RZ 3477 (# 93449) 02/17/03
Computer Science 29 pages

Research Report

# WideBridge: Adaptation of Programming Models from the Linux Kernel to the PowerNP

Mohit Gupta and Robert Haas

IBM Research
Zurich Research Laboratory
8803 Rüschlikon
Switzerland

# WideBridge: Adaptation of Programming Models from the Linux Kernel to the PowerNP

Mohit Gupta and Robert Haas

*IBM Research, Zurich Research Laboratory, 8803 Rüschlikon, Switzerland*

## Abstract

We developed a prototype called WideBridge that dynamically transforms Linux TC (traffic control) commands into PowerNP APIs calls using the kernel netlink interface. In this report, we first describe the Linux TC and PowerNP models used as well as the generic model used by our function. Then, we describe how WideBridge was implemented and tested for typical DiffServ edge and core routers scenarios. WideBridge makes it particularly easy for Linux developers to start using the PowerNP. WideBridge will also serve in the ForCES context (IETF effort to decouple forwarding elements from control elements), as the means to perform the equivalent of ”impedance matching” between a control point and a network processor (NP).

# Table of Contents

# Chapter 1: Introduction

The need for quality of service (QoS) in the Internet for applications such as IP telephony and other multimedia applications is increasing considerably. The currently used best-effort service is not able to satisfy this need because it cannot offer QoS capabilities. Differentiated Services (DiffServ) were proposed in an effort to fill this gap.

A multitude of programming models for networking components such as DiffServ is likely to coexist in the future. To address this, we introduce a new function that serves as a bridge between the models used by application-level programmers to those used for network processors. To develop such a function, a deep understanding is required of each model's capabilities, limitations, and, most importantly, their common properties, before the widest bridge between these models can be built.

We developed a prototype called WideBridge that dynamically transforms Linux TC (traffic control) commands into PowerNP APIs calls using the kernel netlink interface. In this report, we first describe the Linux TC and PowerNP models used as well as the generic model used by our function. Then, we describe how WideBridge was implemented and tested for typical DiffServ edge and core routers scenarios. WideBridge makes it particularly easy for Linux developers to start using the PowerNP. WideBridge will also serve in the ForCES context (IETF effort to decouple forwarding elements from control elements), as the means to perform the equivalent of "impedance matching" between a control point and a network processor (NP).

# Chapter 2: Differentiated services architecture

The differentiated services architecture lays the foundation for implementing service differentiation in the Internet in an efficient and scalable way. The IETF DiffServ Working Group charter states:

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

The IPv4 packets' Type-of-Service header field byte is split into two parts. Two bits (bits 7-6) are currently not used and can be ignored (set to zero). The rest (bits 5-0) are used as the differentiated service codepoints (DSCP). The codepoints are divided into three pools. One is for standards and the other two are for experimental or local use. It is possible that one of these two pools for experimental or local use will be quoted for standardization, too.

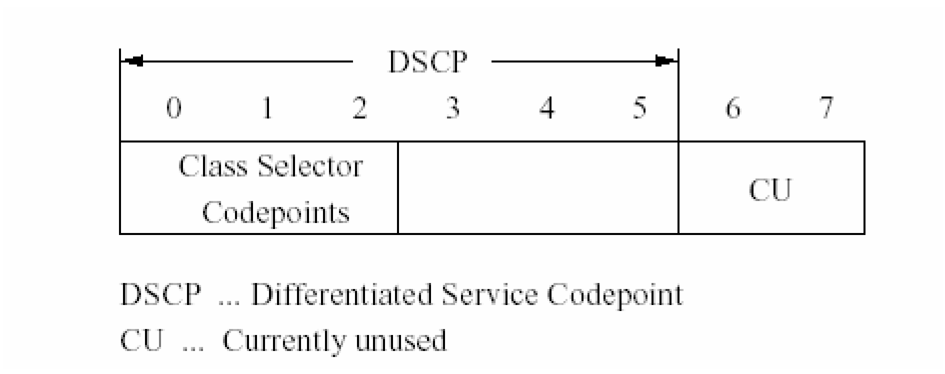
DSCP
0
1
2
3
4
5
6
7
Class Selector Codepoints
CU
DSCP ... Differentiated Service Codepoint
CU ... Currently unused

**Figure 1: Class selector codepoint field.**

## Best-effort service

*Best-effort* is the currently used service in the Internet. This service has the code point `000000` – Default codepoint, default PHB.

## Expedited forwarding service

The recommended codepoint for EF PHB is `101110`. The *expedited forwarding service* has the following properties:

- Peak bit rate (of flows or aggregated flows) guarantee.

- No bursts (only within the peak bit rate).
- Low queuing delay (for real-time applications).

### Assured forwarding service

*Assured forwarding* (AF) *service* dedicates a certain amount of bandwidth to the customers. Four AF service classes are defined. Each of these classes has three levels of dropping precedence (low, medium and high).

**Table 1: Assured forwarding codepoints**

| *Dropping Precedence* | *Class 1* | *Class 2* | *Class 3* | *Class 4* |
|---|---|---|---|---|
| Low | 001010 | 010010 | 011010 | 100010 |
| Medium | 001100 | 010100 | 011100 | 100100 |
| High | 001110 | 010110 | 011110 | 100110 |

**Table 2: IXIA traffic generator TOS usage**

**NOTE**: The IXIA traffic generator uses the TOS bits instead of the DSCP bits to mark packets. The DSCP value should be shifted to the left by 2 bits for use with the IXIA, e.g. 0x22 becomes 0x88.

