Diffserv Services API Specification Implementation Agreement

May 2004
Revision 1.0

Editor(s):

Anurag Bhargava, Ericsson, anurag.bhargava@ericsson.com

Peter Denz, Ericsson, peter.denz@ericsson.com

Copyright © 2004 The Network Processing Forum (NPF). All Rights Reserved.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published and distributed, in whole or in part, without restriction other than the following, (1) the above copyright notice and this paragraph must be included on all such copies and derivative works, and (2) this document itself may not be modified in any way, such as by removing the copyright notice or references to the NPF, except as needed for the purpose of developing NPF Implementation Agreements.

By downloading, copying, or using this document in any manner, the user consents to the terms and conditions of this notice. Unless the terms and conditions of this notice are breached by the user, the limited permissions granted above are perpetual and will not be revoked by the NPF or its successors or assigns.

THIS DOCUMENT AND THE INFORMATION CONTAINED HEREIN IS PROVIDED ON AN "AS IS" BASIS WITHOUT ANY WARRANTY OF ANY KIND. THE INFORMATION, CONCLUSIONS AND OPINIONS CONTAINED IN THE DOCUMENT ARE THOSE OF THE AUTHORS, AND NOT THOSE OF NPF. THE NPF DOES NOT WARRANT THE INFORMATION IN THIS DOCUMENT IS ACCURATE OR CORRECT. THE NPF DISCLAIMS ALL WARRANTIES, WHETHER EXPRESS, IMPLIED OR STATUTORY, INCLUDING BUT NOT LIMITED THE IMPLIED LIMITED WARRANTIES OF MERCHANTABILITY, TITLE OR FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF THIRD PARTY RIGHTS.

The words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", “MAY", and "OPTIONAL" in the remainder of this document are to be interpreted as described in the NPF Software API Conventions Implementation Agreement revision 1.0.

For additional information contact:
The Network Processing Forum, 39355 California Street,
Suite 307, Fremont, CA 94538
+1 510 608-5990 phone ✦ info@npforum.org

Table of Contents

1 Revision History

Revision	Date	Reason for Changes
1.0	05/18/2004	Created Rev 1.0 of the implementation agreement by taking the Difserv Services APIs (npf2003.289.05) and making minor editorial corrections.

2 Introduction

Differentiated Services (Diffserv) provides a framework that enables service providers to offer each customer a range of services that are differentiated on the basis of performance (and perhaps an associated price). Diffserv is designed to scale to large networks and a large customer population. It forces many of the QoS operations out of the network and to the nodes surrounding the network (edge nodes).

The key objectives of Diffserv are to classify traffic at the boundaries of a network, and to condition this traffic at the boundaries. The classification operation entails the assignment of the traffic to behavioral aggregates (BA). These behavioral aggregates consist of collections of packets with common characteristics, as far as how they are identified and treated by the network [RFC2474, RFC2475].

In this document, we present an architectural overview and the Service API (SAPI) which exists between the Diffserv Manager application and the Diffserv Forwarding Control module (also referred to as the service provider). This API will be used to communicate the information necessary to configure the underlying hardware in an appropriate manner to yield the desired QoS processing of packets flowing through this device.

Throughout this document, the term filter is used to represent a complete set of patterns that are associated with a Diffserv policy, and is analogous to a classifier in the Diffserv Management Information Base (MIB) document (RFC 3289). A rule is used to represent an individual element of a filter containing patterns for various fields of a packet header, analogous to the Diffserv MIB classifier element. For each rule, a series of actions are given which specify what is to be done to any packet that matches this rule. These terms and their associated data structures will be further defined in this document

2.1 Architecture Overview

The Diffserv Manager receives policy information from the OAM module. The OAM module, for example, can be a Command Line Interface for passing configured Diffserv policies by an administrator. The Diffserv policies are then communicated across the SAPI to the Diffserv Forwarding Control Layer. The ACL manager as shown in Figure 1 can also be part of the OAM module. It is just shown here for illustrative purposes. The OAM and the ACL Manager definitions are outside of the scope of this document.

The Diffserv manager processes this information into a list of rules and actions (also referred to as a Policy object), as well as various QoS objects, such as policers, schedulers, queues, and so forth, which are to be installed on a particular interface. This information is communicated across the SAPI to the Diffserv Forwarding Control Layer (FCL) which further processes the data and passes it down to the appropriate FAPI modules to lead to the appropriate hardware configuration such that the packets receive appropriate QoS processing. This flow of information is illustrated in Figure 1.

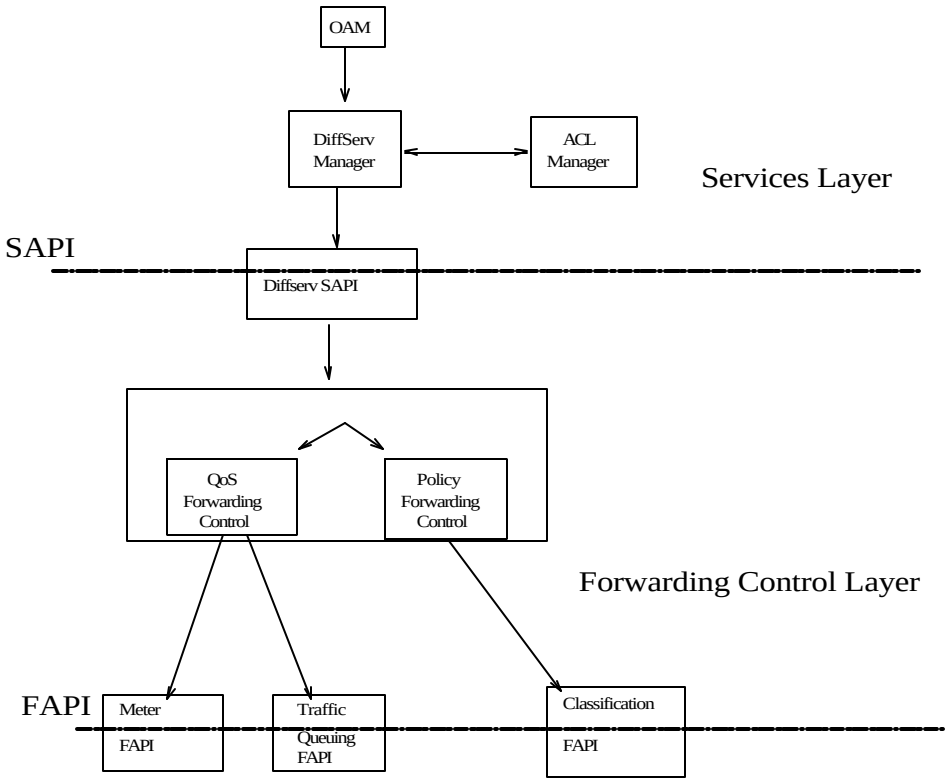

Figure 1: Diffserv SAPI

In this document we make no assumption about the locality of the Diffserv Manager application and the Diffserv Forwarding Control module--they could co-exist in the same process, reside in separate processes on the same processor, or even reside on separate processors. In the latter two cases, some type of IPC mechanism can be used to communicate the SAPI function calls from the application down to the FCL, and similarly, callbacks to the application.

Since we are following a parallel SAPI model, all of the communication across the SAPI involving QoS objects and filter and action processing are completed by the Diffserv Manager application. The SAPI was designed to follow the conventions set forth in the NPF software conventions document [SWAPI]. All function and type names defined herein contain the prefix NPF_DS_.

Figure 2 shows various components of the Diffserv SAPI. Policy Objects have Filter and Action representations. They can be applied on any interface either on the IP ingress, IP egress, locally generated traffic, locally terminated traffic, layer 2 decapsulation, or layer 2 encapsulation planes. There are various Mapping Tables which go along with actions. They can be global in the sense that they are applied to all the interfaces or they can be local to a specific interface. Finally, there are various QoS objects (queues, scheduler, shaper, policer, counter) which can be applied to any interface.

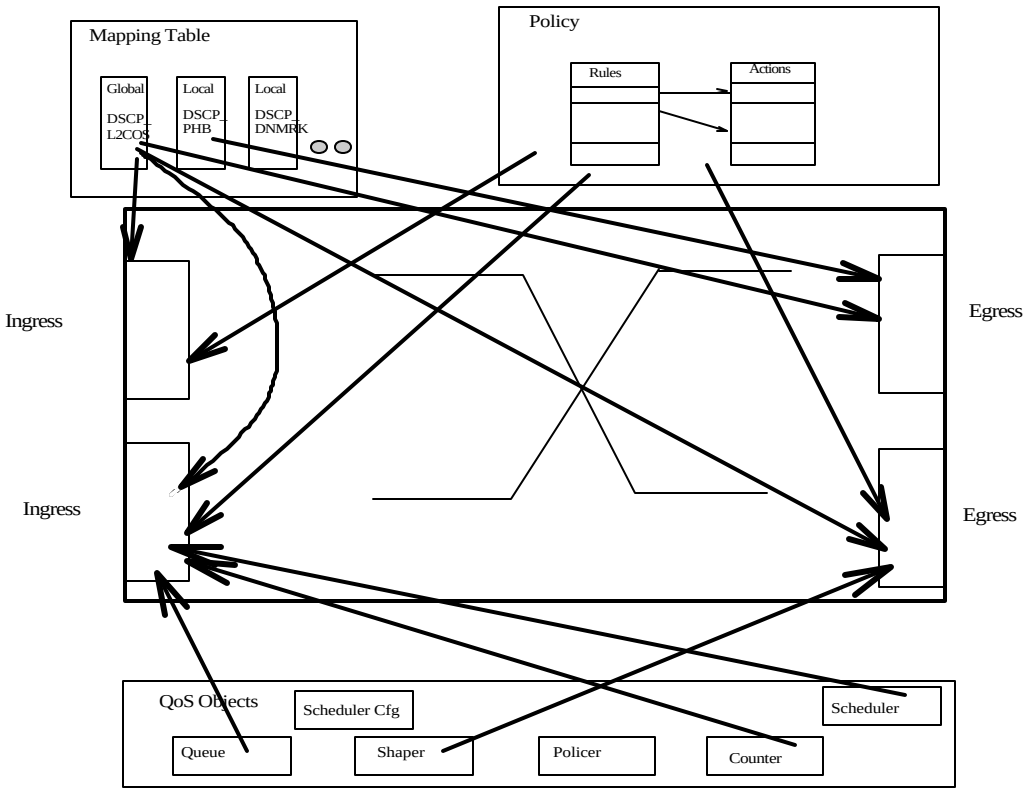

Figure 2: Various Diffserv Components and their locations

The Diffserv Manager will use the SAPI to communicate things such as rules and their associated action chains, policer data structures, queues, mapping tables, schedulers, and so forth to the underlying FCL. The basic logic is as follows. The Diffserv Manager first receives information regarding Diffserv policy that is to be applied on a particular interface. It makes a call to the function `NPF_DS_PolicyCreate()`, in which it receives a handle for a policy object that it will create. This handle will be used from now on to refer to this particular policy object. This policy object consists of a series of rules and actions to be performed when a packet matches the corresponding rule. This rule and action list is communicated down to the service layer through a call to `NPF_DS_PolicyUpdate()`, passing down the handle, filter and action list. This policy object is next bound to a particular interface by calling the function `NPF_DS_PolicyBind()`. The policy can be applied on the IP ingress, IP egress, locally terminated traffic, locally generated traffic, layer 2 decapsulation, or layer 2 encapsulation planes. The policy and QoS objects are also specified to be applied to a particular interface or hierarchical interface entity which can be a group of interfaces, for instance, a hierarchical interface entity (see Figure 3) can specify all logical interfaces on a given physical interface, or the global system.

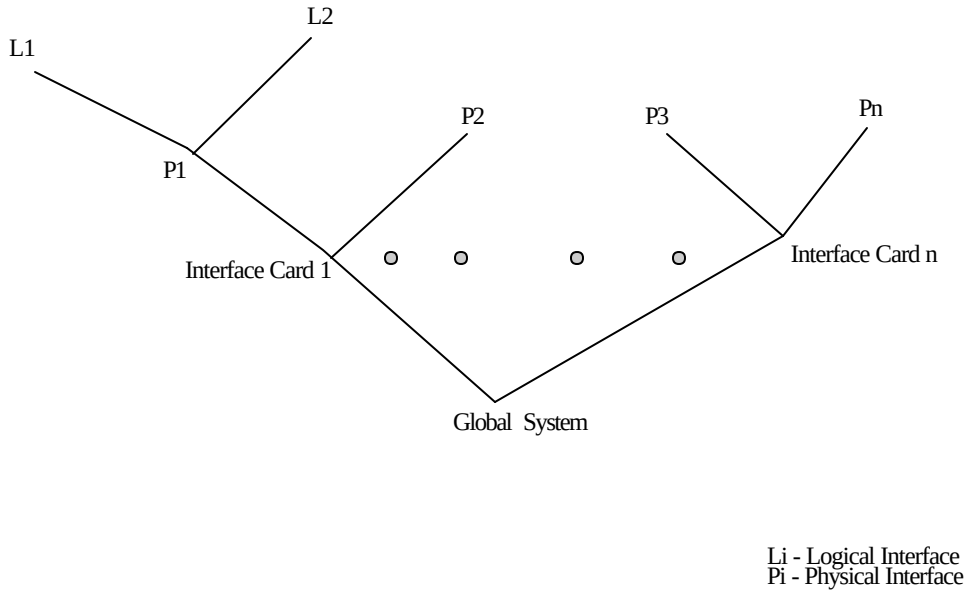

Figure 3: Interface Hierarchy

As changes occur in the system, then the filter and action list may be updated to reflect these changes by calling `NPF_DS_PolicyUpdate()` with the new filter and action list. Any previously registered filter and action list in this policy object are automatically removed and replaced with the new lists. If the application desires to make incremental updates; such as insertion, deletion, or modification; to the currently registered filter and action list, then this can be achieved using appropriate SAPI function calls. These function calls have not yet been defined.

If the policy is applied to new interfaces or deleted from interfaces, calls to `NPF_DS_PolicyBind()` and `NPF_DS_PolicyUnbind()` can be made to reflect these changes. A hardware update will occur when a policy is bound to an interface using `NPF_DS_PolicyBind()`. If policy is already bound to an interface, invoking `NPF_DS_PolicyUpdate()`should also result in hardware update. Once a policy is no longer going to be used it may be removed from the system and the corresponding handle freed by making a call to `NPF_DS_PolicyDestroy().`

Similar to the way the Diffserv Manager creates, updates, and destroys policy objects, it can also create, maintain, and destroy mapping tables and QoS objects, as well as binding to and unbinding them from interfaces. As in the policy object case, the "create" function assigns a handle to this object, and this handle is used to refer back to the object in subsequent function calls. The function calls used for mapping tables and QoS objects are:

- `NPF_DS_MapTableCreate()`
- `NPF_DS_MapTableUpdate()`
- `NPF_DS_MapTableDestroy()`
- `NPF_DS_MapTableBind()`
- `NPF_DS_MapTableUnbind()`
- `NPF_DS_QoSObjectCreate()`

- `NPF_DS_QoSObjectUpdate()`
- `NPF_DS_QoSObjectDestroy()`
- `NPF_DS_QoSObjectBind()`
- `NPF_DS_QoSObjectUnbind()`

The SAPI function calls `NPF_DS_PolicyCountersGet()` and `NPF_DS_QueueCountersGet()` allow the Diffserv Manager to query the current value of the counters on the hardware. `NPF_DS_CountersClear()` can be used to clear the counters.

Because hardware coming from different vendors can have very different packet processing capabilities, an infrastructure must be put into place to determine the capabilities of a given underlying system. To this end, the Diffserv SAPI organizes all of its features into a matrix. Certain features only make sense on certain interface planes. This matrix contains a series of bits for each feature on each plane. With this matrix, the underlying system can communicate to the SAPI user its capabilities (for instance, can a given interface filter based upon the IPv4 source address? If so, can it filter based upon a filter prefix?). Since there will likely be many interfaces in a given system and only a small number of unique capability matrices, the unique capability matrices will be referred to by profile identifiers, assigned by the SAPI implementation. The function `NPF_DS_CapabilityProfileQuery()` can be used to obtain an array of all profile identifiers and their respective capability matrix. The function `NPF_DS_CapabilityInterfaceQuery()` allows the user to obtain the profile identifier for a list of interfaces.

The Diffserv SAPI includes a series of functions through which an application can query objects which have been previously configured. For each object type (policy object, QoS object, and mapping table object), the application can query all handles of the respective objects that have been configured, query all handles of the respective objects that have been bound to a given interface, and query the contents of a given handle.

2.2 Overview of the Callback based Call Model

The NPF specifies that all SAPIs are to follow a callback based call model. In such a model, an application makes an API call into the service provider and return is immediately given back to the application. The response from the API call comes back later in the form of a callback function call from the service provider to the application. This type of system requires the application to register one or more callback functions for the API functions. The function `NPF_DS_Register()` is used to register a callback function for each API function. Similarly, `NPF_DS_Deregister()` is used to unregister a callback function for an API function. These function calls are discussed in more detail in section 4.4.3.

Since many API functions can share the same callback function, there must be a mechanism to decide, when a callback returns, which API function it corresponds to. Additionally, there must be some way to correlate which function call instance a particular callback is responding to. Notice that an API may be called any number of times before the first callback returns. Hence, in order to correlate the callbacks to function call instances, each instance will register a correlator value, which is only meaningful to the application. This correlator should be unique among all outstanding function call instances, hence, when a callback arrives with a given correlator, it is known which function call instance it corresponds to.

2.3 Assumptions

- All API calls are considered asynchronous in nature, unless otherwise specified. The definitions of synchronous and asynchronous behavior are specified by the NPF Software Convention Implementation Agreement Document [SWAPI].

2.4 Scope

The purpose of this document is to provide an overview of the Diffserv SAPI. To this end, we will present each of the function call prototypes that make up the Diffserv SAPI, describe the flow of data down into the forwarding control layer and back up to the Diffserv Manager through the asynchronous callbacks, describe the data structures that carry this data, and describe error return codes that are provided by the function in the event of an error. This document does not cover the actual implementation of the Diffserv SAPI services, nor does it cover any IPC mechanism that is used to communicate the SAPI function calls from the application to the provider layer. Finally, the communication details between the Diffserv Manager and any other module, other than the Diffserv Forwarding Control Manager is considered beyond the scope of this document.

This document covers the following topics for the configuration and management of policy objects, QoS objects and various Mapping Tables used by them.

- Policy objects, QoS objects, Tables, and their related data structure definitions. The data types and structures generally used by all API specifications are defined by the NPF Software Conventions Implementation Agreement Document; however, Diffserv specific structures are defined in this document.

- API definitions for Diffserv related objects configuration and management. The API function details will include input/output parameters, return code specifications and detailed usage notes specific to each invocation.

- No events have been identified for this API. All related sections, although present in this document are specifically noted as 'not applicable'.

2.5 Dependencies

The function declarations described in this document follow the conventions provided by the NPF in the NPF Software API Conventions Implementation Agreement. Following the model provided in these documents, the Diffserv SAPI will involve asynchronous function calls from the Diffserv Manager into the FCL and corresponding callback function calls from the FCL into the Diffserv Manager.

3 Data Types

This section defines return codes, function calls, and call backs that will be used in the SAPI function calls

3.1 Basic Types

3.1.1 Policy Handle Identifier

Here we define the handle identifier which is used to identify a Policy object that has been registered with the Diffserv Forwarding Control module. It will be used across the SAPI layer. The value of the handle only has meaning within the provider layer.

```
typedef NPF_uint32_t NPF_DS_PolicyHandle_t;
```

At the time of creation of the policy handle, the application refers to the object using an application-assigned resource identifier which takes the following form.

```
typedef NPF_uint32_t NPF_DS_PolicyID_t;
```

3.1.2 QoS handle Identifier

Here we define the handle identifier which is used to identify a QoS object that has been registered with the Diffserv Forwarding Control module. It will be used across the SAPI layer. The value of the handle only has meaning within the provider layer.

```
typedef NPF_uint32_t NPF_DS_QoSHandle_t;
```

At the time of creation of the QoS handle, the application refers to the object using an application-assigned resource identifier which takes the following form.

```
typedef NPF_uint32_t NPF_DS_QoSObjectID_t;
```

3.1.3 Table Handle Identifier

Here we define the handle identifier which is used to identify a mapping table object that has been registered with the Diffserv Forwarding Control module. It will be used across the SAPI layer. The value of the handle only has meaning within the provider layer.

```
typedef NPF_uint32_t NPF_DS_MapTableHandle_t;
```

At the time of creation of the mapping table handle, the application refers to the object using an application-assigned resource identifier which takes the following form.

```
typedef NPF_uint32_t NPF_DS_MapTableID_t;
```

3.1.4 Filter Plane Type

As shown in Figure 4, filter planes can be categorized as IP Ingress, IP Egress, local traffic termination, local traffic generation, layer 2 decapsulation, and layer 2 encapsulation. Policies can be applied to any of these planes for treating the traffic appropriately. Ingress and Egress filters will filter out the packets matching the ingress and egress filters criteria, respectively. Similarly, filters can be installed for local generated and local terminated traffic. Layer 2 filter planes can only contain rules specifying layer 2-based rule fields, whereas the other planes may not specify layer 2 fields. Filter Plane types are defined as follows:

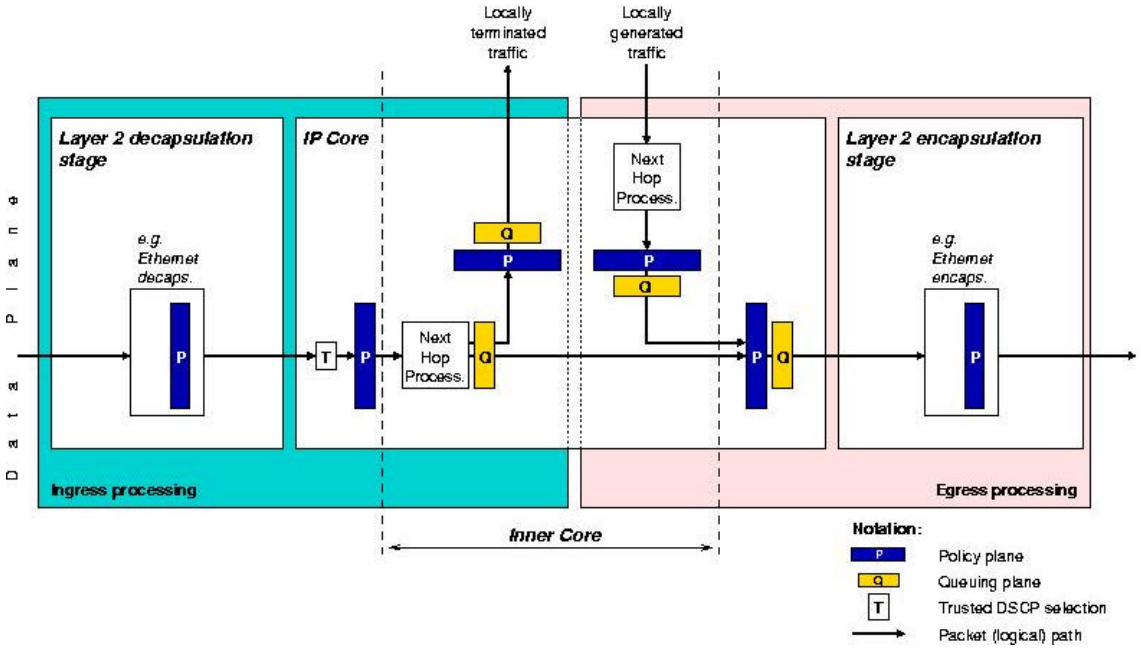

Figure 4: Various Planes

```
typedef enum {
   NPF_DS_PLANE_IPINGRESS         = 0,
   NPF_DS_PLANE_IPEGRESS          = 1,
   NPF_DS_PLANE_LOCAL_TERMINATED  = 2,
   NPF_DS_PLANE_LOCAL_GENERATED   = 3,
   NPF_DS_PLANE_L2DECAP           = 4,
   NPF_DS_PLANE_L2ENCAP           = 5
} NPF_DS_FilterPlane_t;
```

3.1.5 Filter Representation

This section defines the various Diffserv related fields necessary to configure any Diffserv rules.

