DiffServ IPv4

Design Specification

Control Plane-Platform Development Kit 2.11

March 2004

intel

Contents

DiffServ IPv4 i

Contents iii

Part 1: Introduction 5

1 Introduction 7
- 1.1 Purpose 7
- 1.2 Scope 7
- 1.3 Terminology 8
- 1.4 References 8
- 1.5 Document Organization 9

Part 2: Overview 11

2 Overview 13
- 2.1 DiffServ Component Overview 13
- 2.2 Assumptions and Dependencies 13

Part 3: DiffServ Component Design 15

3 DiffServ Component Design 17
- 3.1 Managing Data Path Elements (DPE) 17
  - 3.1.1 Data Structures 18
  - 3.1.2 DS_ReqResp_t 18
  - 3.1.3 DiffServRequest_t 19
  - 3.1.4 DS_DpeInfo_t 19
  - 3.1.5 FPPI_DS_Dpe_t 20
- 3.2 Initialization 21
- 3.3 Shutdown 21
- 3.4 API Calls and Callbacks 21

Part 4: Other Design Details 23

4 Other Design Details 25
- 4.1 Memory Allocation 25
- 4.2 Threading Model 25
- 4.3 Dependencies 25

Part 5: Appendix A 27

Revision History

Revision	Description	Date	Author
2.11	Updated for Release 2.11	March 2004	Ds Sreedhara
2.1	Updated for Release 2.1	December 2003	Ds Sreedhara
2.0	Updated for Release 2.0	August 2003	Ds Sreedhara

Part 1: Introduction

1 Introduction

Network elements such as switches and routers can be classified into three logical operational components:

- Control plane
- Forwarding plane
- Management plane

The control plane controls and configures the forwarding plane and the forwarding plane manipulates the network traffic. The control plane executes different signaling or routing protocols and provides all the routing information to the forwarding plane.

The forwarding plane makes decisions based on this information and performs operations on packets such as forwarding, classification, filtering, and so on.

An orthogonal management plane manages the control and forwarding planes. For example, the control plane in a router executes routing protocols, the forwarding plane performs hardware-based switching, and the management plane starts or stops routing process, and performs logging.

The introduction of the standardized Application Program Interface (API) within the above-mentioned planes can help the system vendors, Original Equipment Manufacturer (OEM), and end users of these network elements, to mix and match the components available from the different vendors, and to create a device of their choice.

The Network Processing Forum (NPF) API is designed for this purpose and thus, it presents a flexible and well-known programming interface to the control plane applications. It makes existence of the multiple forwarding planes, and vendor-specific details, transparent to control plane applications. The hardware properties and nature of interconnect used between the control and the forwarding planes are isolated.

This component implements the DiffServ APIs as prescribed by the NPF. This document outlines the design of the DiffServ data structures, algorithms as well as dependencies on other CP-PDK components, if any.

1.1 Purpose

This document specifies the internal design of the DiffServ- Internet Protocol Version 4 (IPv4) component of the CP-PDK. This includes the description and design of the main internal data structures used within the component.

1.2 Scope

This document describes only the design of the DiffServ component. It does not address the individual DiffServ APIs implemented by this component. This document is intended for the developers implementing or maintaining the DiffServ component and the test engineers who are performing Quality Analysis (QA) on the DiffServ component.

1.3 Terminology

Table 1 lists terms used in this document and an expansion of every term.

Table 1. Terminology table

Term	Description
Control Element (CE)	In a separated control/data system, CE refers to the processor(s) responsible for control and configuration of the forwarding elements. It is used interchangeably with the Control Plane (CP).
Control Plane (CP)	Refer Control Element (CE)
DPE	Data Path Element
Forwarding Element (FE)	Refers to the processor(s) responsible for fast path forwarding of the data, in a separated control/data system. It is used interchangeably with the FP.
Forwarding Plane (FP)	Refer Forwarding Element (FE)
NPF	Network Processing Forum
PDK	Platform Development Kit
PIL	Platform Independence Layer

1.4 References

Table 2 lists the documents that are referenced in this document. All the documents listed in this table are included with the CP-PDK.

Table 2. Reference table

Reference	Document Name
[1]	Software Architecture Overview
[2]	Interface Management API Reference
[3]	Forwarding Plane plug-in API Reference
[4]	Namespace Design Reference
[5]	Platform Independence Layer API Reference
[6]	DiffServ Service APIs

1.5 Document Organization

This document is divided into sections describing the design of all the subcomponents, run-time interactions and pseudo code. The data structures are described in the design section instead of a separate section.