## 2.1 Traffic control in the Linux kernel

The two main elements of the DiffServ conceptual model are *traffic classification* and *traffic conditioning* as shown in Figure 2.

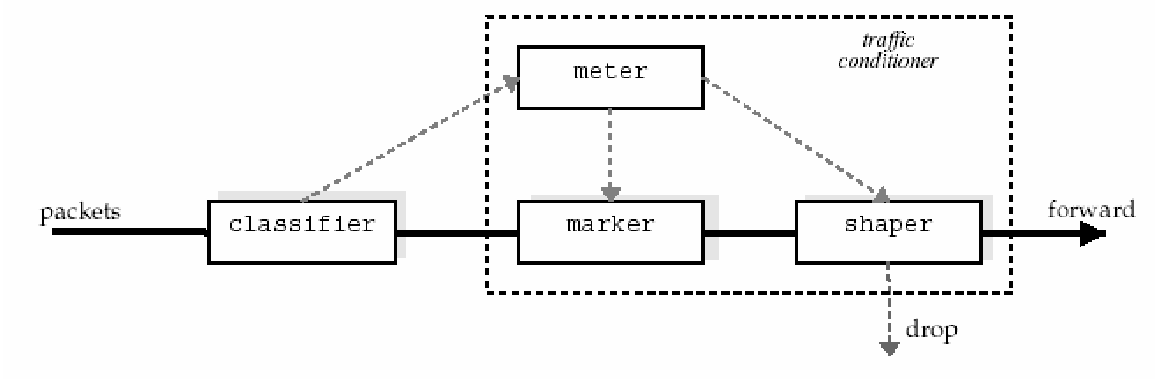

**Figure 2: Logical view of a traffic classifier and conditioner.**

When packets arrive at the ingress interface of a DiffServ router they are typically classified and actions are performed. Then the packets are forwarded to the next hop.

Linux supports for many advanced networking features, such as firewalls, QoS etc., in the form of queues, classes and filters, traffic conditioning, etc.

Network traffic that is received by a host from the network is either destined to that host or should be sent on to the next hop in case it acts as a router (or is discarded if neither of these cases are applicable). If packets are forwarded, which is often a kernel-level process, numerous actions have to be carried out: selection of the output interface, selection of the next hop, encapsulation, etc. When this has been done, packets are queued at the respective output interface. This is the point where Linux network TC comes into play.

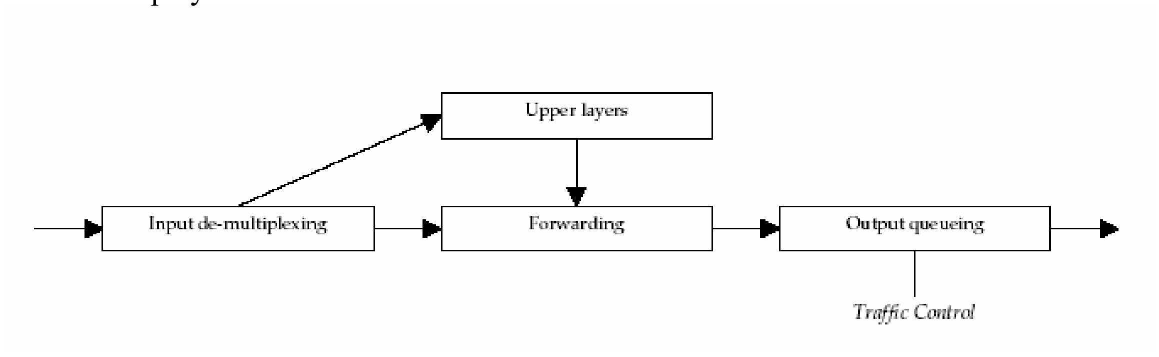

**Figure 3: Processing of network traffic in Linux.**

The important features of the TC model are :

- **Qdisc defines general semantics**
- **Classes implement different behavior**
- **Packets are attributed to classes by filters**
- **Classes in turn may contain queuing disciplines.**

## 2.1.1 **Note about queuing disciplines**

In a system with one or more network interface cards, each of these devices has a queuing discipline (qdisc) attached to it. A qdisc determines in what manner packets enqueued for that device are treated.

Linux supports a hierarchy of qdiscs for any interface. Such sophisticated qdiscs use a filter to assign a class, which in turn determines whether to forward the packet as fast as the interface permits or to enforce a specific maximum traffic rate, depending on the originating IP address of the packet, hence possibly giving priority to one packet over another.