3.1.5.1 Filtering Model Overview

The data structures presented in this section can be used to convey filter rules and corresponding action chains that should be executed if the given filter rule matches. These data structures are designed to support the following advanced classification features:

- **Rule grouping**: The rule and action representation allows an efficient expression of grouped rules, (i.e., when a large number of rules trigger the execution of the same action). This essentially means that the input should contain an ordered set of rule groups, each group "pointing" to a set of actions. When a rule matches in a certain group, the actions related to that rule group are executed. Rule groups essentially express an *OR* logical operation over a set of rules. For example, if a rule group contains rule *A*, *B*, *C*, and *D*, the corresponding action is executed if rule (*A OR B OR C OR D*) matches. This simple form of rule grouping does not depend on any specific ordering of rules within the group.

- **Negated rules:** The API allows the marking of any rule (in any rule group) to have a negated effect. Negated effect of a rule is to consider the next rule group in the group chain instead of executing the actions associated with the rule group.
  Negated rules can have two interpretations. The first one is a pure mathematical interpretation, allowing the formation of expressions such as (*A OR B OR !C OR D*), where the exclamation mark denotes the negated rule. In this form the ordering within the group is still irrelevant. For example, if rule *C* and *D* both match, the group is still considered matching, and its actions will be executed. The second interpretation of negated rules introduces the notion of ordering in the rule group. In this case, for example, the ordered list of rules (*A, B, !C, D*) expresses that the rules must be evaluated in that order, and the first matching rule determines the outcome of the rule group: if the first matching rule is a normal rule, the actions are executed; if the first matching rule is a negated rule, the group is considered non-matching and the next rule group is evaluated. Note that both forms of group evaluation occur in products, and hence it is important to allow the expression of both scenarios in the API.

- **Multiple match:** In certain cases the user may need to express the conditions for certain actions as a simultaneous match of multiple rule groups. This essentially introduces an *AND* operation among rule groups or individual rules (since a rule group can contain a single rule). The model supports the creation of *AND* expressions.

- **Continue action:** In typical user configurations, when a rule group evaluates as matching, the corresponding actions must be executed and the packet is not subject to additional rule evaluations. In certain CLIs, however, it is possible to request further rule evaluation after action execution. This is expressed by an implicit return to the next logical filter element after processing a non-terminating action chain, as well as an optional explicit "RETURN" action.

The filtering data structure allows for the specification of rule groups:

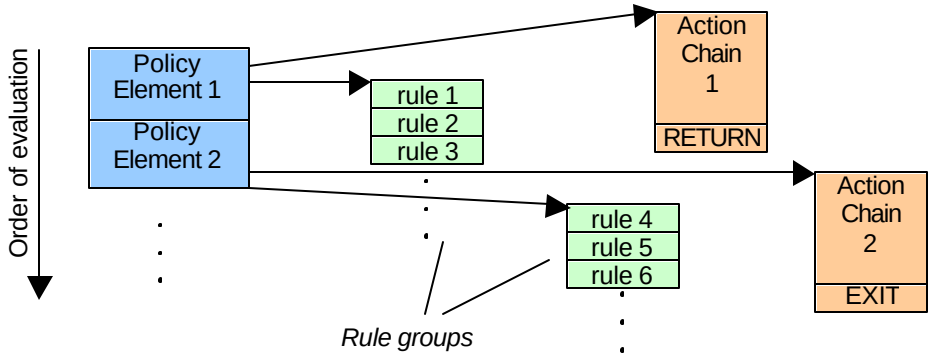

Figure 5 -- High Level View of Rule Group Specification

As can be seen in this Figure 5, an ordered list of **policy elements** (also called **filter elements**) can be expressed. Each policy element can point to a rule group and an action chain. The rule group represents the *condition* of the expression, the action chain represents the *consequence* of the matching condition. If the condition is not satisfied (rule group is non-matching), then the next expression element is evaluated.

The rule group is assumed to express an ACL-like behavior: the rules are evaluated in descending order, and the first matching rule determines if the rule-group is considered matching. If there is no rule in the rule group matching the packet, the rule-group is considered non-matching. As you will see later, the rule group can be substituted by other conditions, allowing for expression of pure *AND*, pure *OR*, or more complex conditions.

Any rule can be marked "**negated**". Any rule in a rule group can be marked as "negated". The meaning of a matching negated rule depends on the type of the rule group.

1. For the simple case when the expression element uses a single rule as a condition, the rule group can be substituted by a **single rule element**. (Conceptually that is the same as a rule group with a single entry, but there are some simplifications in the encoding of this case, for code efficiency.)
2. The rule group can be substituted with a so-called **expression**. Two types of expressions are allowed: *AND* and *OR*. Both types of expressions contain a list of entries. Each entry can refer to a single rule, or a rule group (ACL-like).
3. An atomic action, *return*, specifies explicitly that evaluation must continue from the next policy element. However, this is the default behavior if the previous matching policy element does not contain a terminating action, which will be discussed in a later section.

At its highest level, the policy (filter) is expressed as an array of policy elements which are evaluated in an *OR*-multiple match manner. That is, Each policy element is evaluated, in turn. If it evaluates to true, then the corresponding action chain is executed. If that action chain does not contain a terminating action, then evaluation continues with the next policy element in the array. This evaluation continues until a terminating action is executed, or the end of the policy element list is reached.

Each policy element also contains a type selector, an action ID to refer to an associated action chain to execute if this policy element evaluates to true, and a pointer to the underlying expression. Each expression entry can represent either a singular rule, an *OR* expression, an *AND* expression, or a list of rules that are to be evaluated in a traditional ACL fashion. According to what type this entry is, it will contain a pointer to either a single rule, a pointer to an expression array, or a pointer to a rule array.

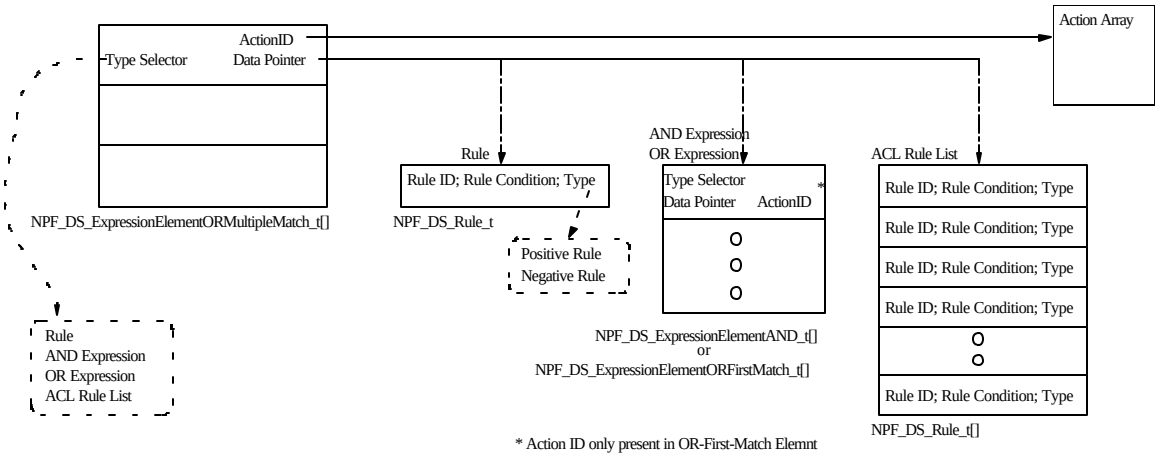

Figure 6 -- High Level View of the Proposed Data Structures

The structure `NPF_DS_ExpressionElementORMultipleMatch_t`, which represents a single policy element, can support representation of Boolean expressions through its ability to express *AND* as well as *OR* expressions. Each of these policy element entries can represent either a singular rule, an `OR-First-Match` expression, an *AND* expression, or a list of rules that are to be evaluated in a traditional ACL fashion. According to what type this entry is, it will contain a pointer to either a single rule, a pointer to an expression array, or a pointer to a rule array.

Now let us consider each of the possible types of conditions that can be referenced from instances of type `NPF_DS_ExpressionElementORMultipleMatch_t`:

1. First, they can point to a single rule. This rule is compared to the packet and will evaluate to true or false.

2. They can refer to an ACL rule group, which is simply a list of two or more rules which are to be evaluated in an ACL fashion. That is, the rules are evaluated, one-by-one, in the order that they exist in the array, until a matching rule is found. If that matching rule does not have its *NEG* flag set (i.e., it is not a negative rule), then the entire ACL list evaluates to true and evaluation of this list can stop. On the other hand, if the matching rule does have its *NEG* flag set, then the entire ACL list evaluates to false and evaluation of this list can stop.

3. An OR-First-Match expression is a list of two or more expression terms (`NPF_DS_ExpressionElementORFirstMatch_t`), each of which will evaluate to true or false. The OR-expression evaluates to true if any one of its terms evaluates to true. Evaluation of the terms is done in the order in which they are specified in the array, and stops as soon as one term evaluates to true.

4. An *AND* expression is a list of two or more expression terms (`NPF_DS_ExpressionElementAND_t`), each of which will evaluate to true or false. The *AND* expression evaluates to true if all of its terms evaluates to true.

Each instance of an policy element and OR-First-Match express element contains an action identifier which is executed if this entry evaluates to true. If it is not desired to have an action associated with a given expression element, then its action identifier should be set to zero.

3.1.5.2 Filtering Model Data Structure

The Diffserv Manager creates a data structure containing rules and their associated actions. The rules are specified in order of decreasing precedence. A basic rule takes the form of `NPF_DS_Rule_t`. As can be seen below, each element contains a rule identifier, a rule condition, and a rule type. As we shall soon see, the rule identifier can be used as a way of referring to rules at a future point in time, such as for enabling references for incremental rule addition, deletion, or modification. These rule identifiers should be positive integers which are defined by the application and must be unique for each rule that is referenced within the policy object.

A rule can be specified, using the `NPF_DS_RuleType_t` enumeration below, as either positive or negative. A positive rule will evaluate to true for a packet if the packet matches the pattern specified by the rule. On the other hand, a negative rule will evaluate to true for a packet if the packet does not match the pattern specified by the rule.

```
typedef NPF_int32_t NPF_DS_RuleID_t;

/* Rule type:  Positive or Negative */
typedef enum {
   NPF_DS_RULE_TYPE_POSITIVE = 0,
   NPF_DS_RULE_TYPE_NEGATIVE = 1
} NPF_DS_RuleType_t;

/* Rule Entry */
typedef struct {
   NPF_DS_RuleID_t id;
   NPF_DS_RuleCond_t ruleCond;
   NPF_DS_RuleType_t ruleType;
} NPF_DS_Rule_t;
```

3.1.5.3 Rule Pattern Representation

Now looking at the IP conditions, specified in type `NPF_DS_RuleCondIP_t`, we see that we may specify either a layer 2 or a layer 3+4 rule condition. Since layer 2 rules can only exist on planes `NPF_DS_PLANE_L2DECAP` and `NPF_DS_PLANE_L2ENCAP`, and layer 3/layer 4 rules can only exist on planes `NPF_DS_PLANE_IPINGRESS, NPF_DS_PLANE_IPEGRESS, NPF_DS_PLANE_LOCAL_TERMINATED`, or `NPF_DS_PLANE_LOCAL_GENERATED` then they cannot coexist on the same plane, hence they are specified in a union.

```
/* Layer 2 or Layer 3+4 rule conditions */
typedef union {
   NPF_DS_RuleCond_L2_t cond_L2;
   NPF_DS_RuleCond_L34_t cond_L34;
} NPF_DS_RuleCond_t;
```

For Layer2 classification, rules can be established on the source and destination MAC addresses. It can also be further classified using the 802.1p priority and/or vlan ID for VLANS (802.1Q). The condition `NPF_DS_RuleCond_L2_t` is further decomposed into layer 2 fields, as follows.

```
/* Layer 2 rule conditions */
typedef struct {
   NPF_uchar8_t mac_SA[6]; /* 0 means any */
```

```
   NPF_uchar8_t mac_DA[6]; /* 0 means any */
   NPF_uchar8_t priority;  /* 802.1p priority */
   NPF_uchar8_t vlan_ID;   /* priority + vlan_ID forms tag
                              for 802.1Q */
} NPF_DS_RuleCond_L2_t;
```

Now looking further at the layer 3 and 4 conditions, specified in type `NPF_DS_RuleCondIP_L34_t`, we see that we may specify values for fields in the layer 3 and layer 4 headers of an IP packet.

```
/* IP layer 3/4 rule conditions */
typedef struct {
   NPF_DS_RuleCondIP_L3_t condIP_L3;
   NPF_DS_RuleCondIP_L4_t condIP_L4;
} NPF_DS_RuleCond_L34_t;
```

The `NPF_DS_RuleCondIP_L3_t` structure is further decomposed into a layer 3 type field, which can specify either `NPF_DS_RULE_IPV4` or `NPF_DS_RULE_IPV6`, and a union containing the IPv4 and IPv6 conditions, respectively.

The type of layer 3 rules that are applicable on a particular interface (e.g., IPv4 or IPv6 rules) are dependent upon the type of network the interface is connected to. Any IPv4 rules that are applied to an IPv6 interface will be rejected, likewise, any IPv6 rules that are applied to an IPv4 interface will be rejected. In a situation where the interface which a filter is bound to is actually a hierarchy of interfaces containing a mix of IPv4 and IPv6 based interfaces, only those rules with the correct version will be applied to the individual interfaces.

```
/* Layer 3 type selector—IPv4 or IPv6 rule */
typedef enum {
   NPF_DS_RULE_IPV4 = 0,
   NPF_DS_RULE_IPV6 = 1
} NPF_DS_RuleL3Type_t;


/* Layer 3 rule conditions */
typedef struct {
   NPF_DS_RuleL3Type_t L3Type;
   union {
      NPF_DS_RuleCondIPv4_L3_t condIPv4_L3;
      NPF_DS_RuleCondIPv6_L3_t condIPv6_L3;
   } condIP_L3;
} NPF_DS_RuleCondIP_L3_t;
```

Further decomposing the IPv4 layer 3 conditions, as shown in `NPF_DS_RuleCondIPv4_L3_t`, we see that rules may be configured to specify values for any of the following IPv4 header fields: IP protocol value, packet length with range, source IP address with an associated left-justified mask, destination IP address with an associated left-justified mask, Type of Service (TOS) byte (which can further be subdivided into Diffserv Codepoint (DSCP), IP Precedence and TOS field using the masks specified below). Finally, a Fragments condition can be set to specify that this rule can match only non-initial fragments (i.e., packets with a nonzero fragment offset field). Each of these fields can be wildcarded, by assigning special values. Macros for this special wildcarded values are given below the structure definition.

```
/* IPv4 layer 3 rule conditions */
typedef struct {
   NPF_char8_t proto;
   NPF_uint32_t lengthMin, lengthMax;
   NPF_IPv4Address_t srcAddr, srcAddrMask;
   NPF_IPv4Address_t destAddr, destAddrMask;
   NPF_char8_t tos, tosMask;
   NPF_uchar8_t fragments;
            /* 0 means any; 1 means */
            /*   non-initial fragments only */

} NPF_DS_RuleCondIPv4_L3_t;

#define NPF_DS_WILDCARD_IPV4_PROTO        255
#define NPF_DS_WILDCARD_IPV4_LENGTHMIN    0
#define NPF_DS_WILDCARD_IPV4_LENGTHMAX    65535
#define NPF_DS_WILDCARD_IPV4_SRCADDRMASK  0
#define NPF_DS_WILDCARD_IPV4_DESTADDRMASK 0
#define NPF_DS_WILDCARD_IPV4_TOSMASK      0
#define NPF_DS_WILDCARD_IPV4_FRAGMENTS    0


#define NPF_DS_TOSBYTE_DSCP_MASK          0xFC
#define NPF_DS_TOSBYTE_IPPREC_MASK        0xE0
#define NPF_DS_TOSBYTE_TOS_MASK           0x07
```

Further decomposing the IPv6 layer 3 conditions, as shown in `NPF_DS_RuleCondIPv6_L3_t`, we see that rules may be configured to specify values for any of the following IPv6 header fields: priority, flow label, next header, source address and mask, destination address and mask. Each of these fields can be wildcarded by providing special values for the fields. These are given as macros following the structure.

```
/* IPv6 layer 3 rule conditions */
typedef struct {
   NPF_char8_t class;
   NPF_int64_t flowLabel;
   NPF_int16_t next_header;
   NPF_IPv6Address_t srcAddr, srcAddrMask;
   NPF_IPv6Address_t destAddr, destAddrMask;

} NPF_DS_RuleCondIPv6_L3_t;

#define NPF_DS_WILDCARD_IPV6_PRIORITY     -1
#define NPF_DS_WILDCARD_IPV6_FLOW_LABEL   0
#define NPF_DS_WILDCARD_IPV6_NEXT_HEADER  -1
#define NPF_DS_WILDCARD_IPV6_SRCADDR      0
#define NPF_DS_WILDCARD_IPV6_SRCADDRMASK  0
#define NPF_DS_WILDCARD_IPV6_DESTADDR     0
#define NPF_DS_WILDCARD_IPV6_DESTADDRMASK 0
```

Similarly, many fields can be specified for the IP packet's layer 4 header, as shown in NPF_DS_RuleCondIP_L4_t. These layer 4 fields are subdivided into those associated with TCP or UDP, and those associated with ICMP. In the case of a TCP or UDP packet, the source port and destination port ranges can be specified. To specify a single source or destination port, simply make the minimum value equal the maximum value, and to specify a wildcarded field, set the minimum and maximum values to NPF_DS_WILDCARD_TCPUDP_SRCPMIN and NPF_DS_WILDCARD_TCPUDP_SRCPMAX; or NPF_DS_WILDCARD_TCPUDP_DSTPMIN and NPF_DS_WILDCARD_TCPUDP_DSTPMAX, respectively.

In the case of an ICMP header, we can specify the ICMP type and code fields. These can be wildcarded setting the fields equal to NPF_DS_WILDCARD_ICMP_TYPE or NPF_DS_WILDCARD_ICMP_CODE.

```
/* IP layer 4 rule conditions */
typedef union {
   struct {
      NPF_uint32_t SRCP_Min, SRCP_Max;
      NPF_uint32_t DSTP_Min, DSTP_Max;
      NPF_char8_t TCP_Flags, TCP_FlagsMask;
   } L4TCP_UDP_Cond;
   struct {
      NPF_int32_t Type;
      NPF_int32_t Code;
   } L4ICMP_Cond;
} NPF_DS_RuleCondIP_L4_t;

#define NPF_DS_WILDCARD_TCPUDP_SRCPMIN     0
#define NPF_DS_WILDCARD_TCPUDP_SRCPMAX     65535
#define NPF_DS_WILDCARD_TCPUDP_DSTPMIN     0
#define NPF_DS_WILDCARD_TCPUDP_DSTPMAX     65535
#define NPF_DS_WILDCARD_TCP_FLAGMASK       0
#define NPF_DS_WILDCARD_ICMP_TYPE          -1
#define NPF_DS_WILDCARD_ICMP_CODE          -1
```

3.1.5.4 Rule Organization

This section covers the data structures that are used to construct the filtering data structures.

The following definitions allow specification of rule arrays.

```
/* Rule Array */
typedef struct {
   NPF_uint32_t ruleArraySize;
   NPF_DS_Rule_t *ruleArray;
} NPF_DS_RuleArray_t;
```

The following enumerations, specify all of the possible forms that the various expression terms (OR-Multiple-Match, OR-First-Match, and AND) can be. An expression element of type OR-Multiple-Match, can be followed by either a single rule, an AND-expression, an OR-First-Match expression, or a ACL rule list. Similarly, the OR-First-Match expression elements and the AND expression elements are followed by either a single rule or an ACL rule list.

```
/* Expression OR-Multiple-Match Type Selector */
typedef enum {
   NPF_DS_MULTI_OR_RULE                        = 0,
   NPF_DS_MULTI_OR_EXPRESSION_AND              = 1,
   NPF_DS_MULTI_OR_EXPRESSION_OR_FIRST_MATCH = 2,
   NPF_DS_MULTI_OR_ACL_RULE_LIST               = 3
} NPF_DS_ExpressionTypeSelectorORMultipleMatch_t;

/* Expression OR-First-Match Type Selector */
typedef enum {
   NPF_DS_OR_FIRST_MATCH_RULE          = 0,
   NPF_DS_OR_FIRST_MATCH_ACL_RULE_LIST = 1
} NPF_DS_ExpressionTypeSelectorORFirstMatch_t;

/* Expression AND Type Selector */
typedef enum {
   NPF_DS_AND_RULE          = 0,
   NPF_DS_AND_ACL_RULE_LIST = 1
} NPF_DS_ExpressionTypeSelectorAND_t;
```

An AND expression element takes the following form. The AND expression can point to different entities, as specified by the `typeSelector` field. The union `expressionData` is to be interpreted appropriately, depending upon the value of the `typeSelector` (i.e., the element will either point to a single rule or an array of ACL rules).

```
/* Expression Element -- AND */
typedef struct {
   NPF_DS_ExpressionTypeSelectorAND_t typeSelector;
   union {
      NPF_DS_Rule_t *rule;  /* for ts = rule */
      NPF_DS_RuleArray_t ACL_ruleArray; /* for ts
                                           RuleList*/
   } expressionData;
} NPF_DS_ExpressionElementAND_t;
```

An array of AND expression elements can be specified using the following definition.

```
/* Expression Array -- AND */
typedef struct {
   NPF_uint32_t arraySize;
   NPF_DS_ExpressionElementAND_t *expressionANDArray;
} NPF_DS_ExpressionArrayAND_t;
```

The OR-First-Match expression element takes the following form. Similar to the AND expression, the OR-First-Match expression can point to different entities, as specified by the `typeSelector` field. The union `expressionData` is to be interpreted appropriately, depending upon the value of the `typeSelector` (i.e., the element will either point to a single rule or an array of ACL rules). The OR-First-Match element can additionally specify an action identifier, which refers to an entry in the action structure. This action is to be executed if the corresponding element evaluates to true.

```
/* Expression Element -- OR-First-Match */
typedef struct {
   NPF_DS_ExpressionTypeSelectorORFirstMatch_t
typeSelector;
union {
```

```
            NPF_DS_Rule_t *rule;  /* for ts = rule */
            NPF_DS_RuleArray_t ACL_ruleArray; /* for ts =
                                                RuleArray */
         } expressionData;

         NPF_DS_ActionID_t actionID;

      } NPF_DS_ExpressionElementORFirstMatch_t;
```

An array of OR-First-Match expression elements can be specified using the following definition.

```
      /* Expression Array -- OR-First-Match */
      typedef struct {
         NPF_uint32_t arraySize;
         NPF_DS_ExpressionElementORFirstMatch_t
                *expressionArrayORFirstMatch;
      } NPF_DS_ExpressionArrayORFirstMatch_t;
```

The OR-Multiple-Match expression element takes the following form. Similar to the previous expression elements, the OR-Multiple-Match expression can point to different entities, as specified by the `typeSelector` field. The union `expressionData` is to be interpreted appropriately, depending upon the value of the `typeSelector` (i.e., the element will either point to a single rule, an array of ACL rules, an AND expression, or an OR-First-Match expression). The OR-Multiple-Match element can additionally specify an action identifier, which refers to an entry in the action structure. This action is to be executed if the corresponding element evaluates to true.

```
      /* Expression Element – OR-Multiple-Match */
      typedef struct {
         NPF_DS_ExpressionTypeSelectorORMultipleMatch_t
      typeSelector;

         union {
            NPF_DS_Rule_t *rule;  /* for ts = rule */
            NPF_DS_ExpressionArrayAND_t expressionArrayAND;
                        /* for ts = Expression_AND */
            NPF_DS_ExpressionArrayORFirstMatch_t
                           expressionArrayORFirst;
                        /* for ts = Expression_OR */

            NPF_DS_RuleArray_t ACL_ruleArray; /* for ts =
                                                RuleArray */
         } expressionData;

         NPF_DS_ActionID_t actionID;

      } NPF_DS_ExpressionElementORMultipleMatch_t;
```

The root of the classifier specification is an array of OR-Multiple-Match expressions, as specified in `NPF_DS_Policy_t`.

```
      typedef struct {
         NPF_uint32_t policyElementCount;
         NPF_DS_ExpressionElementORMultipleMatch_t
                        *policyElementArray;
      } NPF_DS_Policy_t;
```

