 2000.

[EF-PHB] Davie, B., Charny, A., Bennett, J.C.R., Benson, K., Le Boudec, J.Y., Courtney, W., Davari, S., Firoiu, V. and D. Stiliadis, "An Expedited Forwarding PHB (Per-Hop Behavior)", RFC 3246, March 2002.

[FJ95] Floyd, S. and V. Jacobson, "Link Sharing and Resource Management Models for Packet Networks", IEEE/ACM Transactions on Networking, Vol. 3 No. 4, August 1995l.

[INTSERV] Braden, R., Clark, D. and S. Shenker, "Integrated Services in the Internet Architecture: an Overview", RFC 1633, June 1994.

[NEWTERMS] Grossman, D., "New Terminology and Clarifications for Diffserv", RFC 3260, April, 2002

[PDBDEF] K. Nichols and B. Carpenter, "Definition of Differentiated Services Per Domain Behaviors and Rules for Their Specification", RFC 3086, April 2001.

[POLTERM] Westerinen, A., Schnizlein, J., Strassner, J., Scherling, M., Quinn, B., Herzog, S., Huynh, A., Carlson, M., Perry, J. and S. Waldbusser, "Policy Terminology", RFC 3198, November 2001.

[QOSDEVMOD] Strassner, J., Westerinen, A. and B. Moore, "Information Model for Describing Network Device QoS Mechanisms", Work in Progress.

[QUEUEMGMT] Braden, R., Clark, D., Crowcroft, J., Davie, B., Deering, S., Estrin, D., Floyd, S., Jacobson, V., Minshall, C., Partridge, C., Peterson, L., Ramakrishnan, K., Shenker, S., Wroclawski, J. and L. Zhang, "Recommendations on Queue Management and Congestion Avoidance in the Internet", RFC 2309, April 1998.

[SRTCM] Heinanen, J. and R. Guerin, "A Single Rate Three Color Marker", RFC 2697, September 1999.

[TRTCM] Heinanen, J. and R. Guerin, "A Two Rate Three Color Marker", RFC 2698, September 1999.

[VIC] McCanne, S. and Jacobson, V., "vic: A Flexible Framework for Packet Video", ACM Multimedia ’95, November 1995, San Francisco, CA, pp. 511-522. <ftp://ftp.ee.lbl.gov/papers/vic-mm95.ps.Z>

[802.1D] "Information technology - Telecommunications and information exchange between systems - Local and metropolitan area networks - Common specifications - Part 3: Media Access Control (MAC) Bridges: Revision. This is a revision of ISO/IEC 10038: 1993, 802.1j-1992 and 802.6k-1992. It incorporates P802.11c, P802.1p and P802.12e.", ISO/IEC 15802-3: 1998.

Appendix A. Discussion of Token Buckets and Leaky Buckets

"Leaky bucket" and/or "Token Bucket" models are used to describe rate control in several architectures, including Frame Relay, ATM, Integrated Services and Differentiated Services. Both of these models are, by definition, theoretical relationships between some defined burst size, B, a rate, R, and a time interval, t:

$$R = B/t$$

Thus, a token bucket or leaky bucket might specify an information rate of 1.2 Mbps with a burst size of 1500 bytes. In this case, the token rate is 1,200,000 bits per second, the token burst is 12,000 bits and the token interval is 10 milliseconds. The specification says that conforming traffic will, in the worst case, come in 100 bursts per second of 1500 bytes each and at an average rate not exceeding 1.2 Mbps.

A.1 Leaky Buckets

A leaky bucket algorithm is primarily used for shaping traffic as it leaves an interface onto the network (handled under Queues and Schedulers in this model). Traffic theoretically departs from an interface at a rate of one bit every so many time units (in the example, one bit every 0.83 microseconds) but, in fact, departs in multi-bit units (packets) at a rate approximating the theoretical, as measured over a longer interval. In the example, it might send one 1500 byte packet every 10 ms or perhaps one 500 byte packet every 3.3 ms. It is also possible to build multi-rate leaky buckets in which traffic departs from the interface at varying rates depending on recent activity or inactivity.

Implementations generally seek as constant a transmission rate as achievable. In theory, a 10 Mbps shaped transmission stream from an algorithmic implementation and a stream which is running at 10 Mbps because its bottleneck link has been a 10 Mbps Ethernet link should be indistinguishable. Depending on configuration, the approximation to theoretical smoothness may vary by moving as much as an MTU from one token interval to another. Traffic may also be jostled by other traffic competing for the same transmission resources.