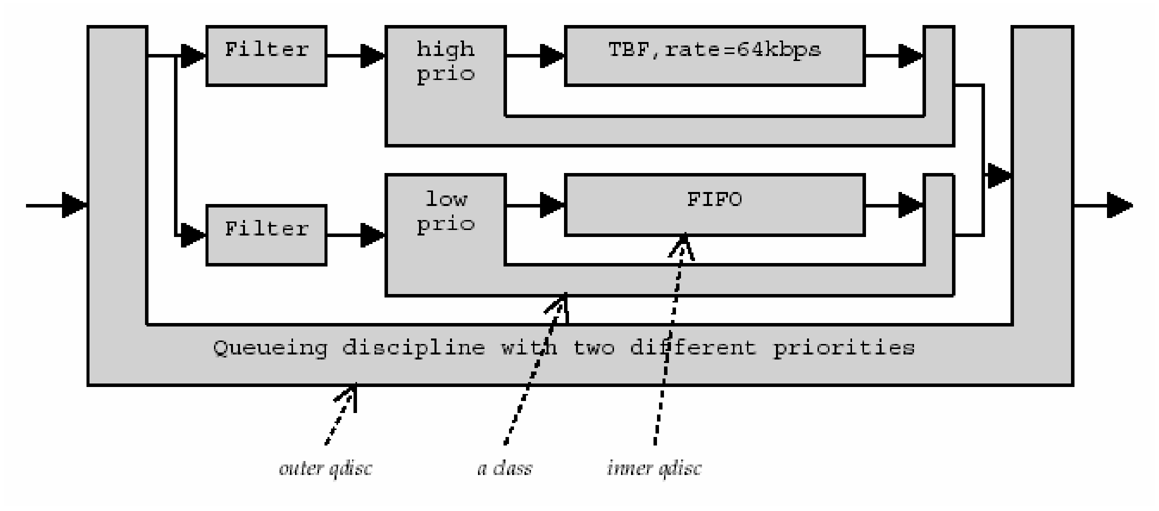

**Figure 4: Hierarchy of queuing disciplines with filters.**

Figure 4 shows a root qdisc with filters and classes. The classes in turn have inner qdiscs associated with them. Typically one qdisc is associated with a class. If no qdisc is specified, FIFO is taken as the default.
Whenever a packet is enqueued at this interface, the root qdisc matches it against the filters and assigns a CLASSID. Based on the value of the CLASSID, the packet is enqueued to the class or more specifically to the qdisc associated with it.

Figure 5 outlines the relationship between the elements of the DiffServ architecture and those in the Linux network TC. The three main functions (classification, metering and queuing/scheduling) are performed by different elements in the two architectures, as is highlighted by the gray boxes. An imminent conclusion is that the DiffServ architecture has not been designed specifically for Linux, nor has the Network Traffic Control code been tailored for DiffServ.

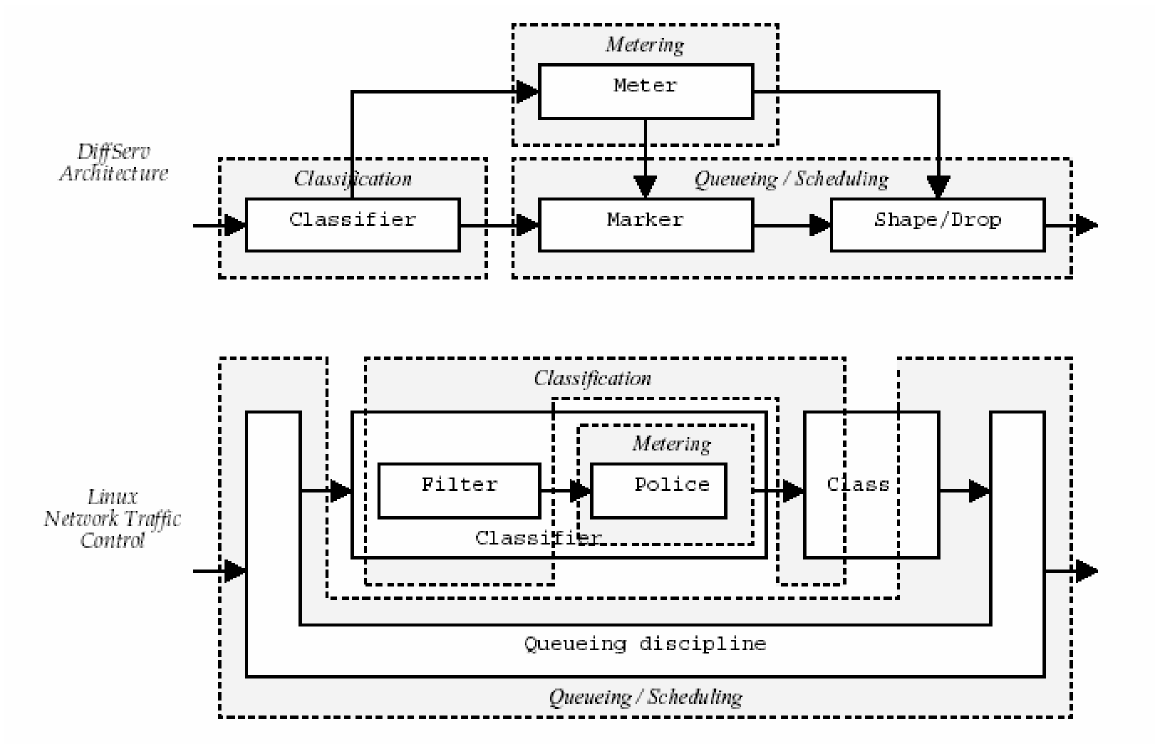

**Figure 5: DiffServ versus TC architecture.**

There is one TC command for each entity in the TC model. Furthermore, there is a one-to-one mapping between a TC command and a netlink message. An important thing to note here is that the DiffServ blocks overlap one or more TC blocks, implying that the complete information about the DiffServ block is scattered across the same or different kinds TC blocks.

## 2.1.2 Note about policing in Linux

In the Linux kernel, metering is done at the ingress. The concepts of special color-aware or color-blind and single or double-rate policer do not exist. Instead, a different model of single-rate meters, associated with filters, exists that meters the traffic at the given rate and burst size. But a combination of these meters could act as the standard policers defined by IETF RFC 2697 and RFC 2698.