3.1.6 Action Representation

Actions are given explicit identifiers, defined by the structure given below. These identifiers are used to map rules to their respective action chains. This can also be used to pinpoint existing actions for incremental addition of new actions, deletion of actions, or modification of actions. These identifiers are assigned and maintained by the application and used by the provider to map rules to action chains. As we will see later, there are some cases where this identifier field is not needed and may merely be set to zero by the application. It is necessary that all of the nonzero referenced identifiers are unique within a given policy object.

```
typedef NPF_uint32_t NPF_DS_ActionID_t;
```

The action types are referred to by the values of the enumeration type, `NPF_DS_ActionType_t.`

```
/* Action Types */
typedef enum {
   NPF_DS_ACTION_NOP          = 0,
   NPF_DS_ACTION_COUNTER      = 1,
   NPF_DS_ACTION_POLICE       = 2,
   NPF_DS_ACTION_SETDSCP      = 3,
   NPF_DS_ACTION_SETIPPREC    = 4,
   NPF_DS_ACTION_SETNEXTHOP   = 5,
   NPF_DS_ACTION_SETFIB       = 6,
   NPF_DS_ACTION_SETCOLOR     = 7,
   NPF_DS_ACTION_DOWNMARK     = 8,
   NPF_DS_ACTION_DROP         = 9,
   NPF_DS_ACTION_SHAPE        = 10,
   NPF_DS_ACTION_FORWARD      = 11,
   NPF_DS_ACTION_RETURN       = 12,
   NPF_DS_ACTION_QUEUE        = 13,
   NPF_DS_ACTION_SETPSC       = 14,
   NPF_DS_ACTION_SETTRUST     = 15
} NPF_DS_ActionType_t;
```

Actions are specified as an array of action elements, where each action element is of the form `NPF_DS_Action_t`. As we see in the definition below, each action element contains an `actionID` field, which provides a way to refer to this action, such as for mapping a rule to an action chain. Next, we have the `actionType` field which specifies how we are to interpret the data within the union. We will look at each of these along with the actions they represent later in this section. The last two fields of the action element allow us to specify a sub-action array, of specified length.

```
/* Action Entry */
typedef struct NPF_DS_Action {
   NPF_DS_ActionID_t actionID;
   NPF_DS_ActionType_t actionType;
   union {
      NPF_DS_ActionNOP_t           nop;
      NPF_DS_ActionCounter_t       counter;
      NPF_DS_ActionPolice_t        police;
      NPF_DS_ActionSetDSCP_t       setdscp;
      NPF_DS_ActionSetIPPREC_t     setipprec;
      NPF_DS_ActionSetNextHop_t    setnexthop;
      NPF_DS_ActionSetFIB_t        setfib;
      NPF_DS_ActionSetColor_t      setcolor;
      NPF_DS_ActionDownmark_t      downmark;
      NPF_DS_ActionDrop_t          drop;
```

```
        NPF_DS_ActionShape_t          shape;
        NPF_DS_ActionForward_t        forward;
        NPF_DS_ActionReturn_t         retrn;
        NPF_DS_ActionQueue_t          queue;
        NPF_DS_ActionSetPSC_t         setpsc;
        NPF_DS_ActionSetTrust_t       settrust;
    } actionData;
    struct NPF_DS_Action *subactionArray;
    NPF_uint16_t subactionArraySize;
} NPF_DS_Action_t;
```

In Figure 7, we see a snapshot of a particular usage of this action structure. In this example, the top-level action structure is configured as a series of NOP actions, which merely provide a base for a pointer to a larger action chain. Each of these top-level NOP actions will be mapped to by one or more rules, as the diagram shows, through the usage of the action identifiers. Action execution of the action array structure occurs in a depth-first fashion, until a terminating action, such as a forward or drop is encountered. For illustrative purposes, let's say that rule 3 is a match. Since the high-level action corresponding to rule 3 is a NOP, then there is nothing to do at the first stage, so we move on to its subaction. Here we execute action b1, and the subaction chain rooted at this node, namely actions c1, c2, and c3. Since there are no further subactions, we move back up a level and execute action b2 and its subaction chain, namely, d1 and d2. Finally, we execute b3 and b4, and transmit the packet.

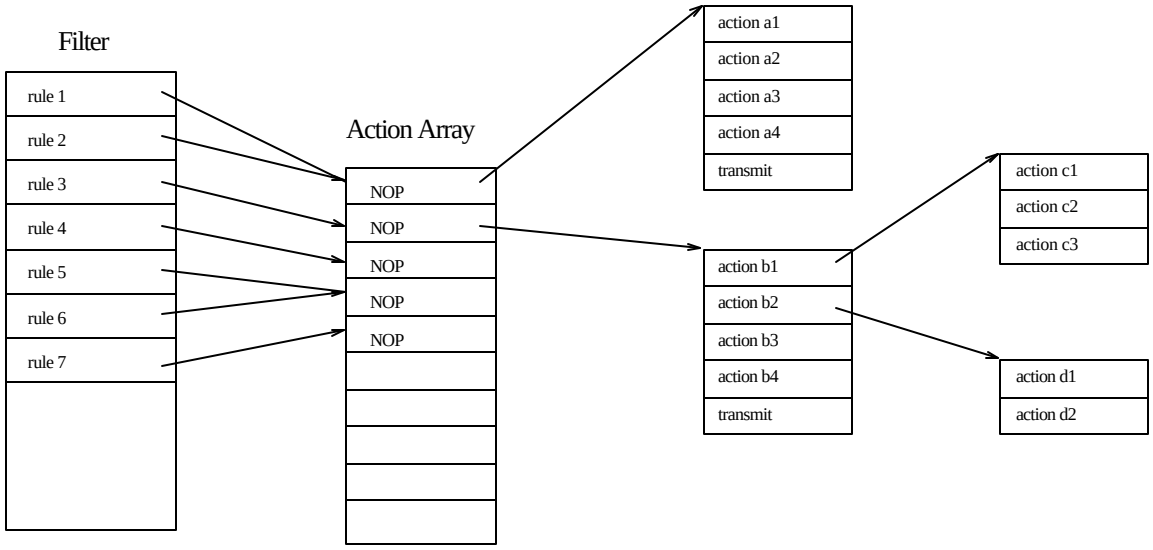

Figure 7: Action Representation

As we shall soon see, this multilevel action specification is very useful for specifying actions which lead to conditional branching, such as the policer instruction. Each branch can be represented by a pointer to a set of actions to execute if that branch is taken.

Now we will look at the contents of the individual action structures. First, we have the dummy action, represented by `NPF_DS_ActionNOP_t`. This action, as its name implies, takes no parameters and yields no processing to a matching packet.

```
/* NOP Action */
typedef struct {
   /* No parameters */
} NPF_DS_ActionNOP_t;
```

The type `NPF_DS_ActionCounter_t` is used to configure a counter for the given rule. This action contains the Counter Object's handle which was obtained by creating the QoS object of type `NPF_DS_QOS_COUNTER`. Using this handle, one can access the Counter Data Structure (see [3.1.22]) which holds values of the count of bytes and the count of packets that have passed through this counter. Since these are represented as 64 bit integers, overflow will not be an issue with today's flow of network traffic.

```
/* Counter Action */
typedef struct {
   NPF_DS_QoSHandle_t counterDataHandle;
} NPF_DS_ActionCounter_t;
```

The type `NPF_DS_ActionPolice_t` is used to configure a policer. This action contains the Policer Object's handle which was obtained by creating the QoS object of type `NPF_DS_QOS_POLICER`. The policer can take any of the three actions based on the color of the packet. If a packet conforms to the committed burst size, it is termed as a green packet and the green action chain will be executed. If a packet doesn't conform to the committed burst size but it conforms to the excess burst size, it is termed as a yellow packet and the yellow action chain will be taken. If a packet exceeds both the committed and excess burst sizes, it will be termed a red packet and the red action chain will be taken. This is illustrated in Figure 8 below. Finally, if the subaction of the policer is specified, then this subaction chain will be executed after any of the three conditional color branch action chains are completed.

```
/* Police Action */
typedef struct {
   NPF_DS_QoSHandle_t policerDataHandle;
   NPF_DS_Action_t    *actionGreen;
   NPF_uint16_t       actionGreenSize;
   NPF_DS_Action_t    *actionYellow;
   NPF_uint16_t       actionYellowSize;
   NPF_DS_Action_t    *actionRed;
   NPF_uint16_t       actionRedSize;
} NPF_DS_ActionPolice_t;
```

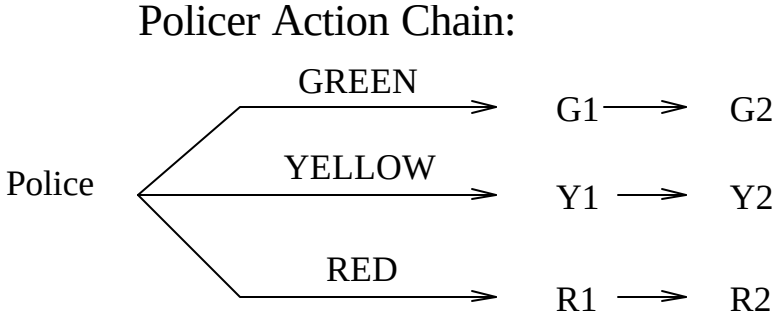

Figure 8: Policer Action Array Example

The types NPF_DS_ActionSetDSCP_t and NPF_DS_ActionSetIPPREC_t both merely specify the DSCP and IP Precedence values, respectively, that the matching packets are to be remarked with.

```
/* Set DSCP Action */
typedef struct {
   NPF_uchar8_t dscp;
} NPF_DS_ActionSetDSCP_t;


/* Set IP Precedence Action */
typedef struct {
  NPF_uchar8_t ipprec;
} NPF_DS_ActionSetIPPREC_t;
```

The SetNextHop action, represented by type NPF_DS_ActionSetNextHop_t, allows the specification of a prioritized list of routing alternatives. These alternatives consist of either next hop information or egress interface information. A packet will follow the first available entry in this list. Note, that this SetNextHop action will override any routing lookup done for the packet.

```
/* Set NextHop Action */
typedef struct {
   NPF_uint32_t numRouteListEntries;
   NPF_DS_RouteList_t *routeListArray;
} NPF_DS_ActionSetNextHop_t;
```

Each entry in the route list array can specify either next hop information or it can specify egress interface information, as shown in the following type definitions:

```
typedef struct {
   NPF_DS_RouteType_t routeType;
   union {
      NPF_IPv4Address_t nextHopIPv4;
      NPF_IPv6Address_t nextHopIPv6;
      NPF_IfHandle_t egressIfaceHandle;
   } routeInfo;
} NPF_DS_RouteList_t;

typedef enum {
   NPF_DS_ROUTE_TYPE_NEXT_HOP_IPV4    = 0,
   NPF_DS_ROUTE_TYPE_NEXT_HOP_IPV6    = 1,
   NPF_DS_ROUTE_TYPE_EGRESS_INTERFACE = 2
} NPF_DS_RouteType_t;
```

To specify a non-default routing table to be used by a packet in the route lookup stage, the SetFIB action can be used. Specified by type NPF_DS_SetFIB_t, this action allows the specification of a routing table identifier.

```
/* Set FIB Action */
typedef struct {
   NPF_IPv4UC_FibTableHandle_t fibTableHandle;
} NPF_DS_ActionSetFIB_t;
```

The action `SetColor` can be used to set a color for a particular packet. The color of the packet can affect its treatment in Color Aware policers.

```
/* Color States */
typedef enum {
   NPF_DS_GREEN    = 0,
   NPF_DS_YELLOW   = 1,
   NPF_DS_RED      = 2
} NPF_DS_COLOR_t;

/* Set Color Action */
typedef struct {
   NPF_DS_COLOR_t color;
} NPF_DS_ActionSetColor_t;
```

The `Downmark` action, typically used as an action of a policer, leads to the remarking of a packet according to a global downmark table. The type `NPF_DS_Downmark_t` associated with the downmark action requires a handle to the downmark table associated with this action.

```
/* Downmark Action */
typedef struct {
   NPF_DS_MapTableHandle_t downmarkTableHandle;
} NPF_DS_ActionDownmark_t;
```

As its name implies, the drop action leads to the dropping of all matching packets. The type `NPF_DS_ActionDrop_t` which is associated with the drop action has no parameters.

```
/* Drop Action */
typedef struct {
   /* no action parameters */
} NPF_DS_ActionDrop_t;
```

The shape action allows the configuration of a shaper for the matching packet stream. The type `NPF_DS_ActionShape_t` allows specification of the rate (expressed in bits per second) and burst (expressed in bytes) for the shaper. No actions can follow a drop action.

```
/* Shape action */
typedef struct {
   NPF_DS_QoSHandle_t shaperDataHandle;
} NPF_DS_ActionShape_t;
```

The forward action, which takes no parameters, merely forwards the packet as its name implies. No actions can follow a forward action.

```
/* Forward Action */
typedef struct {
   /* No parameters */
} NPF_DS_ActionForward_t;
```

The return action causes evaluation of rules to continue with the next policy element specified in the high-level policy element array. This action requires no parameters.

```
/* Return Action */
typedef struct {
   /* No parameters*/
} NPF_DS_ActionReturn_t;
```

The queue action allows the specification of a queue in which this packet will be enqueued.

```
/* Queue action */
typedef struct {
   NPF_DS_QoSHandle_t queueHandle;
} NPF_DS_ActionQueue_t;
```

The set PHB Scheduling Class (PSC) action allows the specification of a Per Hop Scheduling Class in which this packet will be enqueued. It is assumed that these PSCs will be created once the system is initialized.

```
/* PHB Scheduling Classes */
typedef enum{
   NPF_DS_PSC_EF   = 0,
   NPF_DS_PSC_AF1  = 1,
   NPF_DS_PSC_AF2  = 2,
   NPF_DS_PSC_AF3  = 3,
   NPF_DS_PSC_AF4  = 4,
   NPF_DS_PSC_BE   = 5
} NPF_DS_PSC_Type_t;


/* Set PHB Scheduling Class (PSC) action */
typedef struct {
   NPF_DS_PSC_Type_t pscType;
   NPF_DS_QoSHandle_t pscHandle; /* similar to the queue
                                    handle */
} NPF_DS_ActionSetPSC_t;
```

The action `SetTrust` can be used to set a local trust state on a per-policy basis. The trust state is used to determine how a trusted DSCP value is to be obtained for a packet. Although this is not an action which directly impacts a packet, it can set a mode which may lead to possible remarking of the packet. This trust state will override any global trust state set on the interface. All the possible trust states available are enumerated in `NPF_DS_TRUST_t.` In the case of `NPF_DS_TRUST_UNTRUSTED`, the trusted-DSCP is determined through use of the interface Class of Service (COS) value, mapped to a DSCP using the COS-to-DSCP mapping table. For `NPF_DS_TRUST_TRUSTDSCP`, we adopt the packet's DSCP value as the trusted DSCP value. In the case of `NPF_DS_TRUST_TRUSTIPPREC`, the packet's IP Precedence value is used to map to the trusted DSCP value using the IPPrec-to-DSCP mapping table. In the case of `NPF_DS_TRUST_TRUSTCOS`, the layer 2 COS value is used to determine the trusted DSCP value using the L2COS-to-DSCP mapping table. Finally, in the case of `NPF_DS_TRUST_UNSPEC`, the local trust state is unspecified, and we fall back to the trusted DSCP calculated using the global trust state.

```
/* Trust States */
typedef enum {
   NPF_DS_TRUST_UNTRUSTED    = 0,
   NPF_DS_TRUST_TRUSTDSCP    = 1,
   NPF_DS_TRUST_TRUSTIPPREC  = 2,
   NPF_DS_TRUST_TRUSTCOS     = 3,
```

```
   NPF_DS_TRUST_UNSPEC       = 4
} NPF_DS_TRUST_STATE_t;


/* Set Local Trust Mode Action */
typedef struct {
   NPF_DS_TRUST_STATE_t trust;
} NPF_DS_ActionSetTrust_t;
```

3.1.6.1 Embedded QoS Object Actions in Policy Objects

Policy Objects have rules and actions associated with them. The actions can be simple set actions where there is no need to create other objects like QoS Objects. This type of actions are as follows:

- Set DSCP
- Set IPPREC
- Set COLOR
- Set TRUST
- Set PSC
- Set NEXTHOP
- DROP
- FORWARD
- NOP

But for other set actions, there are objects which carry the corresponding parameters which is used to process the packets accordingly. Either these objects are created using Diffserv SAPI (e.g. QoS Objects) or other SAPIs (IPv4 SAPI for FIBs) and their handle is used to bind them to the policy objects. Please refer to Figure 9 for illustration. Examples of these actions are as follows:

- Set FIB
- Set QUEUE
- POLICE
- SHAPE
- Downmark

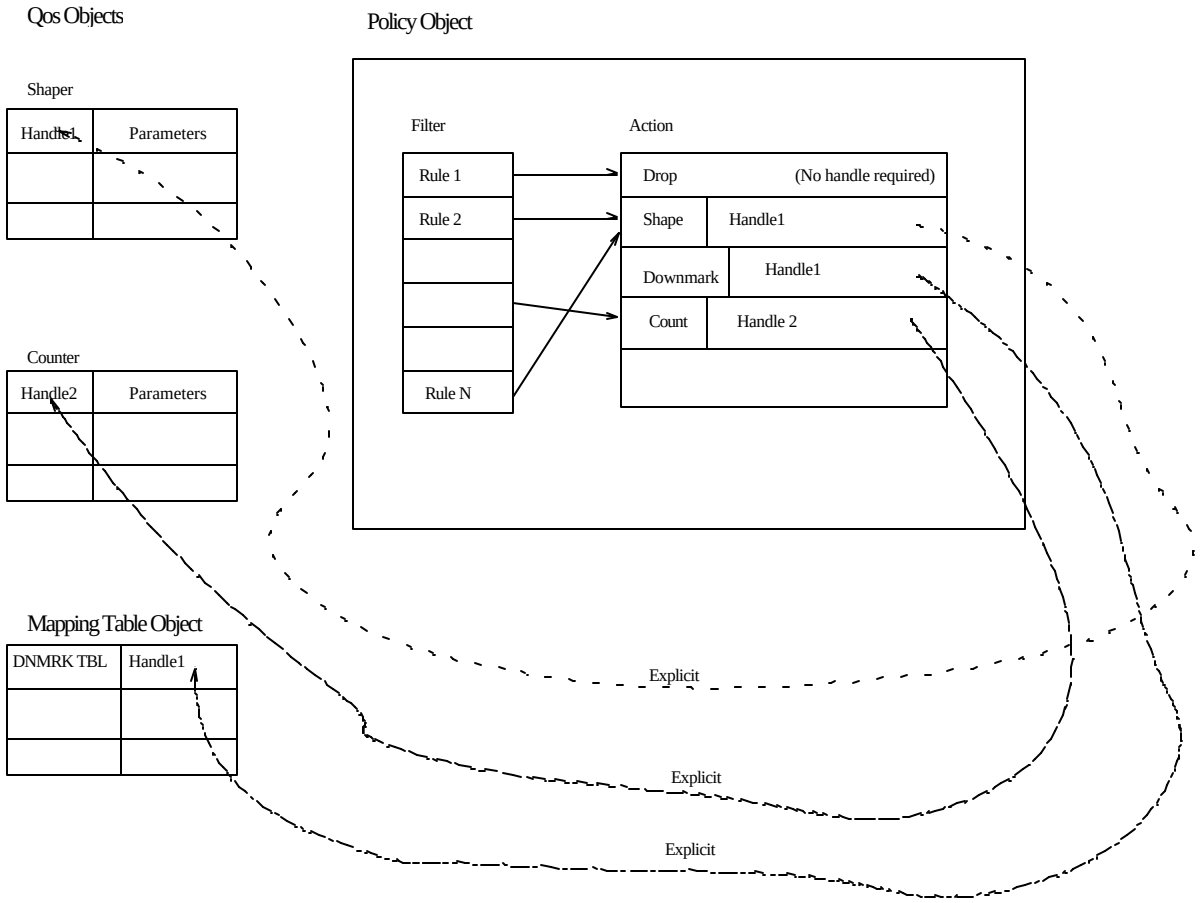

Figure 9: Policy Objects and other Object binding

3.1.7 Explicit Congestion Notification

The Explicit Congestion Notification (ECN) [RFC3168] capability on a router can be turned on/off through the hooks provided by the Diffserv application. This section provides the needed support for upper applications to do so. An application can configure the Queue Objects depending upon the mode of the ECN configuration (see [3.1.21]).

```
/* ECN States */
typedef enum {
   NPF_DS_ECN_UNAWARE = 0,
   NPF_DS_ECN_AWARE   = 1
} NPF_DS_ECN_MODE_t;


/* Set GLOBAL ECN Mode Action */
typedef struct {
   NPF_DS_ECN_MODE_t ecn;
} NPF_DS_ECN_t;
```

3.1.8 Mapping Table Types

Mapping tables which are registered with the Diffserv SAPI may be specified as type `NPF_DS_MapTableType_t`, and can be any of the following values.

```
typedef enum {
   NPF_DS_DSCP_L2COS    = 0,
```

```
   NPF_DS_L2COS_DSCP    = 1,
   NPF_DS_DOWNMARK_DSCP = 2,
   NPF_DS_DSCP_PHB      = 3,
   NPF_DS_IPPREC_DSCP   = 4
} NPF_DS_MapTableType_t;
```

3.1.9 Mapping Table Data Block

The actual mapping tables being registered with the Diffserv SAPI are done so with the type `NPF_DS_MapTableData_t`. As seen in the type definition, this consists of a union of all of the various mapping table arrays. The SAPI knows exactly which table to use since this is specified along with the aforementioned `NPF_DS_MapTableType_t`. The sizes of each of the tables is given as macros preceding the structure definition.

```
#define NPF_DS_MAPTABLE_L2COS_DSCP_MAP_SIZE      8
#define NPF_DS_MAPTABLE_DOWNMARK_DSCP_MAP_SIZE  64
#define NPF_DS_MAPTABLE_DSCP_PHB_MAP_SIZE       64
#define NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE     64
#define NPF_DS_MAPTABLE_IPPREC_DSCP_MAP_SIZE     8
```

```
typedef union {
   NPF_uint16_t
       l2cos_dscp_map[NPF_DS_MAPTABLE_L2COS_DSCP_MAP_SIZE];
   NPF_uint16_t
       downmark_dscp_map
           [NPF_DS_MAPTABLE_DOWNMARK_DSCP_MAP_SIZE];
   NPF_uint16_t
       dscp_phb_map[NPF_DS_MAPTABLE_DSCP_PHB_MAP_SIZE];
   NPF_uint16_t
       dscp_l2cos_map[NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE];
   NPF_uint16_t
      ipprec_dscp_map[NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE];
} NPF_DS_MapTableData_t;
```

3.1.10 Mapping Table Object Structure

Here we have the type definition for a mapping table specified along with its type and scope parameters.

```
typedef struct {
   NPF_DS_MapTableType_t type;
   NPF_DS_MapTableData_t map;
} NPF_DS_MapTable_t;
```

3.1.11 QoS Object Types

QoS Objects which are managed by Diffserv SAPI may be specified as type `NPF_DS_QoSObjectType_t`, and can be any of the following values.

```
typedef enum {
   NPF_DS_QOS_POLICER          = 0,
   NPF_DS_QOS_SHAPER           = 1,
   NPF_DS_QOS_COUNTER          = 2,
   NPF_DS_QOS_SCHEDULER        = 3,
```

```
   NPF_DS_QOS_SCHEDULER_CONFIG = 4,
   NPF_DS_QOS_QUEUE            = 5,
   NPF_DS_QOS_TRUST_STATE      = 6
} NPF_DS_QoSObjectType_t;
```

3.1.12 Active Queue Management

AQM can be enabled on a QoS object of type `NPF_DS_QOS_QUEUE`. The following AQM types are supported in this specification.

```
typedef enum {
   NPF_DS_AQM_HEAD   = 0,
   NPF_DS_AQM_TAIL   = 1,
   NPF_DS_AQM_RED    = 2,
   NPF_DS_AQM_WRED   = 3
} NPF_DS_AQMType_t;
```