Part 2: Overview

2 Overview

This section provides an overview of the DiffServ component and lists the assumptions and dependencies for the component.

2.1 DiffServ Component Overview

The DiffS  odule is split into two logical sub-components:

- **DiffServ API implementation module:** This implementation module is the code that implements the DiffServ APIs. This sub-component is responsible for validating input, and invoking the FP plug-in API to send requests to the forwarding planes.
- **Callback module:** The callback module is responsible for implementing the callback functions that are invoked by the FP Pplug-in API. It invokes the user program's callback to return the response data, if needed.

These two modules run in different contexts and it is split into two different components.

2.2 Assumptions and Dependencies

Following assumptions apply to this design module:

- When any of the FE fails to respond, there is no recovery mechanism. The control plane part waits till the end of all the FE transactions.

The DiffServ API implementation uses the FP plug-in to communicate with the FEs. The FP part module, invokes the core component library for committing the operation in micro blocks.

Part 3: DiffServ Component Design

3 DiffServ Component Design

The primary task of the DiffServ API implementation module is to validate the application input and invoke the FPPAPI [3] calls to the FE. In the multiple-FE topology, the DiffServ request that needs to be broadcasted is committed to all active FEs in the system. To do this, the DiffServ components must maintain the states to keep track of the FE that has reported the result.

After all the FEs, including FE unbind exception, report a success or failure response and the DiffServ component determines the success or failure status of the single request. It is possible to select the FE that responds to send as callback data. The DiffServ component invokes the user callbacks with success/failure after consolidating all the responses from the selected FE.

For each API request, new request state is created, which remains active till the user callback is made. This request state holds all the information required to complete the request with FE. Refer to the data structure `DiffServRequest_t` for individual fields. This state is deleted after invoking the user callback. The request state contains all active FE IDs and sufficient place to hold all responses. On completion of the FE responses, single callback data is prepared from the responses of the selected FE.

In this version, it is assumed that the DiffServ module always receives responses from FE. If any one of the FE fails to respond, no callback is made and the request state is not deleted. If all the FEs response success or failure, the response of the first FE is considered for the application callback. If some of the FEs response success and some failure, response of the first success FE is considered for the user callback. An internal event `DS_FE_RESPONSE` is generated for failed FEs.

On any request of the DPE, state of the DPE is set to `DPE_STATE_INUSE`. DPE remains in this state till the FE response is received. On receipt of the FE response, the DPE state is changed to `DPE_STATE_IDLE. In DPE_STATE_INUSE` state, no other operations are allowed and it generates `NPF_DS_E_DPE_IN_USE` error.

3.1 Managing Data Path Elements (DPE)

For DPE create request, the CP module is responsible to create the DPE handles. It does not maintain any DPE details or DPE associations. All API requests with DPE handle is sent to FP DiffServ module. When DPE is deleted, the handle is also deleted from the DiffServ CP space. Further API requests with this DPE handle results in error. The `FPPI_DS_Dpe_t` data structure is used between the DiffServ module in CP and the FP DiffServ manager in FP.

The DiffServ API invocation goes through the following steps:

- Create new request state. Refer to the `DiffServRequest_t` data structure. This state contains all the active FE information, user information like correlator, callback, error reporting, and callback type.
- In case of the DPE create, find the port associated with the interface handle, and create new DPE info state. `Refer to the DS_DpeInfo_t`. Assign the new DPE handle and send request for all the active FEs. Increment the request count for the FE. Save the DPE info in the `DpeList`. After accepting all the requests, the request state contains total requests sent to each FE. This request state is active, till request count of the FEs equals response count of all the FEs. When the request count of the FEs equals response count of the FEs, the user callback is invoked.

- For all the other requests, check for a valid DPE handle. If the handle is valid, FPPI request is generated, else an error is generated.
- When response is received from the FEs, assign response data and error to the request state, mark the request state to indicate that the response is from the FE. In case of internal errors, the request state is marked as an internally generated response. This information is used to construct the callback data.
- Create new async response using the FE response data. If the number of FEs requests is equal to the number of FEs responses, consolidate all the async responses of the selected FE into the callback data and invoke the user callback function. Delete all the async responses created and the request state. It is assumed that FEs always responds to all the requests made. If it fails to respond, the user callback is not invoked and the request state remains active.