The following is an excerpt from a TC script for a two-rate, color-blind policer:

```
TC filter add dev reth1 parent 1:0 protocol ip prio 1 u32 \
match ip src 10.20.3.12/32 \
police rate 1000kbit burst 90k \
continue flowid :1
TC filter add dev reth1 parent 1:0 protocol ip prio 3 u32 \
match ip src 10.20.3.12/32 \
police rate 2000kbit burst 90k \
continue flowid :2
```

```
TC filter add dev reth1 parent 1:0 protocol ip prio 5 u32 \
match ip src 10.20.3.12/32 \
continue flowid :3
```

All traffic coming on the interface reth1 from ip source is 10.20.3.12 policed and assigned a DSCP color value based on the policing result. The filters are matched in sequence based on their defined priority. If the traffic matches the profile of the meter, it is assigned the corresponding flowid that is later converted into the DCSP value at the egress. Otherwise the "continue" action to match the next filter is executed.

## 2.2 TC vs. NP model – limitations and capabilities

We do not describe the intrinsic details of the NP model but only cite its capabilities and limitations compared to the TC model.

1. The TC model is modular to a greater extent than the NP model. That is to say that the building blocks in the TC model constitute of small part of DiffServ blocks. It provides a lot more flexibility in terms of possible DiffServ scenarios.
2. The TC model is versatile in terms of dynamic changes in configuration. Owing to its modularity, changes at a very small scale are possible. The building blocks in the case of NP are larger and thus mapping small changes in the TC model might be fatal. For example, it is possible to change a single-rate policer to a two-rate policer in Linux by using a single command, whereas on NP it would require first deleting the original single-rate policer and replace it with a two-rate one.
3. TC supports a hierarchy of qdiscs at its interfaces. Individual queues at the egress in the TC model might be scheduled differently. The packet handling at an egress might involve processing in a complex structure of qdiscs, which is impossible to map on the NP model.
4. The number of scheduling algorithms in the Linux kernel is far greater than those on the NP. But NP supports BAT for the flow control, which has no equivalent on Linux.
5. The TC model provides flexibility in terms of policing the incoming traffic at the ingress. The NP on the other hand supports only the standard policers. The heart of Linux policing is the meter element that can meter the traffic for a given rate and a burst. Other complex policers can be built using this meter element.
6. The NP model supports scheduling at the ingress before the packet is enqueued at the switch. NP could have WRED or SARED as the ingress flow control algorithm. This facility is completely missing in the TC model.
7. The NP supports flow control of packets before they are enqueued at the target port. More explicitly it allows flow control of the traffic on a flow basis rather than a port basis. In Linux everything is on an interface basis.
8. The netlink messages are based on the TC model. Hence mapping them to an entirely different NP model involves a certain amount of adaptation. An attempt has been made to map the intersecting region between the two models with respect to the DiffServ capabilities onto the NP.

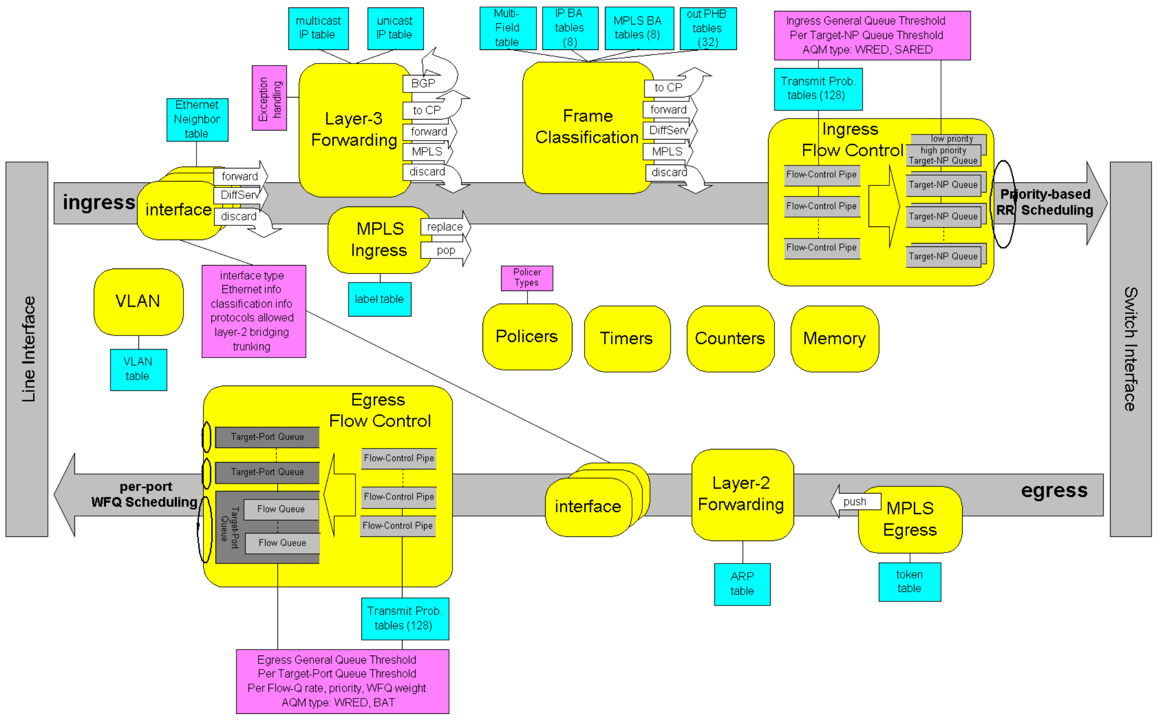

**Figure 6: Datapath diagram of the PowerNP.**

# Chapter 3: Netlink

ForCES aims to define a framework and associated mechanisms for standardizing the exchange of information between the logically separate functionality of the control plane, including entities such as routing protocols, admission control, and signaling, as well as the forwarding plane, where per-packet activities such as packet forwarding, queuing, and header editing occur. By defining a set of standard mechanisms for control and forwarding separation, ForCES will enable rapid innovation in both the control and forwarding planes. A standard separation mechanism allows the control and forwarding planes to innovate in parallel while maintaining interoperability.