3.1.13 Scheduler Config Object Types

Depending on the Hardware capability, it should be possible to choose any one of the following queuing discipline types. Scheduler Config Data Types are based on them. They are taken from the NPF Traffic Queuing FAPI.

```
typedef enum {
    NPF_DS_PQ    = 0, /* Priority Queuing */
    NPF_DS_CBQ   = 1, /* Class Based Queuing */
    NPF_DS_CBWFQ = 2, /* Class Based Weighted
                         Fair Queuing */
} NPF_DS_SchedulerConfigType_t;
```

3.1.14 Scheduler Object Types

Depending on the Hardware capability, it should be possible to choose any one of the following scheduling discipline types. They are taken from the NPF Traffic Queuing FAPI.

```
typedef enum {
    NPF_DS_PRI   = 0,
    NPF_DS_BSP   = 1,
    NPF_DS_WFQ   = 2,
    NPF_DS_WRR   = 3,
    NPF_DS_TB    = 4,
    NPF_DS_DRR   = 5
} NPF_DS_SchedulerType_t;
```

3.1.15 Policer Types

The policer can be of the following types.

```
typedef enum {
   NPF_DS_POLICER_SRTCM_COLOR_BLIND = 0,
   NPF_DS_POLICER_SRTCM_COLOR_AWARE = 1,
   NPF_DS_POLICER_TRTCM_COLOR_BLIND = 2,
   NPF_DS_POLICER_TRTCM_COLOR_AWARE = 3
} NPF_DS_PolicerType_t;
```

3.1.16 Policer QoS Object Data Block

The Policer Data Block is the structure which contains a policer type and various parameters associated with this type.

```
typedef struct {
    NPF_DS_PolicerType_t type;
    NPF_uint32_t cir; /* Committed Info Rate */
    NPF_uint32_t pir; /* Peak Info Rate */
    NPF_uint32_t cbs; /* Committed Burst Size */
    NPF_uint32_t ebs; /* Excess Burst Size */
} NPF_DS_PolicerData_t;
```

3.1.17 Shaper QoS Object Data Block

The Shaper Data block contains the various parameters associated with it.

```
typedef struct {
    NPF_uint32_t cir; /* Committed Info Rate */
    NPF_uint32_t cbs; /* Committed Burst Size */
    NPF_uint32_t pir; /* Peak Rate */
    NPF_uint32_t pbs; /* Peak burst */
} NPF_DS_ShaperData_t;
```

3.1.18 Scheduler Config QoS Object Data Block

The Scheduler Config (SC) QoS data block contains various parameters associated with the scheduling policy applied to a Queue object. Depending on the Scheduler Config Data block type, the SC Data block will have its corresponding data structure.

```
typedef struct {
    NPF_uint16_t prio; /* Priority */
} NPF_DS_SchedulerConfigPrio_t;

typedef struct {
    NPF_uint16_t prio; /* Priority */
    NPF_uint32_t pir;  /* Rate to be bounded on */
} NPF_DS_SchedulerConfigBSP_t;

typedef struct {
    NPF_uint32_t weight;  /* weight */
} NPF_DS_SchedulerConfigWRR_t;

typedef struct {
    NPF_uint32_t weight;  /* weight */
} NPF_DS_SchedulerConfigWFQ_t;

typedef struct {
    NPF_uint32_t pir;     /* Peak Rate */
    NPF_uint32_t cir;     /* Committed Rate */
    NPF_uint32_t pbs;     /* Peak burst */
    NPF_uint32_t cbs;     /* Committed Burst */
} NPF_DS_SchedulerConfigTB_t;

typedef struct {
    NPF_DS_SchedulerConfigType_t configType;
    union {
```

```
            NPF_DS_SchedulerConfigPrio_t configPrio;
            NPF_DS_SchedulerConfigBSP_t configBSP;
            NPF_DS_SchedulerConfigWRR_t configWRR;
            NPF_DS_SchedulerConfigWFQ_t configWFQ;
            NPF_DS_SchedulerConfigTB_t configTB;
        } configParam;
} NPF_DS_SchedulerConfigData_t;
```

3.1.19 Scheduler Object Data Block

The Scheduler data block contains the following parameters. Every creation of Scheduler objects will result in a handle and Queue objects will refer to this Scheduler object using this handle [3.1.21]. Schedulers created with a NULL `downstreamHandle` will be the rightmost scheduling entity on the interface.

```
typedef struct {
   NPF_DS_SchedulerType_t type;
   NPF_uint32_t minRate;                  /* Minimum Rate */
   NPF_uint32_t maxRate;                  /* Maximum Rate */
   NPF_DS_QoSHandle_t downstreamHandle;  /* Handle for the
                                            Downstream
                                            scheduler */
NPF_DS_SchedulerConfigData_t configData; /* Config
                                            Parameters
                                            for downstream
                                            link */
} NPF_DS_SchedulerData_t;
```

3.1.20 AQM Statistics Object Data Block

This structure holds counter statistics for AQM.

```
typedef struct {
   NPF_uint64_t bytes;
   NPF_uint64_t packets;
   NPF_uint64_t randomBytes;
   NPF_uint64_t randomPackets;
} NPF_DS_AQMData_t;
```

3.1.21 Queue QoS Object Data Block

This structure contains an AQM type and parameters related to the this type. It also contains the buffer size of the queue object.

```
typedef struct {
    NPF_DS_AQMType_t type;
    NPF_uint32_t bufferSize;
    NPF_uint32_t minThresh;
    NPF_uint32_t maxThresh;
    NPF_uint16_t maxProb;
    NPF_uint16_t avgWeight;
    NPF_DS_AQMData_t stats;
    NPF_DS_ECN_t ecnMode;
    NPF_DS_SchedulerConfigData_t *schedConfigData;
    NPF_DS_QoSHandle_t schedulerHandle;
} NPF_DS_QueueData_t;
```

3.1.22 Counter Object Data Block

This structure holds counter statistics for the counter action associated with a policy.

```
typedef struct {
   NPF_uint64_t bytes;
   NPF_uint64_t packets;
} NPF_DS_CounterData_t;
```

3.1.23 QoS Object Data Block

QoS object Data block is a union of various QoS object data blocks.

```
typedef union {
    NPF_DS_PolicerData_t policer;
    NPF_DS_ShaperData_t shaper;
    NPF_DS_SchedulerData_t scheduler;
    NPF_DS_CounterData_t counter;
    NPF_DS_QueueData_t queue;
    NPF_DS_TrustStateData_t trust;
} NPF_DS_QoSObjectData_t;
```

3.1.24 QoS Object Structure

QoS Object structure contains a QoS Object Type and configuration parameters for the particular QoS object type.

```
typedef struct {
   NPF_DS_QoSObjectType_t type;
   NPF_DS_QoSObjectData_t data;
} NPF_DS_QoSObject_t;
```

3.1.25 Counter Statistics Data Block

The statistics query returns an array of counter values, one for each counter registered as an action, in the same order as they appeared in the original action list.

```
typedef struct {
   NPF_uint32_t numCounters;
   NPF_DS_CounterData_t *counterData;
} NPF_DS_CounterStatsData_t;
```

3.1.26 AQM Statistics Data Block

The AQM statistics query returns an array of AQM counter values.

```
typedef struct {
   NPF_uint32_t num;
   NPF_DS_AQMData_t *AQMCounters;
} NPF_DS_AQMStatsData_t;
```

3.1.27 Trust State Object Data Block

This object is used to set the trust state on the interface. This trust state can be overridden by applying a new trust state as part of the set trust action as explained in section 3.1.6.

The structure for this Objects is as follows

```
typedef struct {
   NPF_DS_ TRUST_STATE_t trust;
} NPF_DS_TrustStateData_t;
```

3.1.28 Capability Discovery

Due to the inherent differences in the design of underlying system from different vendors, the data plane processing capabilities can differ between two different vendor's products. To this end, we provide data structures and accompanying query functions to discover the capabilities of a given underlying system

3.1.28.1 Diffserv Capability Feature Specification

The following set of data structures provide a representation of all of the basic features of the Diffserv SAPI in the form of a bit mask. A value of 1 for any given field represents an underlying system that supports the given feature. Conversely, a value of 0 for any given field represents an underlying system that does not support the given feature.

The type `NPF_DS_InterfaceCapabilities_t`, as shown below, represents the capabilities of a given underlying system. These capabilities are subdivided into capabilities of each plane that makes up the interface (i.e., the layer 2 encapsulation, layer 2 decapsulation, IP ingress, IP egress, local traffic termination, and local traffic generation planes). First, the filter and action capabilities are given. Next, the four planes that can have queueing present have the queueing/scheduling capabilities given. Next, the mapping table capabilities for the interface are given. Finally, a bit vector provides the necessary switches to communicate the actual existence of the various planes in the underlying hardware.

```
typedef struct {
     /* Filter & Action Capabilities */
     NPF_DS_L2_Encap_PlaneCapabilities_t
                  FilterActionCapL2_encap;
     NPF_DS_L2_Decap_PlaneCapabilities_t
                  FilterActionCapL2_decap;
     NPF_DS_L3_PlaneCapabilities_t
                   FilterActionCapIP_ingress;
     NPF_DS_L3_PlaneCapabilities_t
                   FilterActionCapIP_egress;
     NPF_DS_L3_PlaneCapabilities_t
                  FilterActionCapLocallyTerminated;
     NPF_DS_L3_PlaneCapabilities_t
                  FilterActionCapLocallyGenerated;

     /* Queuing Capabilities */
     NPF_DS_SchedulingCapabilities_t QueueCapIP_ingress;
     NPF_DS_SchedulingCapabilities_t QueueCapIP_egress;
     NPF_DS_SchedulingCapabilities_t
                  QueueCapLocallyTerminated;
     NPF_DS_SchedulingCapabilities_t
                  QueueCapLocallyGenerated;


     /* Mapping Table Capabilities */
     NPF_DS_MappingTablesCapabilities_t MappingTablesCap;

     /* Plane existence */
     NPF_uchar8_t planeL2_Encap:1,
                  planeL2_Decap:1,
                 planeL3_Ingress:1,
```

```
                    planeL3_Egress:1,
                    planeL3_LocallyTerminated:1,
                    planeL3_LocallyGenerated:1;
} NPF_DS_InterfaceCapabilities_t;
```

Now, let us look at the filter and action capabilities for the layer 3 (IP Ingress and IP Egress) planes, represented by `NPF_DS_L3_PlaneCapabilities_t`. This data structure provides the filtering capabilities for both IPv4 and IPv6, as well as layer 4 TCP/UDP and ICMP. Next, the action capabilities are represented in type `NPF_DS_L3_ActionCapabilities_t`. Finally, three overflow bits are specified. The first of the three bits represent the situation where if a child interface on the interface hierarchy does not have a policy set, it should inherit the policy of the parent interface. The second bit represents the situation where if a child interface does have a policy set, but there is no filter match, the parent interface's policy must be applied. Finally, the third bit represents the situation where if a child interface does have a policy set, and there is a filter match, then the parents policy table should also be applied. These three overflow bits will be configurable independently on all planes of the interface.

```
typedef struct {
     NPF_DS_L3_IPv4_FilterCapabilities_t
               capabilityFilterL3_IPv4;
     NPF_DS_L3_IPv6_FilterCapabilities_t
               capabilityFilterL3_IPv6;
     NPF_DS_L4_TCPUDP_FilterCapabilities_t
               capabilityFilterL4_TCPUDP;
     NPF_DS_L4_ICMP_FilterCapabilities_t
               capabilityFilterL4_ICMP;
     NPF_DS_L3_ActionCapabilities_t  capabilityActions;

     /* Overflow bits */

        /* if there is no policy set on a subordinate, the
         * policy of the next higher level interface
         * must be checked/used
         */
     NPF_uchar8_t inheritParentPolicy             :1,

        /* even if there is a policy set on a subordinate,
         * if the packet does not match any policy in the
         * policy table, then the parent interface's policy
         * table must be applied.
         */
                 inheritParentPolicyNoMatch      :1,

        /* same as the previous, but also allow to goto the
         * parents policy even if the packet already
         * matched something in the local policy table.
         */
                  inheritParentPolicyAlreadyMatch  :1;
} NPF_DS_L3_PlaneCapabilities_t;
```

Layer 2 filter capabilities consider only those fields that are applicable to layer 2, as specified in `NPF_DS_L2_FilterCapabilities_t`. The action capabilities are restricted to only those actions which make sense in the layer 2 encapulation and decapsulation planes, as specified in `NPF_DS_L2_Encap_ActionCapabilities_t` and `NPF_DS_L2_Decap_ActionCapabilities_t`, respectively. Finally, the overflow bits, as previously described, are specified for each of these planes.

```
typedef struct {
     NPF_DS_L2_FilterCapabilities_t
                  capabilityFilterL2;
     NPF_DS_L2_Encap_ActionCapabilities_t
                  capabilityActions;

           /* Overflow bits */
     NPF_uchar8_t inheritParentPolicy              :1,
                  inheritParentPolicyNoMatch       :1,
                  inheritParentPolicyAlreadyMatch  :1;
} NPF_DS_L2_Encap_PlaneCapabilities_t;


typedef struct {
     NPF_DS_L2_FilterCapabilities_t
                 capabilityFilterL2;
     NPF_DS_L2_Decap_ActionCapabilities_t
                 capabilityActions;

           /* Overflow bits */
     NPF_uchar8_t inheritParentPolicy              :1,
                  inheritParentPolicyNoMatch       :1,
                   inheritParentPolicyAlreadyMatch  :1;
} NPF_DS_L2_Decap_PlaneCapabilities_t;
```

The scheduling and mapping table capabilities are specified in `NPF_DS_SchedulingCapabilities_t` and `NPF_DS_MappingTablesCapabilities_t`, respectively.

```
typedef struct {
    NPF_uchar8_t SP  :1,
                 WFQ :1,
                 WRR :1,
                 TB  :1;
} NPF_DS_SchedulingCapabilities_t;


typedef struct {
     NPF_uchar8_t L2COS_to_DSCP :1,
                  L2COS_to_PHB  :1,
                 DSCP_to_L2COS :1,
                 DSCP_to_PHB   :1,
                 IPPREC_TO_DSCP:1,
                 DSCP_TO_IPPREC:1;
} NPF_DS_MappingTablesCapabilities_t;
```

The following three data structures communicate the action processing capabilities of each of the corresponding packet processing planes for the underlying system. Only those actions that make sense for the given plane are configurable.

```
typedef struct {
     NPF_uchar8_t NOP       :1,
                  counter   :1,
```

```
                    policer   :1,
                    setDSCP   :1,
                    setIPREC  :1,
                    setNextHop:1,
                    setFIB    :1,
                    setColor  :1,
                    downmark  :1,
                    drop      :1,
                    shape     :1,
                    forward   :1,
                    queue     :1,
                    setPSC    :1,
                    setTrust  :1,
                    retrn     :1;
} NPF_DS_L3_ActionCapabilities_t;

typedef struct {
     NPF_uchar8_t NOP       :1,
                  counter   :1,
                  policer   :1,
                  setNextHop:1,
                  setFIB    :1,
                  setColor  :1,
                  downmark  :1,
                  drop      :1,
                  shape     :1,
                  forward   :1,
                  queue     :1,
                  setPSC    :1,
                  retrn     :1;
} NPF_DS_L2_Encap_ActionCapabilities_t;


typedef struct {
     NPF_uchar8_t NOP       :1,
                  counter   :1,
                  policer   :1,
                  setDSCP   :1,
                  setIPREC  :1,
                  setNextHop:1,
                  setFIB    :1,
                  setColor  :1,
                  downmark  :1,
                  drop      :1,
                  forward   :1,
                  queue     :1,
                  setPSC    :1,
                  setTrust  :1,
                  retrn     :1;
} NPF_DS_L2_Decap_ActionCapabilities_t;
```

The layer 4 filterable field capabilities are communicated through the following data structures. For TCP and UDP, the source and destination ports are independent. Furthermore, the type of filtering allowed for these fields is specified by SRCP_range and DSTP_range, respectively. If the SRCP bit is set and the SRCP_range bit is set, then the underlying system is capable of filtering on a range of source ports, in addition to filtering on a singular port or wildcard. If the SRCP bit is set and the SRCP_range bit is not set, then only singular port filtering or wildcarding is capability is available in the underlying system. Finally, if the SRCP bit is not set, then no source port filtering is allowed, and the value of the SRCP_range bit is irrelevant. The destination port bits work in the same fashion.

```
typedef struct {
     NPF_uchar8_t SRCP      :1,
                  SRCP_range:1,
                  DSTP      :1,
                  DSTP_range:1,
                  TCP_flags :1;
} NPF_DS_L4_TCPUDP_FilterCapabilities_t;
```

The ICMP header filtering capability can indicate whether or not filtering is allowed on the ICMP code and type fields as shown in the following data structure.

```
typedef struct {
     NPF_uchar8_t type:1,
                  code:1;
} NPF_DS_L4_ICMP_FilterCapabilities_t;
```

The layer 2 filtering capability structure allows communication of filtering capabilities for all those filterable fields relevant to layer 2 (i.e., MAC address, priority, and VLAN identifier).

```
typedef struct {
     NPF_uchar8_t MAC_addr:1,
                  priority:1,
                  VLAN_id :1;
} NPF_DS_L2_FilterCapabilities_t;
```

The layer 3 IPv4 filtering capability structure allows communication of filtering capabilities for all those layer 3 IPv4 filterable fields (i.e., protocol, IPSA, IPDA, TOS byte, and fragments). Additionally, for IPSA and IPDA, the type of filtering available (i.e., can we filter based upon an address prefix?) is specifiable. If IPSA is set and IPSA_prefix is not set, then the underlying system can only filter upon fully-specified addresses or wildcards. If IPSA is set and IPSA_prefix is also set, then the underlying system can filter upon fully-specified addresses or wildcards, as before, but it can also filter using an address prefix. If IPSA is not set, then no source address filtering is available and the value of the IPSA_prefix bit is irrelevant. The same applies for IPDA and IPDA_prefix.

```
typedef struct {
     NPF_uchar8_t IPSA       :1,
                  IPSA_prefix:1,
                  IPDA       :1,
                  IPDA_prefix:1,
                  protocol   :1,
                  TOS_byte   :1,
                  fragments  :1;
} NPF_DS_L3_IPv4_FilterCapabilities_t;
```

Finally, the layer 3 IPv6 filtering capabilities is specified in the following data structure, which communicates the filtering capabilities of the underlying system relevant to layer 3, IPv6. Similar to the case in IPv4, the source address and destination address contain an additional `_prefix` bit which specifies whether filtering can be performed using an address prefix.

```
typedef struct {
     NPF_uchar8_t IPSA       :1,
                  IPSA_prefix:1,
```

```
                    IPDA       :1,
                    IPDA_prefix:1,
                    class      :1,
                     flowLabel  :1,
                    nextHeader :1;
} NPF_DS_L3_IPv6_FilterCapabilities_t;
```

3.1.28.2 Capability Profiles

Since in any given system, there will likely be a large number of interfaces, and only a small number of unique capability matrices, we will identify these unique capability matrices with a profile identifier. Thus, any given interface will have a particular profile identifier associated with it. Profile identifiers, assigned by the SAPI provider, will have the following type definition:

```
typedef NPF_uint32_t NPF_DS_CapabilityProfileID_t;
```

We would like to have a query function to return the set of capability profiles that currently exist in the system. These profiles, denoted by profile identifiers, are returned in an array. The elements of this array will take the following form, in which a profile identifier is paired with a interface capability matrix.

```
typedef struct {
   NPF_DS_CapabilityProfileID_t profileID;
   NPF_DS_InterfaceCapabilities_t capabilityMatrix;
} NPF_DS_CapabilityProfileInfo_t;
```

3.2 Return codes

The codes defined here will be used for returns from asynchronous API function calls.

```
/* Asynchronous error codes (returned in function
callbacks) */

typedef NPF_uint32_t NPF_DS_ReturnCode_t;

#define S_DS_ERR(n) ((NPF_DS_ReturnCode_t) \
                          (NPF_DS_BASE_ERR+(n))


#define NPF_DS_POLICY_INVALID_HANDLE          S_DS_ERR(0)
#define NPF_DS_POLICY_INVALID_INPUT_PARAM     S_DS_ERR(1)
#define NPF_DS_POLICY_BOUND                   S_DS_ERR(2)
#define NPF_DS_POLICY_BOUND_IFACE             S_DS_ERR(3)
#define NPF_DS_POLICY_NOT_BOUND               S_DS_ERR(4)
#define NPF_DS_POLICY_NOT_BOUND_IFACE         S_DS_ERR(5)
#define NPF_DS_POLICY_DOES_NOT_EXIST          S_DS_ERR(6)
#define NPF_DS_MAPTABLE_INVALID_HANDLE        S_DS_ERR(7)
#define NPF_DS_MAPTABLE_INVALID_INPUT_PARAM   S_DS_ERR(8)
#define NPF_DS_MAPTABLE_BOUND                 S_DS_ERR(9)
#define NPF_DS_MAPTABLE_BOUND_IFACE           S_DS_ERR(10)
#define NPF_DS_MAPTABLE_NOT_BOUND             S_DS_ERR(11)
#define NPF_DS_MAPTABLE_NOT_BOUND_IFACE       S_DS_ERR(12)
#define NPF_DS_MAPTABLE_DOES_NOT_EXIST        S_DS_ERR(13)
#define NPF_DS_QOS_OBJECT_INVALID_HANDLE      S_DS_ERR(14)
```

```
#define NPF_DS_QOS_OBJECT_INVALID_INPUT_PARAM S_DS_ERR(15)
#define NPF_DS_QOS_OBJECT_BOUND               S_DS_ERR(16)
#define NPF_DS_QOS_OBJECT_BOUND_IFACE         S_DS_ERR(17)
#define NPF_DS_QOS_OBJECT_NOT_BOUND           S_DS_ERR(18)
#define NPF_DS_QOS_OBJECT_NOT_BOUND_IFACE     S_DS_ERR(19)
#define NPF_DS_QOS_OBJECT_DOES_NOT_EXIST      S_DS_ERR(20)
```

4 Diffserv SAPI Calls

4.1 Asynchronous Response Structure

The Asynchronous Response data structure is used during callbacks in response to API invocations. An asynchronous response contains an error/success code, other optional information that correlates the response to an element in a request array, and in some cases a function-specific structure embedded in a union. One or more of these are passed to the callback function as an array within the `NPF_DS_CallbackData_t` data structure.

4.1.1 Policy Object Configure Response Structure

```
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
} NPF_DS_PolicyCreateResponse_t;
```

4.1.2 Table Configure Response Structure

```
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
} NPF_DS_MapTableCreateResponse_t;
```

4.1.3 QoS Object Configure Response Structure

```
typedef struct {
   NPF_DS_QoSHandle_t qosHandle;
} NPF_DS_QoSObjectCreateResponse_t;
```

4.1.4 Policy Update Response Structure

```
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_DS_PolicyID_t resourceID;
} NPF_DS_PolicyUpdateResponse_t;
```

4.1.5 Table Update Response Structure

```
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableID_t resourceID;
} NPF_DS_MapTableUpdateResponse_t;
```

4.1.6 QoS Object Update Response Structure

```
typedef struct {
   NPF_DS_QoSHandle_t qosHandle;
   NPF_DS_QoSObjectID_t resourceID;
} NPF_DS_QoSObjectUpdateResponse_t;
```

4.1.7 Policy Object Get Handle Query Response Structure

The Policy Object Get Handle query function returns with an array (size = count) of policy handles and their associated resource ids. If `configureFlag = TRUE,` then the object was configured by Diffserv client else it is the DEFAULT object.