3.1.1 Data Structures

The following section describes the important data structures used in Data Structure (DS) CP module implementation.

3.1.2 DS_ReqResp_t

This block supports the following key features:

- Holds all the FE IDs in the system
- Holds the number of requests made by an application
- Holds the number of responses received from the FE
- Holds the field used, which informs whether request is sent to the FE or not
- Holds the field done, which informs the FE responses for all the requests made
- Holds the field error, which indicates the error reported by the FE
- Holds async responses, which are generated based on the FE response

When requests equal responses, the responses in the async field are used to construct the call back data.

```
typedef struct DS_ReqResp
{
    FPPI_FEID       id;      /* FE ID*/
    uint32_t        used;    /* tells this request sent to this FE
or not */
    uint32_t        reqs;    /* No. Of APIs called for this FE */
    uint32_t        resps;   /* No. Of API responses received for
this FE */
    uint32_t        done;    /* FE responded for all*/
    uint32_t        error;   /* Error reported */
    NPF_DS_AsyncResponse_t *async [DS_MAX_REQS];/* Array of async
responses */
} DS_ReqResp_t;
```

3.1.3 DiffServRequest_t

This block supports the following key features:

- Interface between the application requests and responses to the application
- Holds the application information such as context, correlator, error reporting, call back type and the application callback
- Holds the number of active FE in the system and it has enough memory to hold the responses of FE
- Holds the number of inputs supplied by the application field to the API and informs whether the response is from the FE or generated internally

This data structure is used to correlate the requests and responses of the FEs. When the DiffServ API request is made, the request object is created and it remains active till the call back is made. On each and every receipt of the FE response, new async response is created using the FE response data and stored in the async field of the `reqResp`.

The FE ID is used to distinguish the responses from different FEs. If the FE response is equal to the requests made, the callback data is created using the responses of the first FE. All other stored asynchronous responses are deleted. The field callback in the `DiffServRequest_t` is the user callback function that was registered before. On invoking the callback, the request object is removed from the DiffServ module.

```
typedef struct
{
    NPF_DS_CallbackType_t type;      /* Callback type */
    NPF_callbackHandle_t    cbh;     /* Callback handle */
    NPF_userContext_t       context; /* User context */
    NPF_correlator_t        corr;    /* User correlator */
    NPF_errorReporting_t    report;  /* user error report */
    uint32_t                no;      /* Number of requests
generated towards  FE */
    uint32_t                fec;     /* Total no of active FEs */
    DS_ReqResp_t            *reqResp;/* Array, with size equal to
fec */
    NPF_DS_CallbackFunc_t callback;  /* User registered callback
function */
    NPF_error_t             error;   /* Error code */
    NPF_boolean_t        feResp;     /* TRUE means callback from
FE, otherwise internal */
} DiffServRequest_t;
```

3.1.4 DS_DpeInfo_t

This block supports the following key features:

- Field `type`: which indicates the DPE type
- Field `dpeHandle`: which indicates the DPE handle

- Field `fec`: which indicates the total number of FE that exists in the DPE
- Field `feid`: which is an array of the FEids that exists in the DPE
- Field `port`: which is the port ID created by the DPE
- Field `direction`: which indicates the DPE created in the ingress or the egress side

This block is used to hold the DPE information in the DiffServ CP module. This object is created during the creation of a new DPE and remains active till the deletion of the DPE.

Note: The DiffServ CP module does not maintain any DPE attributes and associations. Whenever an API request is made with the `dpeHandle` as a parameter, the `dpeHandle` is used to validate the DPE.

```
typedef struct DS_DpeInfo
{
NPF_DS_DpeType_t          type;        /* DPE type */
NPF_DS_DpeHandle_t        dpeHandle;   /* DPE handle */
uint32_t                  fec;         /* FE Count */
FPPI_FEID                 feid [MAXFENUM];
FPPI_PortID               port;
NPF_DS_DpeDirection_t     dir; /* DPE direction INGRESS/EGRESS */
} DS_DpeInfo_t;
```

3.1.5 FPPI_DS_Dpe_t

This block supports the following key features:

- Field `op,`: **which** indicates the operation requested by an application on DPE
- Field `type,`: **which** indicates the operation type requested on the DPE
- Field `error:`, **which** indicates the error reported by the FE, and is valid only in the response
- Filed `dpeHandle:` which indicates the DPE handle
- Field `prevDpeHandle:` is the DPE handle, with which association is to be made for delete. It is valid only in the DPE association add or delete
- Field `port:` which is the port IDof the DPE
- Field `dir:` indicates the DPE that exists in the INGRESS or the EGRESS side

This structure is used for messaging between the DiffServ CP module and the DiffServ manager. The `dpe` field is valid for create and query operations and the stats field is valid for getting the statistics operation.

```
typedef struct
{
    NPF_DS_CallbackType_t      op;             /* DPE Operation */
    NPF_DS_DpeType_t           type;       /* DPE Type */
    NPF_DS_ErrorType_t         error;      /* Error Code */
    NPF_DS_DpeHandle_t         dpeHandle;  /* DPE handle */
```

```
    NPF_DS_DpeHandle_t          prevDpeHandle;  /* DPE handle */
    FPPI_PortID                 port;       /* Port ID*/
    NPF_DS_DpeDirection_t       dir;            /* Direction,
INGRESS/EGRESS */
    union
    {
        NPF_DS_Dpe_t            dpe;          /* Data path element
*/
        NPF_DS_StatsResp_t      stats;
    } u;
} FPPI_DS_Dpe_t;
```

3.2 Initialization

The PDK manager must invoke the DiffServ initialization for the DiffServ component. The initialization routine has the following requirements:

- The FP plug-in API, namespace, and component should be initialized before the DiffServ initialization.
- The DiffServ module-wide critical section lock gets initialized here.
- The control plane callback of the DiffServ module is registered with the forwarding plane.
- All the dynamic data structures DList gets initialized here.

3.3 Shutdown

The PDK manager calls the DiffServ shutdown routine. This routine is responsible for releasing any resources currently in use and terminating the communication with other internal components. The shutdown requirements are as follows:

- The namespace and FP plug-in modules should be shut down after the DiffServ shutdown.
- The DiffServ shutdown must deregister all the FP plug-in callback services by passing the FP Pplug-in callback handle to the FP plug-in deregister call.
- Clean up all the resources and dynamic data structures such as DLists.

3.4 API Calls and Callbacks

The applications must register their callback functions with the DiffServ services to make the DiffServ API calls. Except for the register and deregister APIs, all the other DiffServ routines are asynchronous calls.

When any API is called, it returns one of following:

- NPF_NO_ERROR: The operation is successful. Except during the register & deregister of the APIs, the application should wait for completion of thecallback operation.
- NPF_E_UNKNOWN: An unknown error has occurred.
- NPF_E_BAD_CALLBACK_HANDLE: The callback handle passed is invalid.
- NPF_E_BAD_CALLBACK_FUNCTION: This is returned only in case of the register APIs. It indicates that the callback function passed in NPF_DS_Register () is NULL.

The DiffServ callback component keeps track of the outstanding requests and collects all the necessary FE(s) responses for such requests.

Part 4: Other Design Details

4 Other Design Details

4.1 Memory Allocation

The PDK DiffServ implementation always makes a copy of the user program request data, that isi.e., entries passed in API. During the time of the callback, the DiffServ callback module also copies the FP Pplug-in response data into a new memory location and pushes it to a callback thread to execute the application's callback.

The callback thread frees this data memory immediately on return of the application callback. The user's program should make a copy if they need the data.

4.2 Threading Model

The DiffServ component does not create any new threads by itself during the execution of the PDK. The DiffServ implementation module runs in the same context with an external application, and the callback module runs in the same context with the FP Pplug-in receiving thread.

The external application's callback runs in another callback thread to avoid malicious usage.

4.3 Dependencies

In addition to the dependency on the Iinterface Mmanager and the FP Pplug-in component, the DiffServ Ccomponent must also make use of following other components:

Table 9. Diffserv components dependencies table

Components	Description
Namespace	The external and other internal components use the Namespace component to locate the active FE(s)
Central Callback	A central PDK callback service is provided to all the internal components to register, deregister, and retrieve the callback functions. The external user programs should not use, but should go through the existing NPF APIs for registering/deregistering.
DList (Double Link List)	The DList is the double link list designed in C. It provides multiple thread safety operations to manipulate the list.
PIL (Platform Independence Layer)	PIL helps in reusing software components for different applications. Writing the modules in a high-level language is not sufficient for easy re-use. Differences in the operating system on the platform and the compilers can make porting difficult. The PDK uses PIL to minimize compiler exceptions and provide an operating system independent API. In this way, the porting effort can be greatly reduced. Refer Appendix A for a complete description of the PIL.