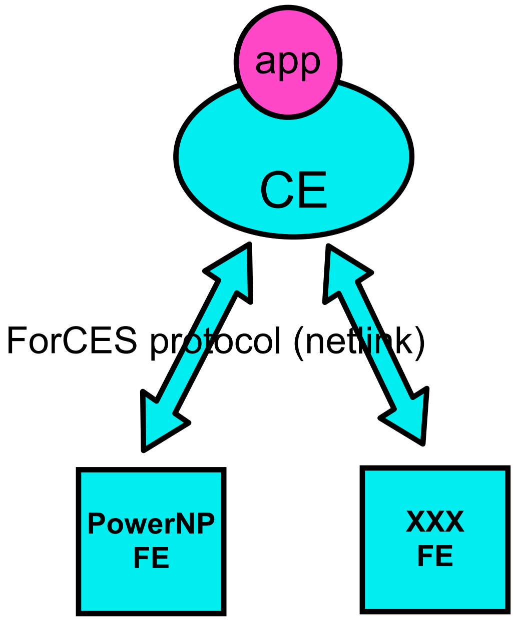

**Figure 7: ForCES protocol from the CE to FE.**

The proposal is to reuse the framework of the Linux netlink messaging protocol and extend it to ForCES. The motivation behind adopting netlink as the framework for ForCES is to enable users to use Linux TC API to configure QoS parameters on the NP. Currently, Linux TC uses netlink messages to configure QoS parameters in the kernel. Netlink is the messaging protocol between the user space and the kernel space for setting up route entries, filtering rules, QoS parameters, etc.

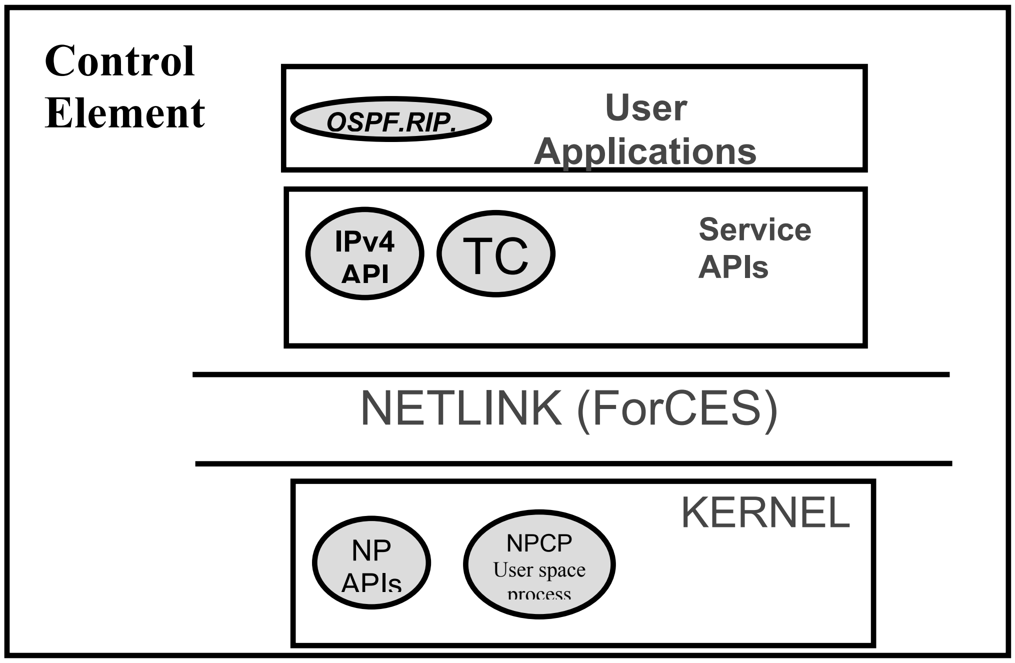

**Figure 8: Control element architecture with netlink.**

TC uses Netlink messages to configure filtering rules and QoS parameters in the Linux kernel. There is a one-to-one mapping between the TC command and the netlink messages. The QoS features are configured and controlled using a netlink socket interface. Routing messages, called rtnetlink, are special netlink messages to control the routing behavior of Linux. The bidirectional communication link consists of a standard socket-based interface for user processes and an internal application program interface (API) for the kernel. Netlink sockets are raw, but it is a datagram-oriented service.

The pseudo-code for the communication over the netlink socket of the TC commands is as follows:

- User process
    - parse TC command
    - construct configuration message
    - open netlink socket
    - send message over netlink socket (with RTM_GROUP=0, meaning that it is only for the kernel module which is listening for the group).
- Kernel module
    - receive and parse message
    - find necessary information in internal data
    - check for existing configuration
    - add default fields to the message
    - send message over netlink socket (with RTM_GROUP=8, the groups ID for traffic controller messages.