When the default object is passed back, the query structure will have resource id as defined below. All APIs would return an error when passed this resource id for "create" APIs. Application clients should make sure that negative resource identifiers are not passed across the SAPI.

```
#define NPF_DS_QUERY_DEFAULT_OBJECT_RESOURCE_ID   -1


typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_DS_PolicyID_t resourceID;
   NPF_char8_t configureFlag;
} NPF_DS_PolicyQueryGetHandleInfoResponse_t;



typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_PolicyQueryGetHandleResponse_t;
```

4.1.8 Policy Object Get Contents Query Response Structure

The Policy Object Get Contents query function returns with an array (size = count) of policy contents (Rules and Actions).

```
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_uint32_t filterSize;
   NPF_DS_Rule_t *filter;
   NPF_uint32_t actionArraySize;
   NPF_DS_Action_t *actions;
} NPF_DS_PolicyQueryGetContentInfoResponse_t;



typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_PolicyQueryGetContentResponse_t;
```

4.1.9 Policy Get Bound Objects Query Response Structure

The Policy Get Bound Objects query function returns with an array (size = count) of policy objects bound on an interface and a plane.

```
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
} NPF_DS_PolicyQueryGetBoundObjectInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
      NPF_IfHandle_t ifaceHandle;
   NPF_DS_FilterPlane_t plane;
} NPF_DS_PolicyQueryGetBoundObjectResponse_t;
```

4.1.10 QoS Object Get Handle Query Response Structure

The QoS Object Get Handle query function returns with an array (size = count) of QoS Object handles and their associated resource ids. If `configureFlag = TRUE,` then the object was configured by Diffserv client else it is the DEFAULT object. QoS Object Type is also returned in the response structure.

```
typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectID_t resourceID;
   NPF_DS_QoSObjectType_t type;
   NPF_char8_t configureFlag;
} NPF_DS_QoSObjectQueryGetHandleInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_QoSObjectQueryGetHandleResponse_t;
```

4.1.11 QoS Object Get Contents Query Response Structure

The QoS Object Get Contents query function returns with an array (size = count) of object contents

```
typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectData_t data;
   NPF_DS_QoSObjectType_t type;
} NPF_DS_QoSObjectQueryGetContentInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_QoSObjectQueryGetContentResponse_t;
```

4.1.12 QoS Get Bound Objects Query Response Structure

The QoS Get Bound Objects query function returns with an array (size = count) of QoS objects bound on an interface and a plane.

```
typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectType_t type;
} NPF_DS_QoSObjectQueryGetBoundObjectInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
      NPF_IfHandle_t ifaceHandle;
   NPF_DS_FilterPlane_t plane;
} NPF_DS_QoSObjectQueryGetBoundObjectResponse_t;
```

4.1.13 Map Table Object Get Handle Query Response Structure

The Map Table Object Get Handle query function returns with an array (size = count) of Mapping Table handles and their associated resource ids. If `configureFlag = TRUE,` then the object was configured by Diffserv client else it is the DEFAULT object.

```
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableID_t resourceID;
   NPF_DS_MapTableType_t tableType;
   NPF_char8_t configureFlag;
} NPF_DS_MapTableQueryGetHandleInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_MapTableQueryGetHandleResponse_t;
```

4.1.14 Map Table Object Get Contents Query Response Structure

The Mapping Table Object Get Contents query function returns with an array (size = count) of table contents.

```
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableType_t type;
   NPF_DS_MapTableData_t data;
} NPF_DS_MapTableQueryGetContentInfoResponse_t;
```

```
typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_MapTableQueryGetContentResponse_t;
```

4.1.15 Map Table Get Bound Objects Query Response Structure

Map Table Get Bound Objects query function returns with an array (size = count) of table objects bound on an interface.

```
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableType_t type;
} NPF_DS_MapTableQueryGetBoundObjectInfoResponse_t;



typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
      NPF_IfHandle_t ifaceHandle;
} NPF_DS_MapTableQueryGetBoundObjectResponse_t;
```

4.1.16 Capability Profile Query Response Structure

The capabililty profile query function returns with an array (size = profileCount) of profile identifiers and their associated capability matrices.

```
typedef struct {
   NPF_uint32_t profileCount;
   NPF_DS_CapabilityProfileInfo_t
              *capabilityProfileInfoArray;
} NPF_DS_CapabilityProfileQueryResponse_t;
```

4.1.17 Interface Capability Query Response Structure

The following data structure returns an array of profile identifiers which corresponds to the input array of interface handles. The two arrays are parallel arrays.

```
typedef struct {
   NPF_uint32_t profileIDCount;
   NPF_DS_CapabilityProfileID_t *profileIDArray;
} NPF_DS_CapabilityInterfaceQueryResponse_t;
```

4.1.18 Common Response Structure

```
typedef struct {
   NPF_DS_ReturnCode_t returnCode;
   union {
     NPF_DS_PolicyCreateResponse_t policyCreate;
     NPF_DS_MapTableCreateResponse_t tableCreate;
     NPF_DS_QoSObjectCreateResponse_t qosCreate;
     NPF_DS_PolicyUpdateResponse_t policyUpdate;
     NPF_DS_MapTableUpdateResponse_t tableUpdate;
     NPF_DS_QoSObjectUpdateResponse_t qosUpdate;
     NPF_DS_PolicyQueryGetHandleResponse_t
                                policyQueryHandle;
     NPF_DS_PolicyQueryGetContentResponse_t
                                policyQueryContent;
     NPF_DS_PolicyQueryGetBoundObjectResponse_t
                                policyQueryHandle;
     NPF_DS_QoSObjectQueryGetHandleResponse_t
                                qosQueryHandle;
     NPF_DS_QoSObjectQueryGetContentResponse_t
                                qosQueryContent;

     NPF_DS_QoSObjectQueryGetBoundObjectResponse_t
                                qosQueryHandle;

     NPF_DS_MapTableQueryGetHandleResponse_t
                                tableQueryHandle;
     NPF_DS_MapTableQueryGetContentResponse_t
                                tableQueryContent;

     NPF_DS_MapTableQueryGetBoundObjectResponse_t
                                tableQueryHandle;

     NPF_DS_CapabilityProfileQueryResponse_t
                                capabilityProfileQuery;
     NPF_DS_CapabilityInterfaceQueryResponse_t
                           capabilityInterfaceQuery;

     NPF_DS_CounterStatsData_t counterStatsData;
     NPF_DS_AQMStatsData_t AQMStatsData;
   } returnData;
} NPF_DS_AsyncResponse_t;
```

4.2 Data Structures for Completion Callbacks

This section defines the control structures needed for Completion Callbacks.

4.2.1 Completion Callback Types

Several API functions permit multiple requests to be bundled together into an API function call. Not all of these requests may complete at the same time. When multiple responses can be returned, the completion callback data structure points to an array along with 3 additional fields: type, `allOk` and `numResp`. The type variable indicates which function generated the callback. When all of the responses in a multiple response completion callback complete successfully, `allOk` will be `NPF_TRUE`, the `numResp` field will be zero, and the pointer to the response array will be null. If not

all of the responses are contained in the callback or if not all of the responses were successful, or if the function returns values other than the entry key and a return code, `allOk` will be `NPF_FALSE`, the `numResp` field will be greater than zero, and the pointer to the element array will be non-null.

- `type` - This field indicates the API function invocation related to this response.
- `allOk` - This field is set to `NPF_TRUE` only when all of the requests are answered at once in the callback invocation. Additionally, in order for allOk to be set to `NPF_TRUE`, the `returnCode` for each response element must be `NPF_NO_ERROR`. When `allOK` equals `NPF_TRUE`, numResp must equal zero and the array pointer within the union must be null. `allOk` is set to `NPF_FALSE` whenever there are fewer response elements than there were request elements or one or more of the response elements did not complete successfully.
- `numResp` - When the `allOK` parameter equals `NPF_TRUE`, then `numResp` will be set to zero and the pointer to the array will be null. However, when the `allOK` parameter equals `NPF_FALSE`, `numResp` indicates the number of elements in the array.
- `resp` - pointer to an array of response elements. Each array element contains a return code, indicating the completion status of each request element, and possibly other information specific to the type of request.

The completion callback type indicates the type of structure in the union of callback structures returned in `NPF_DS_CallbackData_t`. The type dictates which function the response is intended for.

```
/* Common callback definition */

typedef enum {
   NPF_DS_POLICY_CREATE                        = 0,
   NPF_DS_MAP_TABLE_CREATE                     = 1,
   NPF_DS_QOS_OBJECT_CREATE                    = 2,
   NPF_DS_POLICY_UPDATE                        = 3,
   NPF_DS_MAP_TABLE_UPDATE                     = 4,
   NPF_DS_QOS_OBJECT_UPDATE                    = 5,
   NPF_DS_POLICY_BIND_IFACE                    = 6,
   NPF_DS_POLICY_UNBIND_IFACE                  = 7,
   NPF_DS_QOS_OBJECT_BIND_IFACE                = 8,
   NPF_DS_QOS_OBJECT_UNBIND_IFACE              = 9,
   NPF_DS_MAP_TABLE_BIND_IFACE                 = 10,
   NPF_DS_MAP_TABLE_UNBIND_IFACE               = 11,
   NPF_DS_QUERY_COUNTER_STATS                  = 12,
   NPF_DS_QUERY_AQM_STATS                      = 13,
   NPF_DS_POLICY_DESTROY                       = 14,
   NPF_DS_MAP_TABLE_DESTROY                    = 15,
   NPF_DS_QOS_OBJECT_DESTROY                   = 16,
   NPF_DS_COUNTERS_CLEAR                       = 17,
   NPF_DS_POLICY_QUERY_GET_HANDLE              = 18,
   NPF_DS_POLICY_QUERY_GET_CONTENT             = 19,
   NPF_DS_POLICY_QUERY_GET_BOUND_OBJECT        = 20,
   NPF_DS_QOS_OBJECT_QUERY_GET_HANDLE          = 21,
   NPF_DS_QOS_OBJECT_QUERY_GET_CONTENT          = 22,
   NPF_DS_QOS_OBJECT_QUERY_GET_BOUND_OBJECT     = 23,
   NPF_DS_MAP_TABLE_QUERY_GET_HANDLE           = 24,
```

```
    NPF_DS_MAP_TABLE_QUERY_GET_CONTENT          = 25,
    NPF_DS_MAP_TABLE_QUERY_GET_BOUND_OBJECT     = 26,
    NPF_DS_CAPABILITY_PROFILE_QUERY             = 27,
    NPF_DS_CAPABILITY_INTERFACE_QUERY           = 28
} NPF_DS_CallbackType_t;
```

4.2.2 Completion Callback Structure

Diffserv callback functions return an `NPF_DS_CallbackData_t` structure which contains information on the completed function call. The member within the union can be identified by examining the type field.

The callback function receives the following structure containing one or more asynchronous responses from a single function call. There are several possibilities:

The called function does a single request

- `numResp` = 1, and the `resp` array has just one element.
- `allOK = NPF_TRUE` if the request completed without error and the only return value is the response code.
- if `allOK = NPF_FALSE`, the `resp` structure has the error code.

The called function supports an array of requests

- All completed successfully, at the same time
  1. `allOK = NPF_TRUE`, `numResp` = 0.

- Some completed, but not all, or there are values besides the response code to return
  1. `allOK = NPF_FALSE`, `numResp` equals the number completed

  2. The resp array will contain one element for each completed request, with the error code in the `NPF_DS_AsyncResponse_t` structure, along with any other information needed to identify which request element the response belongs to.

  3. Callback function invocations are repeated in this fashion until all requests are complete. Responses are not repeated for request elements already indicated as complete in earlier callback function invocations.

```
typedef struct {
   NPF_DS_CallbackType_t type;
   NPF_boolean_t allOK;
   NPF_uint32_t numCallbackResp;
   NPF_DS_AsyncResponse_t *resp;
} NPF_DS_CallbackData_t;
```

4.3 Data Structures for Event Notification

This section is not applicable as no event notifications will be generated by this subsystem.

4.4 Function Calls

4.4.1 Completion Callback

This callback function is for the application to register an asynchronous response handling routine to the Diffserv API implementation. This callback function is intended to be implemented by the

application, and be registered to the Diffserv API implementation through the `NPF_DS_Register` function.

- **NPF_DS_CallbackFunc**

Syntax

```
typedef void (*NPF_DS_CallbackFunc_t) (
   NPF_IN NPF_userContext_t userContext,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_CallbackData_t data);
```

Description

A registered completion callback routine for handling Diffserv asynchronous responses.

This is a required function.

Input Arguments

1. `userContext` - The context item that was supplied by the application when the completion callback routine was registered.
2. `correlator` - The correlator item that was supplied by the application when a Diffserv API function call was invoked.
3. `data` - The response information related to the particular Diffserv callback type, as defined in section [4.1] above.

Output Arguments

None

Return Values

None

4.4.2 Event Notification

This section is not applicable

4.4.3 Callback Registration/Deregistration Functions

- **NPF_DS_Register**

Syntax

```
NPF_error_t NPF_DS_Register(
   NPF_IN NPF_userContext_t userContext,
   NPF_IN NPF_DS_CallbackFunc_t callbackFunc,
   NPF_OUT NPF_callbackHandle_t *callbackHandle);
```

Description

This function is used by an application to register its completion callback function for receiving asynchronous responses related to the Diffserv SAPI function calls. Applications MAY register multiple callback functions using this function. The callback function is identified by the pair of userContext and callbackFunc, and for each individual pair, a unique callbackHandle will be assigned for future reference. Since the callback function is identified by both userContext and callbackFunc, duplicate registration of the same callback function with a different userContext is allowed. Also, the same userContext can be shared among different callback functions. Duplicate registration of the same userContext and callbackFunc pair has no effect, and will output a handle that is already assigned to the pair, and will return NPF_E_ALREADY_REGISTERED.

This is a required function.

Input Arguments

1. `userContext` - A context item for uniquely identifying the context of the application registering the completion callback function. The exact value will be provided back to the registered completion callback function as its first parameter when it is called. Applications can assign any value to the `userContext` and the value is completely opaque to the Diffserv SAPI implementation.

2. `callbackFunc` - The pointer to the completion callback function to be registered.

Output Arguments

1. `callbackHandle` - A unique identifier assigned for the registered `userContext` and `callbackFunc` pair. This handle will be used by the application to specify which callback function to be called when invoking asynchronous Diffserv SAPI functions. It will also be used when deregistering the `userContext` and `callbackFunc` pair.

Return Values

1. `NPF_NO_ERROR` - The registration completed successfully.

2. `NPF_E_BAD_CALLBACK_FUNCTION` - The `callbackFunc` is NULL, or otherwise invalid.

3. `NPF_E_ALREADY_REGISTERED` - No new registration was made since the `userContext` and `callbackFunc` pair was already registered.

- **NPF_DS_Deregister**

Syntax

```
NPF_error_t NPF_DS_Deregister(
   NPF_IN NPF_callbackHandle_t callbackHandle);
```

Description

This function is used by an application to deregister a user context and callback function pair.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The deregistration completed successfully.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The deregistration did not complete successfully due to problems with the callback handle provided.

Notes

a) This API function MAY be invoked by any application no longer interested in receiving asynchronous responses for Diffserv API function calls.

b) This function operates in a synchronous manner, providing a return value as listed above.

c) There may be a timing window where outstanding callbacks continue to be delivered to the callback routine after the deregistration function has been invoked. It is the implementation's

responsibility to guarantee that the callback function is not called after the deregister function has returned.

4.5 Diffserv Function Calls

- **NPF_DS_PolicyCreate**

Syntax

```
NPF_error_t NPF_DS_PolicyCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_PolicyID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting);
```

Description

This function call is used to create a Policy Object in the SAPI implementation block and return its handle in the asynchronous data structure.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `resourceID` – A unique application-chosen identifier to be associated with this resource.
4. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When errorReporting is set to NPF_REPORT_ERRORS, the application cannot make any assumptions about when all the requests in the call will be completed.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not created because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN - The object was not created because of some internal error.
3. NPF_E_RESOURCE_EXISTS – A duplicate request to create a resource was detected.

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. A structure of the form NPF_S_DS_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_S_DS_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null. If not all of the responses are complete or if not all of the responses were successful, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null.

Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

- **NPF_DS_PolicyUpdate**

Syntax

```
NPF_error_t NPF_DS_PolicyUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_uint32_t filterSize,
   NPF_IN NPF_DS_Rule_t *filter,
   NPF_IN NPF_uint32_t actionArraySize,
   NPF_IN NPF_DS_Action_t *actions);
```

Description

This function call is used to configure a Policy Object by updating the Filter table and Action table in that Policy object.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `policyHandle` - Handle through which the object can be referenced in Forwarding layer.
5. `filterArraySize` - Size of the filter (ie, rule array size).
6. `filter` - Pointer to filter object representing a flat set of rules associated with this policy object.
7. `actionArraySize` - Size of action array.
8. `actions` - Pointer to action array associated with this policy object.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`

3. NPF_DS_POLICY_INVALID_HANDLE

4. NPF_DS_POLICY_INVALID_INPUT_PARAM

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. A structure of the form NPF_S_DS_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_S_DS_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null. If not all of the responses are complete or if not all of the responses were successful, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null.

Failing elements may be determined by examining the return code in each array element.

It is the implementation's choice how many responses to return in a single callback. The minimum is one and the maximum is the number of request elements passed in the original API function call.

- **NPF_DS_PolicyDestroy**

Syntax

```
NPF_error_t NPF_DS_PolicyDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle);
```

Description

This function call is used to destroy a Policy Object. The object is removed from the Diffserv context. It will return an error NPF_DS_POLICY_BOUND if the policy is already bound to atleast an interface.

This is a required function.

Input Arguments

1. callbackHandle - The unique identifier returned to the application when the completion callback routine was registered.

2. correlator - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. errorReporting -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When errorReporting is set to NPF_REPORT_ERRORS, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `policyHandle` - Handle through which the object can be referenced in the FCL.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not destroyed because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_POLICY_INVALID_HANDLE`
4. `NPF_DS_POLICY_BOUND` - Policy is bound to at least one interface. First unbind them before destroying it.

- **NPF_DS_PolicyBind**

Syntax

```
NPF_error_t NPF_DS_PolicyBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to bind/associate this policy object with this interface or a hierarchical interface entity. The interface object is referenced by the interface handle. It will return an error `NPF_DS_POLICY_BOUND_IFACE` if it is already bound to the interface.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `policyHandle` - Handle through which the policy object can be referenced in Forwarding layer.
5. `ifaceHandle` - Interface Object Handle.
6. `plane`

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not bound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_POLICY_INVALID_HANDLE`
4. `NPF_DS_POLICY_BOUND_IFACE` - Policy is already bound to the interface.
5. `NPF_DS_POLICY_NOT_SUPPORTED` - Policy is not supported on this Plane because of lack of feature support
6. `NPF_DS_INVALID_PLANE`

- **NPF_DS_PolicyUnbind**

Syntax

```
NPF_error_t NPF_DS_PolicyUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to unbind this Policy object from this interface or hierarchical interface entity in the Diffserv Forwarding Control layer.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `policyHandle` - Handle through which the Policy object can be referenced in Forwarding layer.
5. `ifaceHandle` - Interface Object Handle.
6. `plane`

Output Arguments

None

Return Values

1. NPF_NO_ERROR - The operation is in progress.
2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not unbound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN
3. NPF_DS_POLICY_INVALID_HANDLE
4. NPF_DS_POLICY_NOT_BOUND_IFACE - Policy is not bound.
5. NPF_DS_INVALID_PLANE

- **NPF_DS_PolicyQueryGetHandles**

Syntax

```
NPF_error_t NPF_DS_PolicyQueryGetHandles(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);
```

Description

This function call is used to query all the object handles of object type POLICY. The call back data structure will hold the object handles and their corresponding resource ids and a flag indicating whether they were configured or are the default ones.

This is an optional function.

Input Arguments

1. callbackHandle - The unique identifier returned to the application when the completion callback routine was registered.
2. correlator - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. errorReporting -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When

`errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. A structure of the form NPF_S_DS_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_S_DS_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null. If not all of the responses are complete or if not all of the responses were successful, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null.

Failing elements may be determined by examining the return code in each array element.

- **NPF_DS_PolicyQueryGetContents**

Syntax

```
NPF_error_t NPF_DS_PolicyQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
```

```
NPF_IN NPF_DS_PolicyHandle_t *handles);
```

Description

This function call is used to query contents of the Policy objects by providing their handles.
This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `handleArraySize` - Size of handle array.
5. `handles` - Pointer to handle array associated with policy objects.

Output Arguments

None

Return Values

1. NPF_NO_ERROR - The operation is in progress.
2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN

- **NPF_DS_PolicyQueryGetBoundObjects**

Syntax

```
NPF_error_t NPF_DS_PolicyQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to query bound Policy objects on an interface and a plane. The handles of these bound Policy Objects are returned in a callback data structure.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` - An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `ifaceHandle – Interface Handle`
5. `plane`

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN

- **NPF_DS_MapTableCreate**

Syntax

```
NPF_error_t NPF_DS_MapTableCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_MapTableID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableType_t type);
```

Description

This function call is used to create a table of type "type" in the Diffserv Forwarding Control layer and return its handle in the Asynchronous data structure.

This is a required function.

Input Arguments

1. callbackHandle - The unique identifier returned to the application when the completion callback routine was registered.
2. correlator - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. resourceID – A unique application-chosen identifier to be associated with this resource.
4. errorReporting -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When errorReporting is set to NPF_REPORT_ERRORS, the application cannot make any assumptions about when all the requests in the call will be completed.
5. type - Table type to be created.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not created because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. `NPF_E_UNKNOWN` - Could not create the table for some failure reason.

3. `NPF_E_RESOURCE_EXISTS – A duplicate request to create a resource was detected.`

- **NPF_DS_MapTableUpdate**

Syntax

```
NPF_error_t NPF_DS_MapTableUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_DS_MapTableData_t data);
```

Description

This function call is used to configure/update the table with the new data.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `tableHandle` - Handle of the table object which has to be updated.

5. `data` - Table data.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. `NPF_E_UNKNOWN` - The table was not configured due to problems encountered when handling the input parameters .

3. `NPF_DS_MapTable_INVALID_HANDLE`

4. `NPF_DS_MapTable_INVALID_INPUT_PARAM`

- **NPF_DS_MapTableDestroy**

Syntax

```
NPF_error_t NPF_DS_MapTableDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle);
```

Description

This function call is used to destroy the table referenced by tableHandle.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `tableHandle` - Handle through which the object can be referenced in Forwarding layer.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not destroyed because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_MapTable_INVALID_HANDLE`
4. `NPF_DS_MapTable_BOUND` - Table is bound to atleast an interface. First unbind them before destroying the table.

.

- **NPF_DS_MapTableQueryGetHandles**

Syntax

```
NPF_error_t NPF_DS_MapTableQueryGetHandles(
```

```
NPF_IN NPF_callbackHandle_t callbackHandle,
NPF_IN NPF_correlator_t correlator,
NPF_IN NPF_errorReporting_t errorReporting);
```

Description

This function call is used to query all the object handles of object type MAPTABLE. The call back data structure will hold the object handles and their corresponding resource ids and a flag indicating whether they were configured or are the default ones.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. A structure of the form `NPF_S_DS_CallbackData_t` will

be returned with each callback. As part of that structure, an array of `NPF_S_DS_AsyncResponse_t` structures will also be returned. If all of the elements in the request array completed successfully, the callback will return an `allOK` value of `NPF_TRUE`, a `numResp` value of zero, and the array pointer will be null. If not all of the responses are complete or if not all of the responses were successful, `allOK` will be `NPF_FALSE`, the `numResp` field will be greater than zero, and the pointer to the element array will be non-null.

Failing elements may be determined by examining the return code in each array element.

- **NPF_DS_MapTableQueryGetContents**

Syntax

```
NPF_error_t NPF_DS_MapTableQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
   NPF_IN NPF_DS_MapTableHandle_t *handles);
```

Description

This function call is used to query contents of the Mapping Table objects by providing their handles. This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `handleArraySize` - Size of handle array.

5. `handles` - Pointer to handle array associated with policy objects.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`

- **NPF_DS_MapTableQueryGetBoundObjects**

Syntax

```
NPF_error_t NPF_DS_MapTableQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_IfHandle_t ifaceHandle);
```

Description

This function call is used to query bound Mapping Table objects on an interface. The handles of these bound Mapping Table Objects are returned in a callback data structure.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` - An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `ifaceHandle` – Interface Handle

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`

- **NPF_DS_QoSObjectCreate**

Syntax

```
NPF_error_t NPF_DS_QoSObjectCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_QoSObjectID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSObjectType_t type);
```

Description

This function call is used to create a object of type "type" in the Diffserv Forwarding Control layer and return its handle in the asynchronous data structure.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `resourceID` – A unique application-chosen identifier to be associated with this resource.

4. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

5. `type` - Object type to be created.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not created because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. `NPF_E_UNKNOWN` - Could not create the QoS Object for some failure reason.

3. `NPF_E_RESOURCE_EXISTS – A duplicate request to create a resource was detected.`

- **NPF_DS_QoSObjectUpdate**

Syntax

```
NPF_error_t NPF_DS_QoSObjectUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
```

```
NPF_IN NPF_correlator_t correlator,
NPF_IN NPF_errorReporting_t errorReporting,
NPF_IN NPF_DS_QoSHandle_t objectHandle,
NPF_IN NPF_DS_QoSObjectData_t data);
```

Description

This function call is used to configure/update the QoS Object with the new data.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `objectHandle` - Handle of the QoS object which has to be updated.
5. `data` - Table data.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN` - The QoS Object was not configured due to problems encountered when handling the input parameters .
3. `NPF_DS_QOS_OBJECT_INVALID_HANDLE`

4. NPF_DS_QOS_OBJECT_INVALID_INPUT_PARAM

- **NPF_DS_QoSObjectDestroy**

Syntax

```
NPF_error_t NPF_DS_QoSObjectDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle);
```

Description

This function call is used to destroy a QoS Object. The object is removed from the Diffserv Forwarding layer.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `objectHandle` - Handle through which the object can be referenced in Forwarding layer.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not destroyed because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN
3. NPF_DS_QOS_OBJECT_INVALID_HANDLE
4. NPF_DS_QOS_OBJECT_BOUND - Couldn't destroy the object as it is bound to an interface. First unbind it.

- **NPF_DS_QoSObjectQueryGetHandles**

Syntax

```
NPF_error_t NPF_DS_QoSObjectQueryGetHandles(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);
```

Description

This function call is used to query all the QoS object handles. The call back data structure will hold the object handles and their corresponding resource ids and a flag indicating whether they were configured or are the default ones.

This is an optional function.

Input Arguments

1. callbackHandle - The unique identifier returned to the application when the completion callback routine was registered.
2. correlator - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. errorReporting -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When errorReporting is set to NPF_REPORT_ERRORS, the application cannot make any assumptions about when all the requests in the call will be completed.

Output Arguments

None

Return Values

1. NPF_NO_ERROR - The operation is in progress.
2. NPF_E_BAD_CALLBACK_HANDLE - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN

The response array returned in the callback may contain between zero and the number of elements requested with this API function call. A structure of the form NPF_S_DS_CallbackData_t will be returned with each callback. As part of that structure, an array of NPF_S_DS_AsyncResponse_t structures will also be returned. If all of the elements in the request array completed successfully, the callback will return an allOK value of NPF_TRUE, a numResp value of zero, and the array pointer will be null. If not all of the responses are complete or if not all of the responses were successful, allOK will be NPF_FALSE, the numResp field will be greater than zero, and the pointer to the element array will be non-null.

Failing elements may be determined by examining the return code in each array element.

- **NPF_DS_QoSObjectQueryGetContents**

Syntax

```
NPF_error_t NPF_DS_QoSObjectQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
   NPF_IN NPF_DS_QoSHandle_t *handles);
```

Description

This function call is used to query contents of the QoS objects by providing their handles.
This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `handleArraySize` - Size of handle array.
5. `handles` - Pointer to handle array associated with policy objects.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`

- **NPF_DS_QoSObjectQueryGetBoundObjects**

Syntax

```
NPF_error_t NPF_DS_QoSObjectQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to query bound QoS objects on an interface and a plane. The handles of these bound QoS Objects are returned in a callback data structure.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `ifaceHandle – Interface Handle`
5. `plane`

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not configured because the callback handle was invalid.