Part 5: Appendix A

Appendix A

The following figure shows the logical modules and the data structures used in the IPv4-DiffServ design.

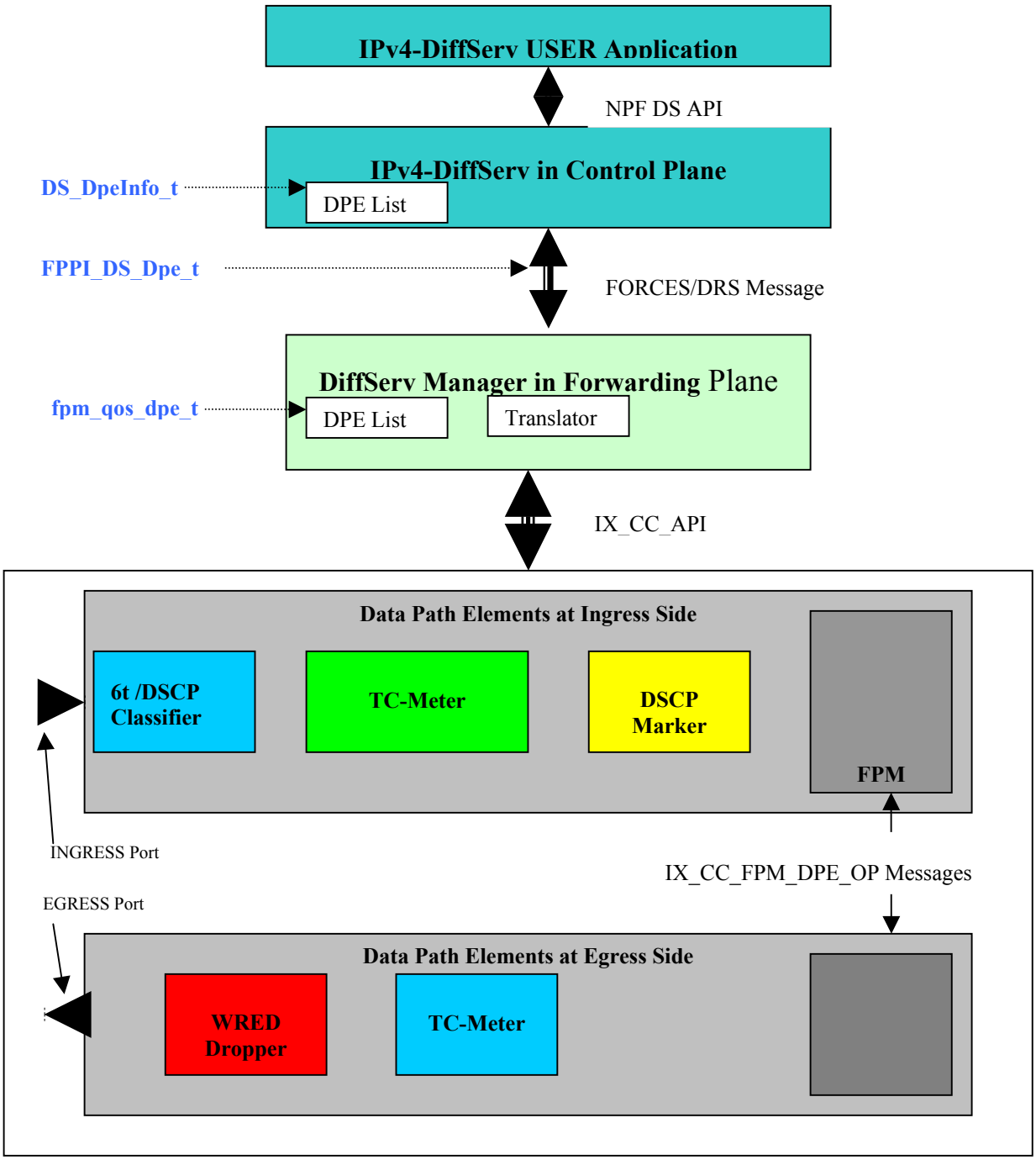

Figure 20: Logical modules and data structures in the IPv4 DiffServ design