The various kernel modules process the netlink messages as described above. The processing has been verified to hold true for all cases but one. The module for the tcindex

**Table 3: Information about bug in the TC kernel module**

```
Linux Kernel Bug: There is kernel bug in the processing of the tcindex
        messages by the linux kernel module. Here is an excerpt from the
        examples in iproute2 DiffServ distribution.

$TC qdisc add $EGDEV handle 2:0 root dsmark indices 64
#
# The class mapping
#

$TC filter add $EGDEV parent 2:0 protocol ip prio 1 \
          handle 1 tcindex classid 2:1
$TC filter add $EGDEV parent 2:0 protocol ip prio 1 \
          handle 2 tcindex  classid 2:2
$TC filter add $EGDEV parent 2:0 protocol ip prio 1 \
          handle 3 tcindex  classid 2:3
$TC filter add $EGDEV parent 2:0 protocol ip prio 1 \
          handle 4 tcindex  classid 2:4

When parsing the tcindex filter messages the handle and the classid
information is found to be missing. Only the default values that are set by
the kernel for mask, shift and fall_through could be seen.
         >> hash 64 mask 0xffff shift 0 fall_through

But >> tc filter ls dev eth0 parent 2:0, returns all the handle - classid
pair I have instantiated. There is something wrong in the way the tcindex
filter messages are being processed by the kernel. The handle and classid
information is no longer there in the message when it is put back on the
netlink socket by the kernel module.
```

DiffServ-specific traffic control messages (RTM_QDISC, RTM_TCLASS and RTM_TFILTER) contains a tcmsg after the initial header, structured as follows:

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

**Table 4: Netlink sniffer tool**

| |
|---|
| **JrSniffer:** JrSniffer is a user space sniffer that opens a netlink socket to listen to all kinds of messages. It prints out the details associated with the traffic control messages. Given the scarcity of documentation about netlink and rtnetlink, it is a handy tool to know all about the information carried by the netlink messages.<br>To use the JrSniffer set the parameter netLinkSockaddr.nl_groups = RTMGRP_LINK \| RTMGRP_IPV4_ROUTE \| RTMGRP_IPV4_IFADDR \| RTMGRP_NEIGH \| RTMGRP_TC \| RTMGRP_NOTIFY in the source file to listen to specific kinds of messages. The message group RTMGRP_TC is for the DiffServ-specific traffic control messages. The JrSniffer prints all the details carried by the traffic control messages RTM_QDISC, RTM_TCLASS and RTM_TFILTER. |

# Chapter 4: WideBridge

WideBridge aims to bridge the gap between the TC and the PowerNP models. It provides the framework to enable the use of TC API on Linux to configure QoS parameters on the NP. But it does not restrict itself to the NP and TC models. It merely translates the netlink messages into an intermediate representation based on the DiffServ model and then converts them into API calls for the NP. Hence it is easily portable to other systems, both on the TC as well as the NP side.

## 4.1 Implementation architecture

The ForCES protocol aims to standardize the exchange of information between the control element (CE) and the forwarding element (FE). In the ideal case the native ForCES protocol should be running between the CE such as a Linux-based machine and the FE such as the PowerNP. The design architecture is shown in Figure 9 below.

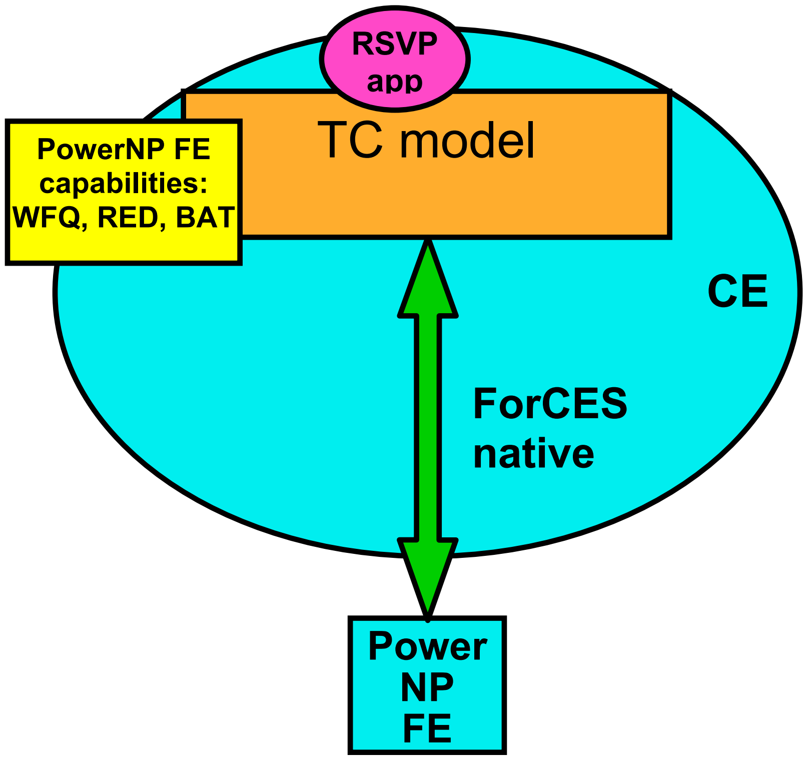

**Figure 9: Ideal case ForCES architecture.**

But the implementation in the real case is different. See Figure 10 for details. In this design the ForCES protocol runs within the CE between the user space and the kernel space instead of between the CE and the FE. The ForCES messages are interpreted by a process that maps them into NP API calls. Thus the flow of information between the CE and FE is in the form of the guided cells. The rationale behind such a model is that the NP can only interpret guided cells and not ForCES messages. Another advantage of this design is that configurations that are not supported by the FE can be identified in the CE itself, which reduces the number of queries between the CE and the FE.

Our design based on this architecture is still portable to the ideal case. Once the NP is capable to interpret ForCES messages, the kernel process for convert these messages into API calls and in turn into guided cells can be transferred to run on the NP.