Asynchronous Response

There will be one asynchronous callback to this request. Possible return codes are:

1. NPF_NO_ERROR
2. NPF_E_UNKNOWN

- **NPF_DS_MapTableBind**

Syntax

```
NPF_error_t NPF_DS_MapTableBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle);
```

Description

This function call is used to associate the given table with the given interface or hierarchical interface entity. In other words, we are binding the table object to the interface.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `tableHandle` - Handle through which the Table can be referenced in Forwarding layer.
5. `ifaceHandle` - Interface Object Handle.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not bound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_MAPTABLE_INVALID_HANDLE`
4. `NPF_DS_MAPTABLE_BOUND_IFACE` - Table is already bound
5. `NPF_DS_MAPTABLE_NOT_SUPPORTED` - Policy is not supported on this Plane because of lack of feature support

- **NPF_DS_MapTableUnbind**

Syntax

```
NPF_error_t NPF_DS_MapTableUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle);
```

Description

This function call is used to unbind the table from the interface or hierarchical interface entity.

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any `assumptions` about when all the requests in the call will be completed.

4. `tableHandle` - Handle through which the table can be referenced in Forwarding layer.

5. `ifaceHandle` - Interface Object Handle.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not unbound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_MAPTABLE_INVALID_HANDLE`
4. `NPF_DS_MAPTABLE_NOT_BOUND_IFACE` - Table is not bound.

- **NPF_DS_QoSObjectBind**

Syntax

```
NPF_error_t NPF_DS_QoSObjectBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
```

```
NPF_IN NPF_correlator_t correlator,
NPF_IN NPF_errorReporting_t errorReporting,
NPF_IN NPF_DS_QoSHandle_t objectHandle,
NPF_IN NPF_IfHandle_t ifaceHandle,
NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to associate the given QoS Object with the given interface or hierarchical interface entity. In other words, we are binding the

This is a required function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `objectHandle` - Handle through which the QoS object can be referenced in Forwarding layer.
5. `ifaceHandle` - Interface Object Handle.
6. `plane`

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not bound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. NPF_E_UNKNOWN
3. NPF_DS_QOS_OBJECT_INVALID_HANDLE
4. NPF_DS_QOS_OBJECT_BOUND_IFACE - QoS Object is already bound to the interface.
5. NPF_DS_QOS_OBJECT_NOT_SUPPORTED – QoS Object is not supported on this Plane because of lack of feature support
6. NPF_DS_INVALID_PLANE

- **NPF_DS_QoSObjectUnbind**

Syntax

```
NPF_error_t NPF_DS_QoSObjectUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to unbind the QoS object from the interface or hierarchical interface entity.

This is a required function.

Input Arguments

1. callbackHandle - The unique identifier returned to the application when the completion callback routine was registered.
2. correlator - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. errorReporting -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When errorReporting is set to NPF_REPORT_ERRORS, the application cannot make any assumptions about when all the requests in the call will be completed.
4. objectHandle - Handle through which the QoS object can be referenced in Forwarding layer.

5. `ifaceHandle` - Interface Object Handle.
6. `plane`

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not unbound because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_QOS_OBJECT_INVALID_HANDLE`
4. `NPF_DS_QOS_OBJECT_NOT_BOUND_IFACE` - QoS Object is not bound.
5. `NPF_DS_INVALID_PLANE`

- **NPF_DS_PolicyCountersGet**

Syntax

```
NPF_error_t NPF_DS_PolicyCountersGet(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to query the policy counters for that interface or hierarchical interface entity.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `policyHandle` - Handle through which the Policy object can be referenced in Forwarding layer.

5. `ifaceHandle` - Interface Object Handle.

6. plane – The plane from which the counters are to be obtained.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not queried because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. `NPF_E_UNKNOWN` - The policy was not bound to the interface.

3. `NPF_DS_POLICY_INVALID_HANDLE`

4. `NPF_DS_POLICY_NOT_BOUND_IFACE`

- **NPF_DS_QueueCountersGet**

Syntax

```
NPF_error_t NPF_DS_QueueCountersGet(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t qosHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);
```

Description

This function call is used to query the QoS Object NPF_DS_QOS_QUEUE for its counters (queue and AQM statistics) for that interface or hierarchical interface entity.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` - An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `qosHandle` - Handle through which the QoS object can be referenced in Forwarding layer.
5. `ifaceHandle` - Interface Object Handle.
6. plane – The plane from which the counters are to be obtained.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not queried because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`
3. `NPF_DS_QOS_OBJECT_INVALID_HANDLE`
4. `NPF_DS_QOS_OBJECT_NOT_BOUND_IFACE`

- **NPF_DS_CountersClear**

Syntax

```
NPF_error_t NPF_DS_CountersClear(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t numCounters,
   NPF_IN NPF_DS_QoSHandle_t *counterHandleArray);
```

Description

This function call is used to clear specific counters which are specified in the counterHandleArray.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.
3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.
4. `numCounters` - The number of counters in the counter handle array.
5. `counterHandleArray` - A pointer to an array of counter handles which are to be cleared.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not queried because the callback handle was invalid.

Asynchronous Response

There may be multiple asynchronous callbacks to this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN` - The policy was not bound to the interface.
3. `NPF_DS_QOS_OBJECT_INVALID_HANDLE`

- **NPF_DS_CapabilityProfileQuery**

Syntax

```
NPF_error_t NPF_DS_CapabilityProfileQuery(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);
```

Description

This function call is used to obtain a list of all unique capability profile identifiers in the underlying system.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.
2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.
2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not queried because the callback handle was invalid.

Asynchronous Response

There is a single asynchronous callback which results from this request. Possible return codes are:

1. `NPF_NO_ERROR`
2. `NPF_E_UNKNOWN`.

- **NPF_DS_CapabilityInterfaceQuery**

Syntax

```
NPF_error_t NPF_DS_CapabilityInterfaceQuery(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t ifHandlesCount,
   NPF_IN NPF_IfHandle_t *ifHandlesArray);
```

Description

This function call is used to obtain a list of profile identifiers that are associated with the input list of interface handles.

This is an optional function.

Input Arguments

1. `callbackHandle` - The unique identifier returned to the application when the completion callback routine was registered.

2. `correlator` - A unique application invocation context that will be supplied to the asynchronous completion callback routine.

3. `errorReporting` -An indication of whether the application desire to receive an asynchronous completion callback for this API function invocation. When `errorReporting` is set to `NPF_REPORT_ERRORS`, the application cannot make any assumptions about when all the requests in the call will be completed.

4. `i`fHandlesCount – the count of the number of handles in the interface handle array.

5. `i`fHandlesArray – the array of interface handles for which the profile query is to occur.

Output Arguments

None

Return Values

1. `NPF_NO_ERROR` - The operation is in progress.

2. `NPF_E_BAD_CALLBACK_HANDLE` - The objects were not queried because the callback handle was invalid.

Asynchronous Response

There is a single asynchronous callback which results from this request. Possible return codes are:

1. `NPF_NO_ERROR`

2. `NPF_E_UNKNOWN`.

5 Normative References

The following documents contain provisions, which through reference in this text constitute provisions of this specification. At the time of publication, the editions indicated were valid. All referenced documents are subject to revision, and parties to agreements based on this specification are encouraged to investigate the possibility of applying the most recent editions of the standards indicated below.

[SWAPI] "Software API Conventions – Implementation Agreement," Network Processing Forum, Revision 1.0 (http://www.npforum.org/techinfo/FoundationsIAV1.pdf), Aug. 2002.

[IF] "Interface Management API Revision 2 – Implementation Agreement," Network Processing Forum, Dec. 2003.

[RFC2474] "Definition of the Differentiated Services Field in the IPv4 and IPv6 headers," K. Nichols, S. Blake, F. Baker, and D. Black, RFC 2474, Dec. 1998.

[RFC2475] "An Architecture for Differentiated Services," S. Blake, D. Black, M. Carlson, E. Davies, Z. Wang, and W. Weiss, RFC 2475, Dec. 1998.

[RFC3168] "A Proposal to add Explicit Congestion Notification (ECN) to IP," K. Ramakrishnan, S. Floyd and D. Black, RFC 3168, Sep. 2001.

[RFC3289] "Management Information Base for the Differentiated Services Architecture," F. Baker, K. Chan, and A. Smith, RFC 3289, May 2002.

6 Acronyms and Abbreviations

The following acronyms and abbreviations are used in this specification:

- ACL Access Control List
- AF Assured Forwarding
- AQM Active Queue Management
- BE Best Effort
- BSP Bounded Strict Priority
- CMAP Class-map
- DRR Deficit Round Robin
- DS Differentiated Services
- DSCP Diffserv Code Point
- EF Expedited Forwarding
- FAPI Functional Application Programming Interface
- FE Forwarding Engine
- IP Internet Protocol
- OAM Operation and Management
- PBR Policy Based Routing
- PHB Per-Hop Behavior
- PMAP Policy Map
- PSC Per-hop Scheduling Class
- QoS Quality of Service
- RED Random Early Detection
- SAPI Services Application Programming Interface
- SP Strict Priority
- SRTCM Single-Rate Three-Color Meter
- TCP Transmission Control Protocol
- TOS Type of Service
- TRTCM Three-Rate Three-Color Meter
- UDP User Datagram Protocol
- WFQ Weighted Fair Queue
- WRED Weighted RED
- WRR Weighted Round Robin

APPENDIX A INFORMATIVE ANNEXES

A.1. ANNEX: CISCO CONFIGURATION EXAMPLES

The following example is a short-hand Cisco policy map (PMAP) which contains a single class map (CMAP). If this class map evaluates to true for a given packet, then the packet is to be dropped. The class map contains three match conditions. Since this class map is of type "match any" then if any of the three match conditions evaluates to true, then the entire class map evaluates to true.

In the examples below, f1...f5 represent rules that we would like to match in the incoming packets. For instance, f1 may represent {IPSA = 192.168.x.x && IPDA = 1.1.1.2 && IP Protocol = UDP}.

```
PMAP p1 {
     CMAP c1 : DROP
}

CMAP c1 match any {
     match DSCP 1
     match ACL acl1
     match ACL acl2
}

ACL acl1 {
     f1 permit
     f2 deny
     f3 permit
}

ACL acl2 {
     f4 deny
     f5 permit
}
```

The action of the policy map p1 will execute if any of the three match conditions in the class map c1 evaluate to true. That is,

```
DSCP == 1 or
f1 or
!f1 and !f2 and f3 or
!f4 and f5
```

Hence this policy would be represented as follows in the current SAPI filter/action representation:

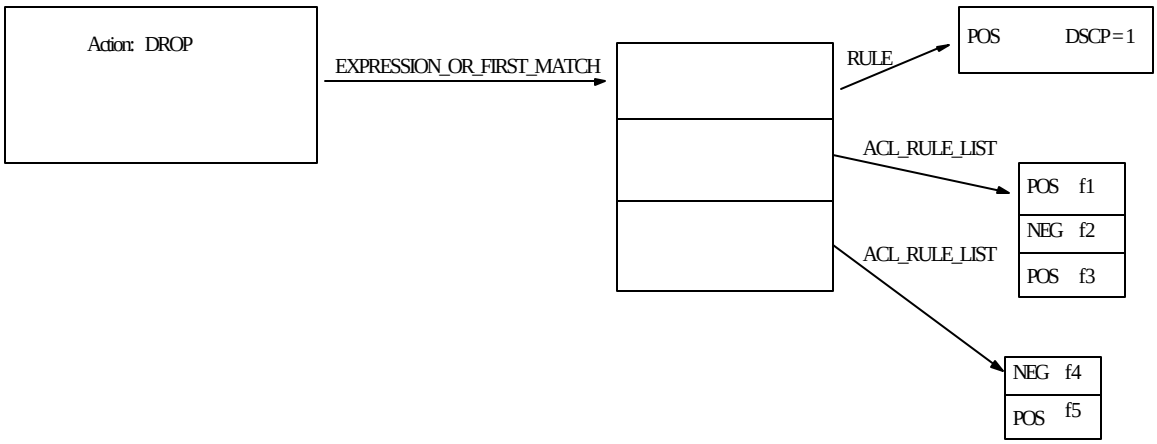

Figure 10: Example 1 (Using OR Expression)

Now if the previous example were changed slightly such that the CMAP was a match-all (i.e., all of the match conditions must evaluate to true), then the SAPI representation presented below could be used.

```
CMAP c1 match all {
     match DSCP 1
     match ACL acl1
     match ACL acl2
}
```

In this case, in order for the action to be executed, the packet must satisfy all three match conditions in the CMAP. That is, its DSCP field must contain a 1, acl1 must be satisfied, as well as acl2. Let us now elaborate the conditions which must be true in order for the packet to satisfy the ACLs. In acl1, if rule f1 matches, then the ACL is satisfied since this rule contains a permit action. If f1 is not satisfied, but f2 is satisfied, then the ACL is not satisfied since this rule contains a deny action. Finally, if f1 is not satisfied and f2 is not satisfied and f3 is satisfied, then the entire ACL is satisfied because of the permit action of the final rule. Similarly, looking at acl2, for this ACL to be satisfied, we must NOT satisfy rule f4 and we must satisfy rule f5 due to the respective deny and permit actions for each rule. So, in summary, the drop action will get executed if the packet satisfies either of the following conditions:

```
(DSCP == 1 and f1 and !f4 and f5) or
(DSCP == 1 and !f1 and !f2 and f3 and !f4 and f5)
```

Now, this example can be represented in the current DS SAPI as the following:

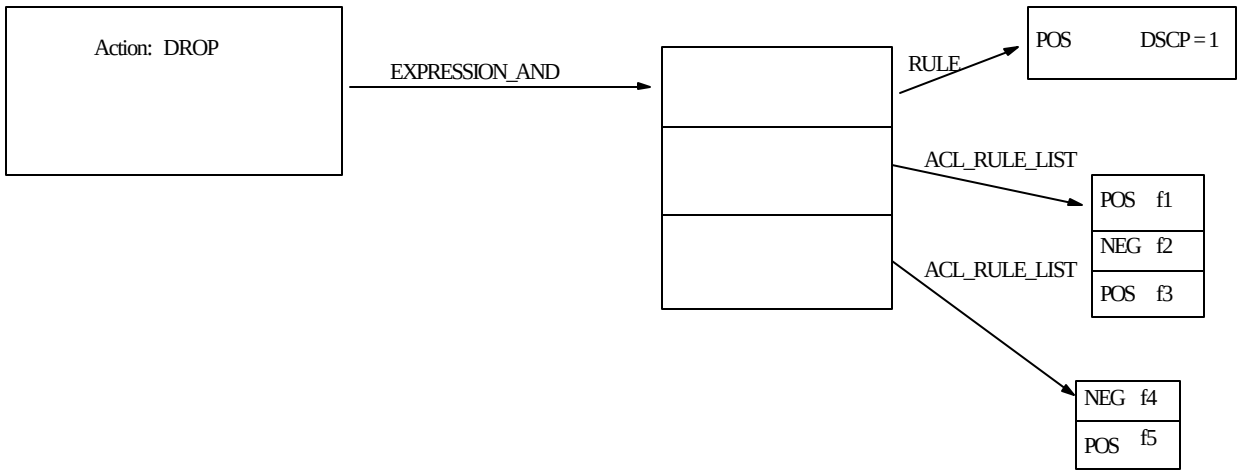

Figure 11: Example 2 (Using AND Expression)

A.2. JUNIPER CONFIGURATION EXAMPLE

Here is an example filter configuration similar to what was presented for a Cisco style example in Annex A.1.

Here is a similar example in Juniper syntax. No exact match to the ACL object (although it may be there)was found. Instead rules, which may include multiple fields, are written within the 'from' statement of a 'term'. There is a concept of a prefix list which can be matched in the 'from' statement. which can be chosen to provide a similar list of match conditions with mix of 'permit' and 'deny'. The 'except' statement is like the 'deny' in ACL.

From firewall filter config mode:

```
filter p1 {
  term dscp-match-term {
    from {
      dscp 1;
    }
    then {
      discard;
    }
  }
  term pl1-match-term {
    from {
      prefix-list pl1;
    }
    then {
      discard;
    }
  }
  term pl2-match-term {
    from {
      prefix-list pl2;
    }
    then {
      discard;
    }
  }
}

prefix-list pl1 {
  addr1;
  addr2 except;
  addr3;
}

prefix-list pl2 {
  addr4 except;
```

```
  addr5;
}
```

This filter will result in the following treatment,

```
if(DSCP == 1)
     drop;
else if ((addr1) || (!addr1  &&  !addr2  && addr3))
     drop;
else if (!addr4)  && (addr5))
     drop;
```

which will map to the same filter array as in the Cisco match-any example.

Here is an example similar to the Cisco match-all using the same building blocks as the previous example. Since this example looks quite different from the previous one (an in Cisco style only the match-all/any was changed), it raises the question of whether there may be a better way to represent these.

```
filter p1 {
  term match-all-term {
    from {
      dscp 1;
      prefix-list {
        pl1;
      }
      prefix-list {
        pl2;
      }
    }
    then {
      discard;
    }
  }
}
```

This filter will result in the following treatment,

```
 if ((DSCP == 1) &&
      ((addr1) || (!addr1 && !addr2 && addr3)) &&
      (!addr4 && addr5))
    drop;
```

which will map to the same filter array as in the Cisco match-all example.

A.3. DIFFSERV SAPI HEADER FILE

```
/*
 *
 * dssapi.h - Diffserv Services API definitions,
 *            data structures, and
 *            API function prototypes
 */

#ifndef __DSSAPI_H__
#define __DSSAPI_H__
#ifdef __cplusplus
extern "C" {
#endif

#include <npf.h>


/* DS SAPI Handles used by the API to refer to Policy
* Objects, QoS Objects and Mapping Tables
*/
typedef NPF_uint32_t NPF_DS_PolicyHandle_t;
typedef NPF_uint32_t NPF_DS_QoSHandle_t;
typedef NPF_uint32_t NPF_DS_MapTableHandle_t;


/*
 * Resource IDs
 */
typedef NPF_uint32_t NPF_DS_PolicyID_t;
typedef NPF_uint32_t NPF_DS_QoSObjectID_t;
typedef NPF_uint32_t NPF_DS_MapTableID_t;
#define NPF_DS_QUERY_DEFAULT_OBJECT_RESOURCE_ID   -1



/*
 * Filter-related data structures + definitions
 */

typedef enum {
   NPF_DS_PLANE_IPINGRESS        = 0,
   NPF_DS_PLANE_IPEGRESS         = 1,
   NPF_DS_PLANE_LOCAL_TERMINATED = 2,
   NPF_DS_PLANE_LOCAL_GENERATED  = 3,
   NPF_DS_PLANE_L2DECAP          = 4,
   NPF_DS_PLANE_L2ENCAP          = 5
} NPF_DS_FilterPlane_t;


/* Rule Identifier */
typedef NPF_int32_t NPF_DS_RuleID_t;


/* Action Identifier */
typedef NPF_uint32_t NPF_DS_ActionID_t;


/* Layer 2 rule conditions */
```

```
typedef struct {
   NPF_uchar8_t mac_SA[6]; /* 0 means any */
   NPF_uchar8_t mac_DA[6]; /* 0 means any */
   NPF_uchar8_t priority;  /* 802.1p priority */
   NPF_uchar8_t vlan_ID;   /* priority + vlan_ID forms tag for
                              802.1Q */
} NPF_DS_RuleCond_L2_t;


/* IPv4 layer 3 rule conditions */
typedef struct {
   NPF_char8_t proto;
   NPF_uint32_t lengthMin, lengthMax;
   NPF_IPv4Address_t srcAddr, srcAddrMask;
   NPF_IPv4Address_t destAddr, destAddrMask;
   NPF_char8_t tos, tosMask;
   NPF_uchar8_t fragments;
            /* 0 means any; 1 means */
            /*   non-initial fragments only */

} NPF_DS_RuleCondIPv4_L3_t;

/* Special IPv4 layer 3 rule field values for wildcards */
#define NPF_DS_WILDCARD_IPV4_PROTO        255
#define NPF_DS_WILDCARD_IPV4_LENGTHMIN    0
#define NPF_DS_WILDCARD_IPV4_LENGTHMAX    65535
#define NPF_DS_WILDCARD_IPV4_SRCADDRMASK  0
#define NPF_DS_WILDCARD_IPV4_DESTADDRMASK 0
#define NPF_DS_WILDCARD_IPV4_TOSMASK      0
#define NPF_DS_WILDCARD_IPV4_FRAGMENTS    0


#define NPF_DS_TOSBYTE_DSCP_MASK          0xFC
#define NPF_DS_TOSBYTE_IPPREC_MASK        0xE0
#define NPF_DS_TOSBYTE_TOS_MASK           0x07


/* IPv6 layer 3 rule conditions */
typedef struct {
   NPF_char8_t class;
   NPF_int64_t flowLabel;
   NPF_int16_t next_header;
   NPF_IPv6Address_t srcAddr, srcAddrMask;
   NPF_IPv6Address_t destAddr, destAddrMask;

} NPF_DS_RuleCondIPv6_L3_t;