A.2 Token Buckets

A token bucket, on the other hand, measures the arrival rate of traffic from another device. This traffic may originally have been shaped using a leaky bucket shaper or its equivalent. The token bucket determines whether the traffic (still) conforms to the specification. Multi-rate token buckets (e.g., token buckets with both a peak rate and a mean rate, and sometimes more) are commonly used, such as those described in [SRTCM] and [TRTCM]. In this case, absolute smoothness is not expected, but conformance to one or more of the specified rates is.

Simplistically, a data stream is said to conform to a simple token bucket parameterized by a {R, B} if the system receives in any time interval, t, at most, an amount of data not exceeding (R * t) + B.

For a multi-rate token bucket case, the data stream is said to conform if, for each of the rates, the stream conforms to the token-bucket profile appropriate for traffic of that class. For example, received traffic that arrives pre-classified as one of the "excess" rates (e.g., AF12 or AF13 traffic for a device implementing the AF1x PHB) is only compared to the relevant "excess" token bucket profile.

A.3 Some Consequences

The fact that Internet Protocol data is organized into variable length packets introduces some uncertainty in the conformance decision made by any downstream Meter that is attempting to determine conformance to a traffic profile that is theoretically designed for fixed-length units of data.

When used as a leaky bucket shaper, the above definition interacts with clock granularity in ways one might not expect. A leaky bucket releases a packet only when all of its bits would have been allowed: it does not borrow from future capacity. If the clock is very fine grain, on the order of the bit rate or faster, this is not an issue. But if the clock is relatively slow (and millisecond or multi-millisecond clocks are not unusual in networking equipment), this can introduce jitter to the shaped stream.

This leaves an implementor of a token bucket Meter with a dilemma. When the number of bandwidth tokens, b, left in the token bucket is positive but less than the size of the packet being operated on, L, one of three actions can be performed:

(1) The whole size of the packet can be subtracted from the bucket, leaving it negative, remembering that, when new tokens are next added to the bucket, the new token allocation, B, must be added to b rather than simply setting the bucket to "full". This option potentially puts more than the desired burst size of data into this token bucket interval and correspondingly less into the next. It does, however, keep the average amount accepted per token bucket interval equal to the token burst. This approach accepts traffic if any one bit in the packet would have been

accepted and borrows up to one MTU of capacity from one or more subsequent intervals when necessary. Such a token bucket meter implementation is said to offer "loose" conformance to the token bucket.

(2) Alternatively, the packet can be rejected and the amount of tokens in the bucket left unchanged (and maybe an attempt could be made to accept the packet under another threshold in another bucket), remembering that, when new tokens are next added to the bucket, the new token allocation, B, must be added to b rather than simply setting the bucket to "full". This potentially puts less than the permissible burst size of data into this token bucket interval and correspondingly more into the next. Like the first option, it keeps the average amount accepted per token bucket interval equal to the token burst. This approach accepts traffic only if every bit in the packet would have been accepted and borrows up to one MTU of capacity from one or more previous intervals when necessary. Such a token bucket meter implementation is said to offer "strict" (or perhaps "stricter") conformance to the token bucket. This option is consistent with [SRTCM] and [TRTCM] and is often used in ATM and frame-relay implementations.

(3) The TB variable can be set to zero to account for the first part of the packet and the remainder of the packet size can be taken out of the next-colored bucket. This, of course, has another bug: the same packet cannot have both conforming and non-conforming components in the Diffserv architecture and so is not really appropriate here and we do not discuss this option further here.

Unfortunately, the thing that cannot be done is exactly to fit the token burst specification with random sized packets: therefore token buckets in a variable length packet environment always have a some variance from theoretical reality. This has also been observed in the ATM Guaranteed Frame Rate (GFR) service category specification and Frame Relay. A number of observations may be made:

- Operationally, a token bucket meter is reasonable for traffic which has been shaped by a leaky bucket shaper or a serial line. However, traffic in the Internet is rarely shaped in that way: TCP applies no shaping to its traffic, but rather depends on longer-range ACK-clocking behavior to help it approximate a certain rate and explicitly sends traffic bursts during slow start, retransmission, and fast recovery. Video-on-IP implementations such as [VIC] may have a leaky bucket shaper available to them, but often do not, and simply enqueue the output of their codec for transmission on the appropriate interface. As a result, in each of these cases, a token bucket meter may reject traffic in the short term (over a single token interval) which it would have accepted if it had a longer time in view and which it needs to accept for the application to work properly. To work around this, the token interval, B/R, must approximate or exceed the RTT of the session(s) in question and the burst size, B, must accommodate the largest burst that the originator might send.

- The behavior of a loose token bucket is significantly different from the token bucket description for ATM and for Frame Relay.