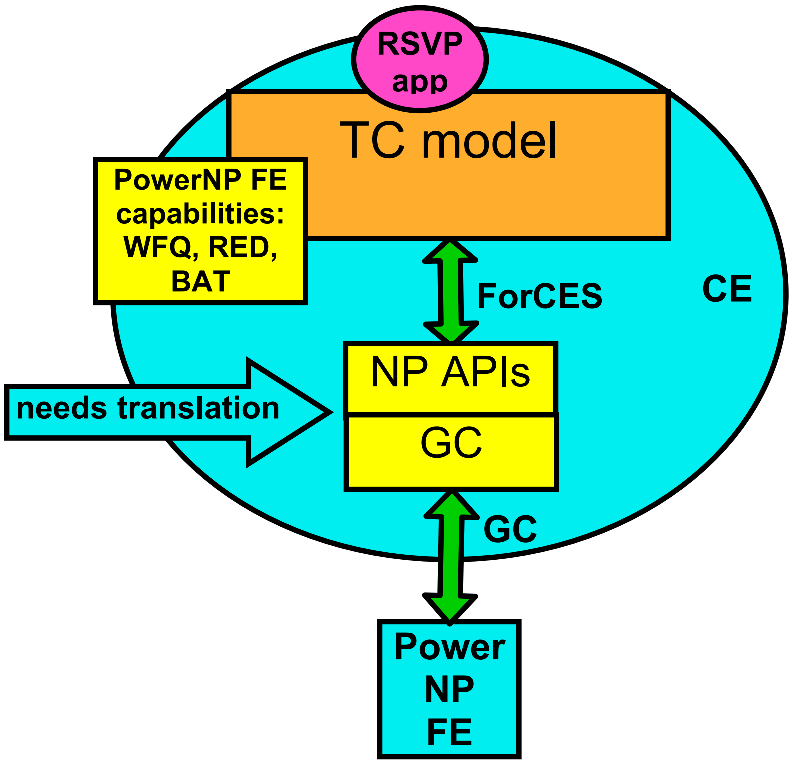

**Figure 10: Real case ForCES architecture.**

## 4.2 Evolution of the Design

Having enough knowledge about QoS, TC, netlink and the NP model, let us now examine the development of the WideBridge project. The design consists of two essential components.

1. **Translations** – This is the process of mapping the TC-based API calls into the NP API calls. Ideally the mapping should be done in real time. Each netlink message should be mapped into an NP API call. But because of the distributed nature of information about the API calls across many netlink messages, the logical solution is to buffer the messages and then make the NP API call.
2. **Adaptation** – As stated above, the DiffServ capabilities of the NP and TC model are different. Only the intersecting subset can be mapped perfectly. But for other cases there is some degree of adaptation involved, i.e. the mapping of one model

to another on a best-effort basis. Even after this adaptation there is a set of features that are supported on the TC model and not on the NP and vice versa.

### 4.2.1 DESIGN ONE: TC-parameterized target models

Initial design focused on the TC model as the core. An attempt was made to parameterize the TC model to express its DiffServ capabilities. Other models then simply describe their capabilities in terms of these parameters. Any application that uses the TC model to configure features on some NP would have to declare upfront its requirements based on these TC model parameters.

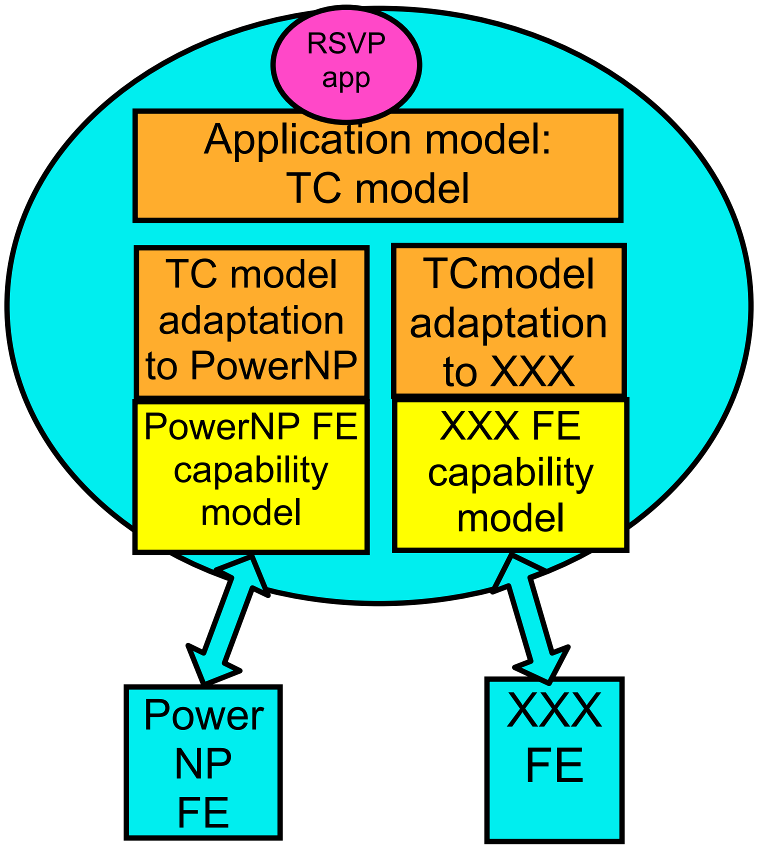
RSVP app
Application model: TC model
TC model adaptation to PowerNP
TCmodel adaptation to XXX
PowerNP FE capability model
XXX FE capability model
Power NP FE
XXX FE

**Figure 11: Parameterized TC model design.**

#### 4.2.1.1 Advantages

The advantage of this model are that once the requirements have been declared by the user application, they can be matched against the capabilities of the NP, making it possible to decide whether the configuration is acceptable. It would avoid runtime errors arising due to incompatible models.