/* Special IPv6 layer 3 rule field values for wildcards */
#define NPF_DS_WILDCARD_IPV6_PRIORITY     -1
#define NPF_DS_WILDCARD_IPV6_FLOW_LABEL   0
#define NPF_DS_WILDCARD_IPV6_NEXT_HEADER  -1
#define NPF_DS_WILDCARD_IPV6_SRCADDR      0
#define NPF_DS_WILDCARD_IPV6_SRCADDRMASK  0
#define NPF_DS_WILDCARD_IPV6_DESTADDR     0
#define NPF_DS_WILDCARD_IPV6_DESTADDRMASK 0


/* Layer 3 type selector--IPv4 or IPv6 rule */
typedef enum {
   NPF_DS_RULE_IPV4 = 0,
   NPF_DS_RULE_IPV6 = 1
```

```
} NPF_DS_RuleL3Type_t;


/* Layer 3 rule conditions */
typedef struct {
   NPF_DS_RuleL3Type_t L3Type;
   union {
      NPF_DS_RuleCondIPv4_L3_t condIPv4_L3;
      NPF_DS_RuleCondIPv6_L3_t condIPv6_L3;
   } condIP_L3;
} NPF_DS_RuleCondIP_L3_t;


/* IP layer 4 rule conditions */
typedef union {
   struct {
      NPF_uint32_t SRCP_Min, SRCP_Max;
      NPF_uint32_t DSTP_Min, DSTP_Max;
      NPF_char8_t TCP_Flags, TCP_FlagsMask;
   } L4TCP_UDP_Cond;
   struct {
      NPF_int32_t Type;
      NPF_int32_t Code;
   } L4ICMP_Cond;
} NPF_DS_RuleCondIP_L4_t;

/* Special layer 4 rule field values for wildcards */
#define NPF_DS_WILDCARD_TCPUDP_SRCPMIN     0
#define NPF_DS_WILDCARD_TCPUDP_SRCPMAX     65535
#define NPF_DS_WILDCARD_TCPUDP_DSTPMIN     0
#define NPF_DS_WILDCARD_TCPUDP_DSTPMAX     65535
#define NPF_DS_WILDCARD_TCP_FLAGMASK       0
#define NPF_DS_WILDCARD_ICMP_TYPE          -1
#define NPF_DS_WILDCARD_ICMP_CODE          -1


/* IP layer 3/4 rule conditions */
typedef struct {
   NPF_DS_RuleCondIP_L3_t condIP_L3;
   NPF_DS_RuleCondIP_L4_t condIP_L4;
} NPF_DS_RuleCond_L34_t;


/* Layer 2 or Layer 3+4 rule conditions */
typedef union {
   NPF_DS_RuleCond_L2_t cond_L2;
   NPF_DS_RuleCond_L34_t cond_L34;
} NPF_DS_RuleCond_t;


/* Rule type:  Positive or Negative */
typedef enum {
   NPF_DS_RULE_TYPE_POSITIVE = 0,
   NPF_DS_RULE_TYPE_NEGATIVE = 1
} NPF_DS_RuleType_t;


/* Rule Entry */
typedef struct {
   NPF_DS_RuleID_t id;
   NPF_DS_RuleCond_t ruleCond;
```

```
   NPF_DS_RuleType_t ruleType;
} NPF_DS_Rule_t;


/* Rule Array */
typedef struct {
   NPF_uint32_t ruleArraySize;
   NPF_DS_Rule_t *ruleArray;
} NPF_DS_RuleArray_t;


/* Expression OR-Multiple-Match Type Selector */
typedef enum {
   NPF_DS_MULTI_OR_RULE                       = 0,
   NPF_DS_MULTI_OR_EXPRESSION_AND             = 1,
   NPF_DS_MULTI_OR_EXPRESSION_OR_FIRST_MATCH = 2,
   NPF_DS_MULTI_OR_ACL_RULE_LIST              = 3
} NPF_DS_ExpressionTypeSelectorORMultipleMatch_t;


/* Expression OR-First-Match Type Selector */
typedef enum {
   NPF_DS_OR_FIRST_MATCH_RULE          = 0,
   NPF_DS_OR_FIRST_MATCH_ACL_RULE_LIST = 1
} NPF_DS_ExpressionTypeSelectorORFirstMatch_t;


/* Expression AND Type Selector */
typedef enum {
   NPF_DS_AND_RULE          = 0,
   NPF_DS_AND_ACL_RULE_LIST = 1
} NPF_DS_ExpressionTypeSelectorAND_t;


/* Expression Element -- AND */
typedef struct {
   NPF_DS_ExpressionTypeSelectorAND_t typeSelector;
   union {
      NPF_DS_Rule_t *rule;  /* for ts = rule */
      NPF_DS_RuleArray_t ACL_ruleArray; /* for ts RuleList */
   } expressionData;
} NPF_DS_ExpressionElementAND_t;


/* Expression Array -- AND */
typedef struct {
   NPF_uint32_t arraySize;
   NPF_DS_ExpressionElementAND_t *expressionANDArray;
} NPF_DS_ExpressionArrayAND_t;


/* Expression Element -- OR-First-Match */
typedef struct {
   NPF_DS_ExpressionTypeSelectorORFirstMatch_t typeSelector;
union {
      NPF_DS_Rule_t *rule;  /* for ts = rule */
      NPF_DS_RuleArray_t ACL_ruleArray; /* for ts = RuleArray */
   } expressionData;

   NPF_DS_ActionID_t actionID;
```

```
} NPF_DS_ExpressionElementORFirstMatch_t;


/* Expression Array -- OR-First-Match */
typedef struct {
   NPF_uint32_t arraySize;
   NPF_DS_ExpressionElementORFirstMatch_t
*expressionArrayORFirstMatch;
} NPF_DS_ExpressionArrayORFirstMatch_t;


/* Expression Element -- OR-Multiple-Match */
typedef struct {
   NPF_DS_ExpressionTypeSelectorORMultipleMatch_t typeSelector;

   union {
      NPF_DS_Rule_t *rule;  /* for ts = rule */
      NPF_DS_ExpressionArrayAND_t expressionArrayAND;
                  /* for ts = Expression_AND */
      NPF_DS_ExpressionArrayORFirstMatch_t
                     expressionArrayORFirst;
                  /* for ts = Expression_OR */

      NPF_DS_RuleArray_t ACL_ruleArray; /* for ts = RuleArray */
   } expressionData;

   NPF_DS_ActionID_t actionID;

} NPF_DS_ExpressionElementORMultipleMatch_t;


typedef struct {
   NPF_uint32_t policyElementCount;
   NPF_DS_ExpressionElementORMultipleMatch_t
                  *policyElementArray;
} NPF_DS_Policy_t;


/*
 * Action-related data structures + definitions
 */


/* Action Types */
typedef enum {
   NPF_DS_ACTION_NOP           = 0,
   NPF_DS_ACTION_COUNTER       = 1,
   NPF_DS_ACTION_POLICE        = 2,
   NPF_DS_ACTION_SETDSCP       = 3,
   NPF_DS_ACTION_SETIPPREC     = 4,
   NPF_DS_ACTION_SETNEXTHOP    = 5,
   NPF_DS_ACTION_SETFIB        = 6,
   NPF_DS_ACTION_SETCOLOR      = 7,
   NPF_DS_ACTION_DOWNMARK      = 8,
   NPF_DS_ACTION_DROP          = 9,
   NPF_DS_ACTION_SHAPE         = 10,
   NPF_DS_ACTION_FORWARD       = 11,
   NPF_DS_ACTION_RETURN        = 12,
   NPF_DS_ACTION_QUEUE         = 13,
   NPF_DS_ACTION_SETPSC        = 14,
   NPF_DS_ACTION_SETTRUST      = 15
```

```
} NPF_DS_ActionType_t;


/* NOP Action */
typedef struct {
   /* No parameters */
} NPF_DS_ActionNOP_t;


/* Counter Action */
typedef struct {
   NPF_DS_QoSHandle_t counterDataHandle;
} NPF_DS_ActionCounter_t;


struct NPF_DS_Action; /* forward definition of action structure
*/

/* Police Action */
typedef struct {
   NPF_DS_QoSHandle_t       policerDataHandle;
   struct NPF_DS_Action    *actionGreen;
   NPF_uint16_t             actionGreenSize;
   struct NPF_DS_Action    *actionYellow;
   NPF_uint16_t             actionYellowSize;
   struct NPF_DS_Action    *actionRed;
   NPF_uint16_t             actionRedSize;
} NPF_DS_ActionPolice_t;


/* Set DSCP Action */
typedef struct {
   NPF_uchar8_t dscp;
} NPF_DS_ActionSetDSCP_t;


/* Set IP Precedence Action */
typedef struct {
   NPF_uchar8_t ipprec;
} NPF_DS_ActionSetIPPREC_t;


/* Set NextHop Action */

typedef enum {
   NPF_DS_ROUTE_TYPE_NEXT_HOP_IPV4    = 0,
   NPF_DS_ROUTE_TYPE_NEXT_HOP_IPV6    = 1,
   NPF_DS_ROUTE_TYPE_EGRESS_INTERFACE = 2
} NPF_DS_RouteType_t;

typedef struct {
   NPF_DS_RouteType_t routeType;
   union {
      NPF_IPv4Address_t nextHopIPv4;
      NPF_IPv6Address_t nextHopIPv6;
      NPF_IfHandle_t egressIfaceHandle;
   } routeInfo;
} NPF_DS_RouteList_t;

typedef struct {
   NPF_uint32_t numRouteListEntries;
```

```
   NPF_DS_RouteList_t *routeListArray;
} NPF_DS_ActionSetNextHop_t;


/* Set FIB Action */
typedef struct {
   NPF_IPv4UC_FibTableHandle_t fibTableHandle;
} NPF_DS_ActionSetFIB_t;


/* Color States */
typedef enum {
   NPF_DS_GREEN    = 0,
   NPF_DS_YELLOW   = 1,
   NPF_DS_RED      = 2
} NPF_DS_COLOR_t;

/* Set Color Action */
typedef struct {
   NPF_DS_COLOR_t color;
} NPF_DS_ActionSetColor_t;


/* Downmark Action */
typedef struct {
   NPF_DS_MapTableHandle_t downmarkTableHandle;
} NPF_DS_ActionDownmark_t;


/* Drop Action */
typedef struct {
   /* no action parameters */
} NPF_DS_ActionDrop_t;


/* Shape action */
typedef struct {
   NPF_DS_QoSHandle_t shaperDataHandle;
} NPF_DS_ActionShape_t;


/* Forward Action */
typedef struct {
   /* No parameters */
} NPF_DS_ActionForward_t;


/* Return Action */
typedef struct {
   /* No parameters */
} NPF_DS_ActionReturn_t;


/* Queue action */
typedef struct {
   NPF_DS_QoSHandle_t queueHandle;
} NPF_DS_ActionQueue_t;


/* PHB Scheduling Classes */
typedef enum{
```

```
   NPF_DS_PSC_EF = 0,
   NPF_DS_PSC_AF1 = 1,
   NPF_DS_PSC_AF2 = 2,
   NPF_DS_PSC_AF3 = 3,
   NPF_DS_PSC_AF4 = 4,
   NPF_DS_PSC_BE = 5
} NPF_DS_PSC_Type_t;


/* Set PHB Scheduling Class (PSC) action */
typedef struct {
   NPF_DS_PSC_Type_t pscType;
   NPF_DS_QoSHandle_t pscHandle; /* similar to the queue
                                    handle */
} NPF_DS_ActionSetPSC_t;


/* Trust States */
typedef enum {
   NPF_DS_TRUST_UNTRUSTED   = 0,
   NPF_DS_TRUST_TRUSTDSCP   = 1,
   NPF_DS_TRUST_TRUSTIPPREC = 2,
   NPF_DS_TRUST_TRUSTCOS    = 3,
   NPF_DS_TRUST_UNSPEC      = 4
} NPF_DS_TRUST_STATE_t;


/* Set Local Trust Mode Action */
typedef struct {
   NPF_DS_TRUST_STATE_t trust;
} NPF_DS_ActionSetTrust_t;


/* ECN States */
typedef enum {
   NPF_DS_ECN_UNAWARE = 0,
   NPF_DS_ECN_AWARE   = 1
} NPF_DS_ECN_MODE_t;


/* Set GLOBAL ECN Mode Action */
typedef struct {
   NPF_DS_ECN_MODE_t ecn;
} NPF_DS_ECN_t;


/* Action Entry */
typedef struct NPF_DS_Action {
   NPF_DS_ActionID_t actionID;
   NPF_DS_ActionType_t actionType;
   union {
      NPF_DS_ActionNOP_t            nop;
      NPF_DS_ActionCounter_t        counter;
      NPF_DS_ActionPolice_t         police;
      NPF_DS_ActionSetDSCP_t        setdscp;
      NPF_DS_ActionSetIPPREC_t      setipprec;
      NPF_DS_ActionSetNextHop_t     setnexthop;
      NPF_DS_ActionSetFIB_t         setfib;
      NPF_DS_ActionSetColor_t       setcolor;
      NPF_DS_ActionDownmark_t       downmark;
      NPF_DS_ActionDrop_t           drop;
```

```
      NPF_DS_ActionShape_t          shape;
      NPF_DS_ActionForward_t        forward;
      NPF_DS_ActionReturn_t         retrn;
      NPF_DS_ActionQueue_t          queue;
      NPF_DS_ActionSetPSC_t         setpsc;
      NPF_DS_ActionSetTrust_t       settrust;
   } actionData;
   struct NPF_DS_Action *subactionArray;
   NPF_uint16_t subactionArraySize;
} NPF_DS_Action_t;


/*
 * Mapping Table-related data structures + definitions
 */


/* Mapping Table Types... */
typedef enum {
   NPF_DS_DSCP_L2COS    = 0,
   NPF_DS_L2COS_DSCP    = 1,
   NPF_DS_DOWNMARK_DSCP = 2,
   NPF_DS_DSCP_PHB      = 3,
   NPF_DS_IPPREC_DSCP   = 4
} NPF_DS_MapTableType_t;


#define NPF_DS_MAPTABLE_L2COS_DSCP_MAP_SIZE      8
#define NPF_DS_MAPTABLE_DOWNMARK_DSCP_MAP_SIZE  64
#define NPF_DS_MAPTABLE_DSCP_PHB_MAP_SIZE       64
#define NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE     64
#define NPF_DS_MAPTABLE_IPPREC_DSCP_MAP_SIZE     8


typedef union {
   NPF_uint16_t
       l2cos_dscp_map[NPF_DS_MAPTABLE_L2COS_DSCP_MAP_SIZE];
   NPF_uint16_t
       downmark_dscp_map[NPF_DS_MAPTABLE_DOWNMARK_DSCP_MAP_SIZE];
   NPF_uint16_t
       dscp_phb_map[NPF_DS_MAPTABLE_DSCP_PHB_MAP_SIZE];
   NPF_uint16_t
       dscp_l2cos_map[NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE];
   NPF_uint16_t
       ipprec_dscp_map[NPF_DS_MAPTABLE_DSCP_L2COS_MAP_SIZE];
} NPF_DS_MapTableData_t;


/* Mapping Table */
typedef struct {
   NPF_DS_MapTableType_t type;
   NPF_DS_MapTableData_t map;
} NPF_DS_MapTable_t;


/*
* QoS Object-related data structures + definitions
*/

/* QoS Object Types */
typedef enum {
```

```
   NPF_DS_QOS_POLICER          = 0,
   NPF_DS_QOS_SHAPER           = 1,
   NPF_DS_QOS_COUNTER          = 2,
   NPF_DS_QOS_SCHEDULER        = 3,
   NPF_DS_QOS_SCHEDULER_CONFIG = 4,
   NPF_DS_QOS_QUEUE            = 5,
   NPF_DS_QOS_TRUST_STATE      = 6
} NPF_DS_QoSObjectType_t;


/* AQM types */
typedef enum {
   NPF_DS_AQM_HEAD = 0,
   NPF_DS_AQM_TAIL = 1,
   NPF_DS_AQM_RED  = 2,
   NPF_DS_AQM_WRED = 3
} NPF_DS_AQMType_t;


/* Scheduler Config Types */
typedef enum {
    NPF_DS_PQ    = 0, /* Priority Queuing */
    NPF_DS_CBQ   = 1, /* Class Based Queuing */
    NPF_DS_CBWFQ = 2, /* Class Based Weighted
                         Fair Queuing */
} NPF_DS_SchedulerConfigType_t;


/* Scheduler Types */
typedef enum {
    NPF_DS_PRI = 0,
    NPF_DS_BSP = 1,
    NPF_DS_WFQ = 2,
    NPF_DS_WRR = 3,
    NPF_DS_TB  = 4,
    NPF_DS_DRR = 5
} NPF_DS_SchedulerType_t;


/* Policer Types */
typedef enum {
   NPF_DS_POLICER_SRTCM_COLOR_BLIND = 0,
   NPF_DS_POLICER_SRTCM_COLOR_AWARE = 1,
   NPF_DS_POLICER_TRTCM_COLOR_BLIND = 2,
   NPF_DS_POLICER_TRTCM_COLOR_AWARE = 3
} NPF_DS_PolicerType_t;


/* Policer Data */
typedef struct {
    NPF_DS_PolicerType_t type;
    NPF_uint32_t cir; /* Committed Info Rate */
    NPF_uint32_t pir; /* Peak Info Rate */
    NPF_uint32_t cbs; /* Committed Burst Size */
    NPF_uint32_t ebs; /* Excess Burst Size */
} NPF_DS_PolicerData_t;


/* Shaper Data */
typedef struct {
    NPF_uint32_t cir; /* Committed Info Rate */
```

```
    NPF_uint32_t cbs; /* Committed Burst Size */
    NPF_uint32_t pir; /* Peak Rate */
    NPF_uint32_t pbs; /* Peak burst */
} NPF_DS_ShaperData_t;


/* Scheduler Config Data */
typedef struct {
    NPF_uint16_t prio; /* Priority */
} NPF_DS_SchedulerConfigPrio_t;


typedef struct {
    NPF_uint16_t prio; /* Priority */
    NPF_uint32_t pir;  /* Rate to be bounded on */
} NPF_DS_SchedulerConfigBSP_t;


typedef struct {
    NPF_uint32_t weight;  /* weight */
} NPF_DS_SchedulerConfigWRR_t;


typedef struct {
    NPF_uint32_t weight;  /* weight */
} NPF_DS_SchedulerConfigWFQ_t;


typedef struct {
    NPF_uint32_t pir;     /* Peak Rate */
    NPF_uint32_t cir;     /* Committed Rate */
    NPF_uint32_t pbs;     /* Peak burst */
    NPF_uint32_t cbs;     /* Committed Burst */
} NPF_DS_SchedulerConfigTB_t;


typedef struct {
    NPF_DS_SchedulerConfigType_t configType;
    union {
        NPF_DS_SchedulerConfigPrio_t configPrio;
        NPF_DS_SchedulerConfigBSP_t configBSP;
        NPF_DS_SchedulerConfigWRR_t configWRR;
        NPF_DS_SchedulerConfigWFQ_t configWFQ;
        NPF_DS_SchedulerConfigTB_t configTB;
    } configParam;
} NPF_DS_SchedulerConfigData_t;


/* Scheduler Data */
typedef struct {
   NPF_DS_SchedulerType_t type;
   NPF_uint32_t minRate;                      /* Minimum Rate */
   NPF_uint32_t maxRate;                      /* Maximum Rate */
   NPF_DS_QoSHandle_t downstreamHandle;       /* Handle for the
                                            Downstream   scheduler
*/
   NPF_DS_SchedulerConfigData_t configData;  /* Config Parameters
                                          for  the  downstream  link
*/
} NPF_DS_SchedulerData_t;
```

```
/* AQM Data */
typedef struct {
   NPF_uint64_t bytes;
   NPF_uint64_t packets;
   NPF_uint64_t randomBytes;
   NPF_uint64_t randomPackets;
} NPF_DS_AQMData_t;


/* Queue Data */
typedef struct {
    NPF_DS_AQMType_t type;
    NPF_uint32_t bufferSize;
    NPF_uint32_t minThresh;
    NPF_uint32_t maxThresh;
    NPF_uint16_t maxProb;
    NPF_uint16_t avgWeight;
    NPF_DS_AQMData_t stats;
    NPF_DS_ECN_t ecnMode;
    NPF_DS_SchedulerConfigData_t *schedConfigData;
    NPF_DS_QoSHandle_t schedulerHandle;
} NPF_DS_QueueData_t;


/* Counter Data */
typedef struct {
   NPF_uint64_t bytes;
   NPF_uint64_t packets;
} NPF_DS_CounterData_t;


/* Trust State Data */
typedef struct {
   NPF_DS_TRUST_STATE_t trust;
} NPF_DS_TrustStateData_t;


/* QoS Object Data */
typedef union {
    NPF_DS_PolicerData_t policer;
    NPF_DS_ShaperData_t shaper;
    NPF_DS_SchedulerData_t scheduler;
    NPF_DS_CounterData_t counter;
    NPF_DS_QueueData_t queue;
    NPF_DS_TrustStateData_t trust;
} NPF_DS_QoSObjectData_t;


/* QoS Object */
typedef struct {
   NPF_DS_QoSObjectType_t type;
   NPF_DS_QoSObjectData_t data;
} NPF_DS_QoSObject_t;


/* Array of Counters */
typedef struct {
   NPF_uint32_t numCounters;
   NPF_DS_CounterData_t *counterData;
} NPF_DS_CounterStatsData_t;
```

```
/* Array of AQM Counters */
typedef struct {
   NPF_uint32_t num;
   NPF_DS_AQMData_t *AQMCounters;
} NPF_DS_AQMStatsData_t;


/*
 * Capability Structures
 */


typedef struct {
     NPF_uchar8_t MAC_addr:1,
                  priority:1,
                  VLAN_id :1;
} NPF_DS_L2_FilterCapabilities_t;


typedef struct {
     NPF_uchar8_t IPSA       :1,
                  IPSA_prefix:1,
                  IPDA       :1,
                  IPDA_prefix:1,
                  protocol   :1,
                  TOS_byte   :1,
                  fragments  :1;
} NPF_DS_L3_IPv4_FilterCapabilities_t;


typedef struct {
     NPF_uchar8_t IPSA       :1,
                  IPSA_prefix:1,
                  IPDA       :1,
                  IPDA_prefix:1,
                  class      :1,
                   flowLabel  :1,
                  nextHeader :1;
} NPF_DS_L3_IPv6_FilterCapabilities_t;


typedef struct {
        NPF_uchar8_t SRCP      :1,
                     SRCP_range:1,
                    DSTP       :1,
                     DSTP_range:1,
                    TCP_flags :1;
} NPF_DS_L4_TCPUDP_FilterCapabilities_t;


typedef struct {
     NPF_uchar8_t type:1,
                  code:1;
} NPF_DS_L4_ICMP_FilterCapabilities_t;


typedef struct {
        NPF_uchar8_t NOP       :1,
                     counter   :1,
```

```
                    policer   :1,
                    setDSCP   :1,
                    setIPREC  :1,
                    setNextHop:1,
                    setFIB    :1,
                    setColor  :1,
                    downmark  :1,
                    drop      :1,
                    shape     :1,
                    forward   :1,
                    queue     :1,
                     setPSC    :1,
                    setTrust  :1,
                    retrn     :1;
} NPF_DS_L3_ActionCapabilities_t;


typedef struct {
        NPF_uchar8_t NOP       :1,
                     counter   :1,
                     policer   :1,
                     setNextHop:1,
                     setFIB    :1,
                     setColor  :1,
                     downmark  :1,
                     drop      :1,
                     shape     :1,
                     forward   :1,
                     queue     :1,
                     setPSC    :1,
                     retrn     :1;
} NPF_DS_L2_Encap_ActionCapabilities_t;


typedef struct {
        NPF_uchar8_t NOP       :1,
                     counter   :1,
                     policer   :1,
                     setDSCP   :1,
                     setIPREC  :1,
                     setNextHop:1,
                     setFIB    :1,
                     setColor  :1,
                     downmark  :1,
                     drop      :1,
                     forward   :1,
                     queue     :1,
                     setPSC    :1,
                     setTrust  :1,
                     retrn     :1;
} NPF_DS_L2_Decap_ActionCapabilities_t;


typedef struct {
        NPF_uchar8_t SP  :1,
                     WFQ :1,
                     WRR :1,
                     TB  :1;
} NPF_DS_SchedulingCapabilities_t;
```