- A loose token bucket does not accept packets while the token count is negative. This means that, when a large packet has just borrowed tokens from the future, even a small incoming packet (e.g., a 40-byte TCP ACK/SYN) will not be accepted. Therefore, if such a loose token bucket is configured with a burst size close to the MTU, some discrimination against smaller packets can take place: use of a larger burst size avoids this problem.

- The converse of the above is that a strict token bucket sometimes does not accept large packets when a loose one would do so. Therefore, if such a strict token bucket is configured with a burst size close to the MTU, some discrimination against larger packets can take place: use of a larger burst size avoids this problem.

- In real-world deployments, MTUs are often larger than the burst size offered by a link-layer network service provider. If so then it is possible that a strict token bucket meter would find that traffic never matches the specified profile: this may be avoided by not allowing such a specification to be used. This situation cannot arise with a loose token bucket since the smallest burst size that can be configured is 1 bit, by definition limiting a loose token bucket to having a burst size of greater than one MTU.

- Both strict token bucket specifications, as specified in [SRTCM] and [TRTCM], and loose ones, are subject to a persistent under-run. These accumulate burst capacity over time, up to the maximum burst size. Suppose that the maximum burst size is exactly the size of the packets being sent - which one might call the "strictest" token bucket implementation. In such a case, when one packet has been accepted, the token depth becomes zero and starts to accumulate again. If the next packet is received any time earlier than a token interval later, it will not be accepted. If the next packet arrives exactly on time, it will be accepted and the token depth again set to zero. If it arrives later, however, accumulation of tokens will have stopped because it is capped by the maximum burst size: during the interval between the bucket becoming full and the actual arrival of the packet, no new tokens are added. As a result, jitter that accumulates across multiple hops in the network conspires against the algorithm to reduce the actual acceptance rate. Thus it usually makes sense to set the maximum token bucket size somewhat greater than the MTU in order to absorb some of the jitter and allow a practical acceptance rate more in line with the desired theoretical rate.

A.4 Mathematical Definition of Strict Token Bucket Conformance

The strict token bucket conformance behavior defined in [SRTCM] and [TRTCM] is not mandatory for compliance with any current Diffserv standards, but we give here a mathematical definition of two-parameter token bucket operation which is consistent with those documents and which can also be used to define a shaping profile.

Define a token bucket with bucket size B, token accumulation rate R and instantaneous token occupancy b(t). Assume that b(0) = B. Then after an arbitrary interval with no packet arrivals, b(t) will not change since the bucket is already full of tokens.

Assume a packet of size L bytes arrives at time t’. The bucket occupancy is still B. Then, as long as L <= B, the packet conforms to the meter, and afterwards

$$b(t') = B - L.$$

Assume now an interval delta_t = t - t’ elapses before the next packet arrives, of size L’ <= B. Just before this, at time t-, the bucket has accumulated delta_t*R tokens over the interval, up to a maximum of B tokens so that:

$$b(t-) = \min\{ B, b(t') + \text{delta\_t}*R \}$$

For a strict token bucket, the conformance test is as follows:

```
if (b(t-) - L’ >= 0) {
    /* the packet conforms */
    b(t) = b(t-) - L’;
}
else {
    /* the packet does not conform */
    b(t) = b(t-);
}
```

This function can also be used to define a shaping profile. If a packet of size L arrives at time t, it will be eligible for transmission at time te given as follows (we still assume L <= B):

$$te = \max\{ t, t'' \}$$

where t" = (L - b(t') + t'*R) / R and b(t") = L, the time when L credits have accumulated in the bucket, and when the packet would conform if the token bucket were a meter. te != t" only if t > t".

A mathematical definition along these lines for loose token bucket conformance is left as an exercise for the reader.

Authors' Addresses

Yoram Bernet
Microsoft
One Microsoft Way
Redmond, WA 98052

Phone: +1 425 936 9568
EMail: ybernet@msn.com

Steven Blake
Ericsson
920 Main Campus Drive, Suite 500
Raleigh, NC 27606

Phone: +1 919 472 9913
EMail: steven.blake@ericsson.com

Daniel Grossman
Motorola Inc.
20 Cabot Blvd.
Mansfield, MA 02048

Phone: +1 508 261 5312
EMail: dan@dma.isg.mot.com

Andrew Smith (editor)
Harbour Networks
Jiuling Building
21 North Xisanhuan Ave.
Beijing, 100089
PRC

Fax: +1 415 345 1827
EMail: ah_smith@acm.org

Full Copyright Statement

Acknowledgement

Funding for the RFC Editor function is currently provided by the Internet Society.