```
typedef struct {
        NPF_uchar8_t L2COS_to_DSCP :1,
                     L2COS_to_PHB  :1,
                     DSCP_to_L2COS :1,
                     DSCP_to_PHB   :1,
                     IPPREC_TO_DSCP:1,
                     DSCP_TO_IPPREC:1;
} NPF_DS_MappingTablesCapabilities_t;


typedef struct {
      NPF_DS_L2_FilterCapabilities_t capabilityFilterL2;
      NPF_DS_L2_Encap_ActionCapabilities_t  capabilityActions;

      /* Overflow bits */

       /* if there is no policy set on a subordinate, the policy
        * of the next higher level interface must be checked/used
        */
      NPF_uchar8_t inheritParentPolicy             :1,

       /* even if there is a policy set on a subordinate, if the
        * packet does not match any policy in the policy table,
        *  then  the  parent  interface's  policy  table  must  be
applied.
        */

                  inheritParentPolicyNoMatch       :1,

       /* same as the previous, but also allow to go to the
        * parents policy even if the packet already matched
        * something in the local policy table.
        */
                     inheritParentPolicyAlreadyMatch  :1;

} NPF_DS_L2_Encap_PlaneCapabilities_t;


typedef struct {
      NPF_DS_L2_FilterCapabilities_t       capabilityFilterL2;
      NPF_DS_L2_Decap_ActionCapabilities_t capabilityActions;

      /* Overflow bits */
      NPF_uchar8_t inheritParentPolicy             :1,
                   inheritParentPolicyNoMatch      :1,
                    inheritParentPolicyAlreadyMatch  :1;
} NPF_DS_L2_Decap_PlaneCapabilities_t;


typedef struct {
      NPF_DS_L3_IPv4_FilterCapabilities_t
                 capabilityFilterL3_IPv4;
      NPF_DS_L3_IPv6_FilterCapabilities_t
                 capabilityFilterL3_IPv6;
      NPF_DS_L4_TCPUDP_FilterCapabilities_t
                 capabilityFilterL4_TCPUDP;
      NPF_DS_L4_ICMP_FilterCapabilities_t
                 capabilityFilterL4_ICMP;
      NPF_DS_L3_ActionCapabilities_t  capabilityActions;

      /* Overflow bits */
```

```
         /* if there is no policy set on a subordinate, the
          * policy of the next higher level interface
          * must be checked/used
          */
        NPF_uchar8_t inheritParentPolicy             :1,

         /* even if there is a policy set on a subordinate,
          * if the packet does not match any policy in the
          * policy table, then the parent interface's policy
          * table must be applied.
          */
                     inheritParentPolicyNoMatch      :1,

         /* same as the previous, but also allow to go to the
          * parents policy even if the packet already
          * matched something in the local policy table.
          */
                     inheritParentPolicyAlreadyMatch  :1;
} NPF_DS_L3_PlaneCapabilities_t;


typedef struct {
      /* Filter & Action Capabilities */
      NPF_DS_L2_Encap_PlaneCapabilities_t
                     FilterActionCapL2_encap;
      NPF_DS_L2_Decap_PlaneCapabilities_t
                     FilterActionCapL2_decap;
      NPF_DS_L3_PlaneCapabilities_t FilterActionCapIP_ingress;
      NPF_DS_L3_PlaneCapabilities_t FilterActionCapIP_egress;
      NPF_DS_L3_PlaneCapabilities_t
                     FilterActionCapLocallyTerminated;
      NPF_DS_L3_PlaneCapabilities_t
                     FilterActionCapLocallyGenerated;

      /* Queuing Capabilities */
      NPF_DS_SchedulingCapabilities_t QueueCapIP_ingress;
      NPF_DS_SchedulingCapabilities_t QueueCapIP_egress;
      NPF_DS_SchedulingCapabilities_t QueueCapLocallyTerminated;
      NPF_DS_SchedulingCapabilities_t QueueCapLocallyGenerated;

      /* Mapping Table Capabilities */
      NPF_DS_MappingTablesCapabilities_t MappingTablesCap;

      /* Plane existence */
      NPF_uchar8_t planeL2_Encap:1,
                   planeL2_Decap:1,
                  planeL3_Ingress:1,
                  planeL3_Egress:1,
                  planeL3_LocallyTerminated:1,
                  planeL3_LocallyGenerated:1;
} NPF_DS_InterfaceCapabilities_t;


typedef NPF_uint32_t NPF_DS_CapabilityProfileID_t;


typedef struct {
   NPF_DS_CapabilityProfileID_t profileID;
   NPF_DS_InterfaceCapabilities_t capabilityMatrix;
} NPF_DS_CapabilityProfileInfo_t;
```

```
/*
 * Asynchronous error codes (returned in function callbacks)
 */


typedef NPF_uint32_t NPF_DS_ReturnCode_t;

#define S_DS_ERR(n) ((NPF_DS_ReturnCode_t) \
                        (NPF_DS_BASE_ERR+(n))
#define NPF_DS_POLICY_INVALID_HANDLE          S_DS_ERR(0)
#define NPF_DS_POLICY_INVALID_INPUT_PARAM     S_DS_ERR(1)
#define NPF_DS_POLICY_BOUND                   S_DS_ERR(2)
#define NPF_DS_POLICY_BOUND_IFACE             S_DS_ERR(3)
#define NPF_DS_POLICY_NOT_BOUND               S_DS_ERR(4)
#define NPF_DS_POLICY_NOT_BOUND_IFACE         S_DS_ERR(5)
#define NPF_DS_POLICY_DOES_NOT_EXIST          S_DS_ERR(6)
#define NPF_DS_MAPTABLE_INVALID_HANDLE        S_DS_ERR(7)
#define NPF_DS_MAPTABLE_INVALID_INPUT_PARAM   S_DS_ERR(8)
#define NPF_DS_MAPTABLE_BOUND                 S_DS_ERR(9)
#define NPF_DS_MAPTABLE_BOUND_IFACE           S_DS_ERR(11)
#define NPF_DS_MAPTABLE_NOT_BOUND             S_DS_ERR(12)
#define NPF_DS_MAPTABLE_NOT_BOUND_IFACE       S_DS_ERR(13)
#define NPF_DS_MAPTABLE_DOES_NOT_EXIST        S_DS_ERR(14)
#define NPF_DS_QOS_OBJECT_INVALID_HANDLE      S_DS_ERR(15)
#define NPF_DS_QOS_OBJECT_INVALID_INPUT_PARAM S_DS_ERR(16)
#define NPF_DS_QOS_OBJECT_BOUND               S_DS_ERR(17)
#define NPF_DS_QOS_OBJECT_BOUND_IFACE         S_DS_ERR(18)
#define NPF_DS_QOS_OBJECT_NOT_BOUND           S_DS_ERR(19)
#define NPF_DS_QOS_OBJECT_NOT_BOUND_IFACE     S_DS_ERR(20)
#define NPF_DS_QOS_OBJECT_DOES_NOT_EXIST      S_DS_ERR(21)


/* Policy Object Configure Response Structure */
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
} NPF_DS_PolicyCreateResponse_t;


/* Table Configure Response Structure */
typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
} NPF_DS_MapTableCreateResponse_t;


/* QoS Object Configure Response Structure */
typedef struct {
   NPF_DS_QoSHandle_t qosHandle;
} NPF_DS_QoSObjectCreateResponse_t;


/* Policy Update Response Structure */
typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_DS_PolicyID_t resourceID;
} NPF_DS_PolicyUpdateResponse_t;


/* Table Update Response Structure */
typedef struct {
```

```
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableID_t resourceID;
} NPF_DS_MapTableUpdateResponse_t;


/* QoS Object Update Response Structure */
typedef struct {
   NPF_DS_QoSHandle_t qosHandle;
   NPF_DS_QoSObjectID_t resourceID;
} NPF_DS_QoSObjectUpdateResponse_t;


typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_DS_PolicyID_t resourceID;
   NPF_char8_t configureFlag;
} NPF_DS_PolicyQueryGetHandleInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_PolicyQueryGetHandleResponse_t;


typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
   NPF_uint32_t filterSize;
   NPF_DS_Rule_t *filter;
   NPF_uint32_t actionArraySize;
   NPF_DS_Action_t *actions;
} NPF_DS_PolicyQueryGetContentInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_PolicyQueryGetContentResponse_t;


typedef struct {
   NPF_DS_PolicyHandle_t policyHandle;
} NPF_DS_PolicyQueryGetBoundObjectInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_PolicyQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
   NPF_IfHandle_t ifaceHandle;
   NPF_DS_FilterPlane_t plane;
} NPF_DS_PolicyQueryGetBoundObjectResponse_t;


typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectID_t resourceID;
   NPF_DS_QoSObjectType_t type;
   NPF_char8_t configureFlag;
```

```
} NPF_DS_QoSObjectQueryGetHandleInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_QoSObjectQueryGetHandleResponse_t;


typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectData_t data;
   NPF_DS_QoSObjectType_t type;
} NPF_DS_QoSObjectQueryGetContentInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_QoSObjectQueryGetContentResponse_t;


typedef struct {
   NPF_DS_QoSHandle_t objectHandle;
   NPF_DS_QoSObjectType_t type;
} NPF_DS_QoSObjectQueryGetBoundObjectInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_QoSObjectQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
   NPF_IfHandle_t ifaceHandle;
   NPF_DS_FilterPlane_t plane;
} NPF_DS_QoSObjectQueryGetBoundObjectResponse_t;


typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableID_t resourceID;
   NPF_DS_MapTableType_t tableType;
   NPF_char8_t configureFlag;
} NPF_DS_MapTableQueryGetHandleInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetHandleInfoResponse_t
     *handleInfoArray;
} NPF_DS_MapTableQueryGetHandleResponse_t;


typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableType_t type;
   NPF_DS_MapTableData_t data;
} NPF_DS_MapTableQueryGetContentInfoResponse_t;
```

```
typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetContentInfoResponse_t
     *contentInfoArray;
} NPF_DS_MapTableQueryGetContentResponse_t;


typedef struct {
   NPF_DS_MapTableHandle_t tableHandle;
   NPF_DS_MapTableType_t type;
} NPF_DS_MapTableQueryGetBoundObjectInfoResponse_t;


typedef struct {
   NPF_uint32_t count;
   NPF_DS_MapTableQueryGetBoundObjectInfoResponse_t
     *contentInfoArray;
   NPF_IfHandle_t ifaceHandle;
} NPF_DS_MapTableQueryGetBoundObjectResponse_t;


typedef struct {
   NPF_uint32_t profileCount;
   NPF_DS_CapabilityProfileInfo_t
              *capabilityProfileInfoArray;
} NPF_DS_CapabilityProfileQueryResponse_t;


typedef struct {
   NPF_uint32_t profileIDCount;
   NPF_DS_CapabilityProfileID_t *profileIDArray;
} NPF_DS_CapabilityInterfaceQueryResponse_t;


/* Common Response Structure */
typedef struct {
   NPF_DS_ReturnCode_t returnCode;
   union {
      NPF_DS_PolicyCreateResponse_t policyCreate;
      NPF_DS_MapTableCreateResponse_t tableCreate;
      NPF_DS_QoSObjectCreateResponse_t qosCreate;
      NPF_DS_PolicyUpdateResponse_t policyUpdate;
      NPF_DS_MapTableUpdateResponse_t tableUpdate;
      NPF_DS_QoSObjectUpdateResponse_t qosUpdate;
      NPF_DS_PolicyQueryGetHandleResponse_t policyQueryHandle;
      NPF_DS_PolicyQueryGetContentResponse_t policyQueryContent;
      NPF_DS_PolicyQueryGetBoundObjectResponse_t
          policyQueryBoundHandle;
      NPF_DS_QoSObjectQueryGetHandleResponse_t qosQueryHandle;
      NPF_DS_QoSObjectQueryGetContentResponse_t qosQueryContent;
      NPF_DS_QoSObjectQueryGetBoundObjectResponse_t
          qosQueryBoundHandle;
      NPF_DS_MapTableQueryGetHandleResponse_t tableQueryHandle;
      NPF_DS_MapTableQueryGetContentResponse_t tableQueryContent;
      NPF_DS_MapTableQueryGetBoundObjectResponse_t
          tableQueryBoundHandle;
      NPF_DS_CapabilityProfileQueryResponse_t
          capabilityProfileQuery;
      NPF_DS_CapabilityInterfaceQueryResponse_t
          capabilityInterfaceQuery;
      NPF_DS_CounterStatsData_t counterStatsData;
```

```
      NPF_DS_AQMStatsData_t AQMStatsData;
   } returnData;
} NPF_DS_AsyncResponse_t;


/* Common callback definition */
typedef enum {
   NPF_DS_POLICY_CREATE = 0,
   NPF_DS_MAP_TABLE_CREATE = 1,
   NPF_DS_QOS_OBJECT_CREATE = 2,
   NPF_DS_POLICY_UPDATE = 3,
   NPF_DS_MAP_TABLE_UPDATE = 4,
   NPF_DS_QOS_OBJECT_UPDATE = 5,
   NPF_DS_POLICY_BIND_IFACE = 6,
   NPF_DS_POLICY_UNBIND_IFACE = 7,
   NPF_DS_QOS_OBJECT_BIND_IFACE = 8,
   NPF_DS_QOS_OBJECT_UNBIND_IFACE = 9,
   NPF_DS_MAP_TABLE_BIND_IFACE = 10,
   NPF_DS_MAP_TABLE_UNBIND_IFACE = 11,
   NPF_DS_QUERY_COUNTER_STATS = 12,
   NPF_DS_QUERY_AQM_STATS = 13,
   NPF_DS_POLICY_DESTROY = 14,
   NPF_DS_MAP_TABLE_DESTROY = 15,
   NPF_DS_QOS_OBJECT_DESTROY = 16,
   NPF_DS_COUNTERS_CLEAR = 17,
   NPF_DS_POLICY_QUERY_GET_HANDLE = 18,
   NPF_DS_POLICY_QUERY_GET_CONTENT = 19,
   NPF_DS_POLICY_QUERY_GET_BOUND_OBJECT = 20,
   NPF_DS_QOS_OBJECT_QUERY_GET_HANDLE = 21,
   NPF_DS_QOS_OBJECT_QUERY_GET_CONTENT = 22,
   NPF_DS_QOS_OBJECT_QUERY_GET_BOUND_OBJECT = 23,
   NPF_DS_MAP_TABLE_QUERY_GET_HANDLE = 24,
   NPF_DS_MAP_TABLE_QUERY_GET_CONTENT = 25,
   NPF_DS_MAP_TABLE_QUERY_GET_BOUND_OBJECT = 26,
   NPF_DS_CAPABILITY_PROFILE_QUERY = 27,
   NPF_DS_CAPABILITY_INTERFACE_QUERY = 28
} NPF_DS_CallbackType_t;


/* Callback Data */
typedef struct {
   NPF_DS_CallbackType_t type;
   NPF_boolean_t allOK;
   NPF_uint32_t numCallbackResp;
   NPF_DS_AsyncResponse_t *resp;
} NPF_DS_CallbackData_t;


/* Callback Function Pointer Prototype */
typedef void (*NPF_DS_CallbackFunc_t) (
   NPF_IN NPF_userContext_t userContext,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_CallbackData_t data);


/*
 * Diffserv Services API Function Prototypes
 */


NPF_error_t NPF_DS_Register(
```

```
   NPF_IN NPF_userContext_t userContext,
   NPF_IN NPF_DS_CallbackFunc_t callbackFunc,
   NPF_OUT NPF_callbackHandle_t *callbackHandle);

NPF_error_t NPF_DS_Deregister(
   NPF_IN NPF_callbackHandle_t callbackHandle);

NPF_error_t NPF_DS_PolicyCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_PolicyID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting);

NPF_error_t NPF_DS_PolicyUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_uint32_t filterSize,
   NPF_IN NPF_DS_Rule_t *filter,
   NPF_IN NPF_uint32_t actionArraySize,
   NPF_IN NPF_DS_Action_t *actions);

NPF_error_t NPF_DS_PolicyDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle);

NPF_error_t NPF_DS_PolicyBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_PolicyUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_PolicyQueryGetHandles(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);

NPF_error_t NPF_DS_PolicyQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
   NPF_IN NPF_DS_PolicyHandle_t *handles);

NPF_error_t NPF_DS_PolicyQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
```

```
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_MapTableCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_MapTableID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableType_t type);

NPF_error_t NPF_DS_MapTableUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_DS_MapTableData_t data);

NPF_error_t NPF_DS_MapTableDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle);

NPF_error_t NPF_DS_MapTableQueryGetHandles(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);

NPF_error_t NPF_DS_MapTableQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
   NPF_IN NPF_DS_MapTableHandle_t *handles);

NPF_error_t NPF_DS_MapTableQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_IfHandle_t ifaceHandle);

NPF_error_t NPF_DS_QoSObjectCreate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_DS_QoSObjectID_t resourceID,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSObjectType_t type);

NPF_error_t NPF_DS_QoSObjectUpdate(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle,
   NPF_IN NPF_DS_QoSObjectData_t data);

NPF_error_t NPF_DS_QoSObjectDestroy(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle);
```

```
NPF_error_t NPF_DS_QoSObjectQueryGetHandles(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);

NPF_error_t NPF_DS_QoSObjectQueryGetContents(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t handleArraySize,
   NPF_IN NPF_DS_QoSHandle_t *handles);

NPF_error_t NPF_DS_QoSObjectQueryGetBoundObjects(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_MapTableBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle);

NPF_error_t NPF_DS_MapTableUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_MapTableHandle_t tableHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle);

NPF_error_t NPF_DS_QoSObjectBind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_QoSObjectUnbind(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_QoSHandle_t objectHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_PolicyCountersGet(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_DS_PolicyHandle_t policyHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_QueueCountersGet(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
```

```
   NPF_IN NPF_DS_QoSHandle_t qosHandle,
   NPF_IN NPF_IfHandle_t ifaceHandle,
   NPF_IN NPF_DS_FilterPlane_t plane);

NPF_error_t NPF_DS_CountersClear(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t numCounters,
   NPF_IN NPF_DS_QoSHandle_t *counterHandleArray);

NPF_error_t NPF_DS_CapabilityProfileQuery(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting);

NPF_error_t NPF_DS_CapabilityInterfaceQuery(
   NPF_IN NPF_callbackHandle_t callbackHandle,
   NPF_IN NPF_correlator_t correlator,
   NPF_IN NPF_errorReporting_t errorReporting,
   NPF_IN NPF_uint32_t ifHandlesCount,
   NPF_IN NPF_IfHandle_t *ifHandlesArray);


#ifdef __cplusplus

}

#endif

#endif /* __DSSAPI_H__ */
```

A.4. DIFFSERV SLA EXAMPLE

Now we consider the configuration of a particular Diffserv Service Level Agreement (SLA), as shown in Figure 12. In this SLA, we look for all traffic with IP source address prefix 10.10/16. All such traffic will be subject to a rate limiter, R1 which limits the flow to 5 Mbps. Additionally, this traffic is also checked to see if it is TCP or UDP traffic. If so, the traffic is also subject to rate limiters R2 (1 Mbps) or R3 (500 Kbps), respectively.

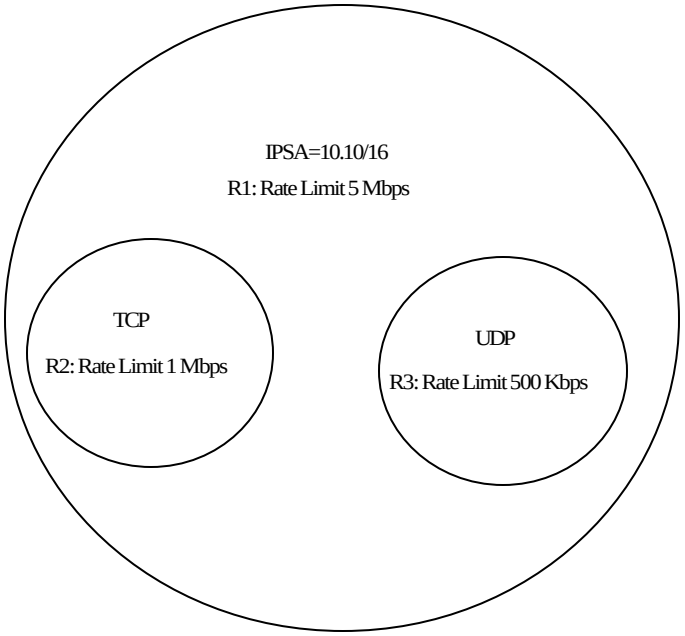

Figure 12: Example Diffserv SLA

This SLA can be configured as shown in Figure 13. In this representation, the top level policy element array contains two elements. The first element points to an OR-First-Match expression which has two elements corresponding to the TCP & UDP cases. Appropriate actions are encoded in these expression elements for each of the two cases, and each points to a rule which specifies the exact pattern that is being matched against. Matching either of these cases will automatically cause the next policy element to be evaluated since this is the default behavior if no terminating action is present. The second policy element contains a single rule which checks to see of the packet matches IPSA 10.10/16. If so, the appropriate rate meter is executed according to the action specified.

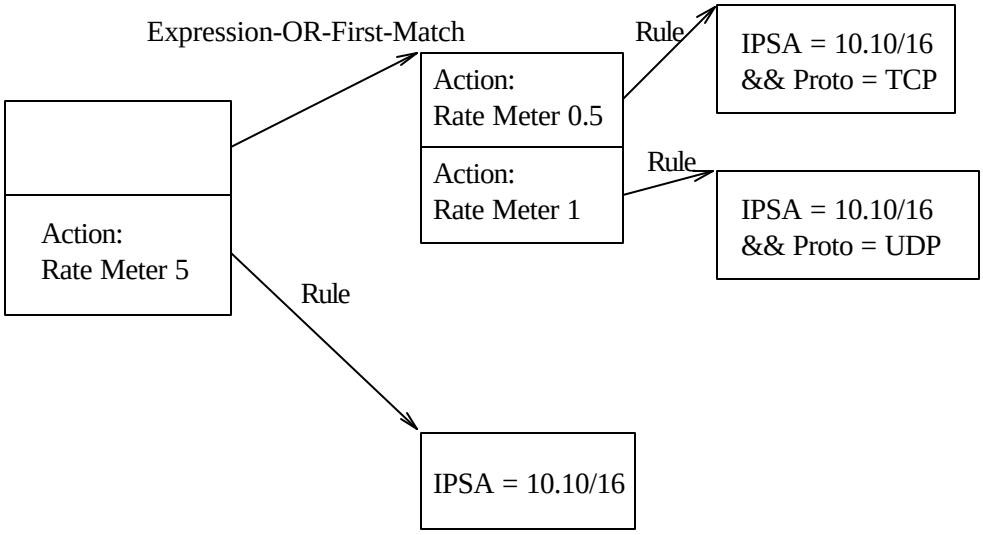

Figure 13: SLA example

APPENDIX B ACKNOWLEDGEMENTS

Working Group Chair: Vinoj Kumar

Task Group Chair: Zsolt Haraszti

Anurag Bhargava	(Ericsson)
Chad Bryant	(ISIC)
Peter Denz	(Ericsson)
Marc Edwards	(ISIC)
Alan Finkelstein	(Ericsson)
Jason Goldschmidt	(Sun Microsystems)
Anand Gorti	(IBM)
David Maxwell	(IDT)
Kannan Babu Ramia	(Intel)
John Renwick	(Agere Systems)
Bala Shankar	(HCL Technologies)
Marcin Spiewak	(Intel)
Michael Speer	(Sun Microsystems)
Sanjeev Verma	(Nokia)

APPENDIX C LIST OF COMPANIES BELONGING TO NPF DURING APPROVAL PROCESS

Agere Systems
Altera
AMCC
Analog Devices
Avici
Cypress Semiconductor
Ericsson
Erlang Technology
ETRI
EZ Chip
Flextronics
FutureSoft
HCL Technologies
Hi/fn
IBM
IDT
Intel
IP Fabrics
IP Infusion
Kawasaki LSI
Motorola
Nokia
Nortel Networks
NTT Electronics
PMC-Sierra
Sun Microsystems
Teja Technologies
TranSwitch
U4EA Group
Xelerated
Xilinx
Zettacom