#### 4.2.1.2 Disadvantages

The major drawback of this approach is that the DiffServ capabilities are restricted to the TC model. Any other NP model is just a subset of the TC model. The TC model is adapted as the standard for the implementation of QoS. Second, applications need to know the parameterization of the TC model and how to express their capabilities based on the TC model. This is not always possible to decide upfront. Even all this does not rule out the possibility of run-time errors.

### 4.2.2 DESIGN TWO: Real-time netlink to NP

The second design is based on converting netlink messages into NP API calls on the fly. The idea behind this design is to demonstrate that QoS features could be configured on the NP by TC API calls. The design is very simplistic because the underlying logic to make the NP API calls assumed a certain set of netlink messages for certain API calls. Below is the pseudocode for the design logic:

```
o  While(1){
       o  Listen to the netlink messages
       o  Interpret the kind (RTM_NEWTCLASS, RTM_NEWTQDISC,
          RTM_NEWTFILTER, etc)
       o  Extract relevant information
       o  Store information in usable format
       o  If  ( information == complete)
              ▪  Make NP API call
o  }
```

Here is a TC configuration script example mapped into NP API calls. The information in ***red*** is about the interface and the semantic action. The information in ***blue*** is the policer information and the information in ***green*** is the classifier information.

**Table 5: Example : TC to NP API calls (relevant information in color).**

```
▪  TC qdisc add dev $INDEV handle ffff: ingress

▪  TC filter add dev $INDEV parent ffff: protocol ip prio 1 u32 match ip src 10.2.0.0/24
$meter1 continue flowid :1
▪  >TC filter add dev $INDEV parent ffff: protocol ip prio 3 u32 match ip src 10.2.0.0/24
$meter2 continue flowid :2
▪  >TC filter add dev $INDEV parent ffff: protocol ip prio 5 u32 match ip src 10.2.0.0/24
$meter3 drop flowid :3
▪  >TC filter add dev $INDEV parent ffff: protocol ip prio 7 u32 match ip src 0/0 $meter5
drop flowid :4
▪  >TC qdisc add $EGDEV handle 1:0 root dsmark indices 64
▪  >TC class change $EGDEV classid 1:1 dsmark mask 0x3 value 0x88
▪  >TC class change $EGDEV classid 1:2 dsmark mask 0x3 value 0x90
▪  >TC class change $EGDEV classid 1:3 dsmark mask 0x3 value 0x98
▪  >TC class change $EGDEV classid 1:4 dsmark mask 0x3 value 0x0
▪>TC filter add $EGDEV parent 1:0 protocol ip prio 1 handle 1 tcindex classid 1:1
▪>TC filter add $EGDEV parent 1:0 protocol ip prio 1 handle 2 tcindex  classid 1:2
▪>TC filter add $EGDEV parent 1:0 protocol ip prio 1 handle 3 tcindex  classid 1:3
▪>TC filter add $EGDEV parent 1:0 protocol ip prio 1 handle 4 tcindex  classid 1:4
```

#### 4.2.2.1 Advantage

This prototype is just to demonstrate the various configurations on the NP with user space TC API calls. This system maps the netlink messages to NP API calls in real time. It exposes the drawback of a real-time system to serve the purpose of mapping and lays the foundation of the next design phase.

#### 4.2.2.2 Disadvantages

The disadvantages with this approach are:

- Not generic — limited to only a few DiffServ scenarios
- Fails to map TC model to the NP — maps only a small subset
- Fails if the sequence of messages is changed
- Static configuration of NP.

## 4.2.3 DESIGN THREE: Parser-based netlink to NP mapping

The next proposed design is not a real-time mapping between models. It is based on the language parsing techniques to parse the netlink messages and make the API calls. The parser was based on the grammar of the NP. The grammar defined the capabilities of the NP in terms of the netlink messages. The basis structure of the design is as follows:

- Buffer all netlink messages before making an API call
- Store in a tree structure
- Walk through the tree to find interrelationships and dependencies
- "Reorder" the tree if necessary
- Input netlink messages as tokens to PARSER
- Grammar rules of the PARSER corresponds to API calls to NP PARSER - FSM with final state >> API call.
-

### 4.2.3.1 Advantages

The highlight of this design is that it attempts to map the NP model to the TC model rather than the other way round. This design preserves all the original netlink messages in a tree structure. So the information available to map the configuration into the NP API calls is complete when the first API call is made. The advantage is the accuracy of mapping the configuration onto the NP. The design was versatile because of its ability to accommodate new grammar rules at any time to expand NP to TC mapping. This design served as the framework for the final design of WideBridge.

### 4.2.3.2 Disadvantages

The major drawback of this design is the complexity of the parser. As the intermediate buffering is done in a tree structure of netlink messages, the accurate maintenance and parsing of such a structure is difficult. Second, the parser is specific to the FE, so for every FE a new parser has to be hard-coded.

# Chapter 5: WideBridge – XML-based TC to NP mapping

Based on the parser approach, this design replaces the netlink tree structure with an XML-based intermediate representation (IR). The DiffServ model described in the RFC 3289 has been selected as the schema for the intermediate representation. The DiffServ-based intermediate representation of the QoS configuration is generic, standard and extensible. The first step of the design is to map netlink messages into IR and the second step is to convert the IR to NP API calls. The complexity in this case has been distributed over two separate processes. The design is described in detail in the following sections.

## 5.1 Architecture

The architecture of the design is centered around the XML-based intermediate representation. The implementation architecture of the design is shown in Figure12.

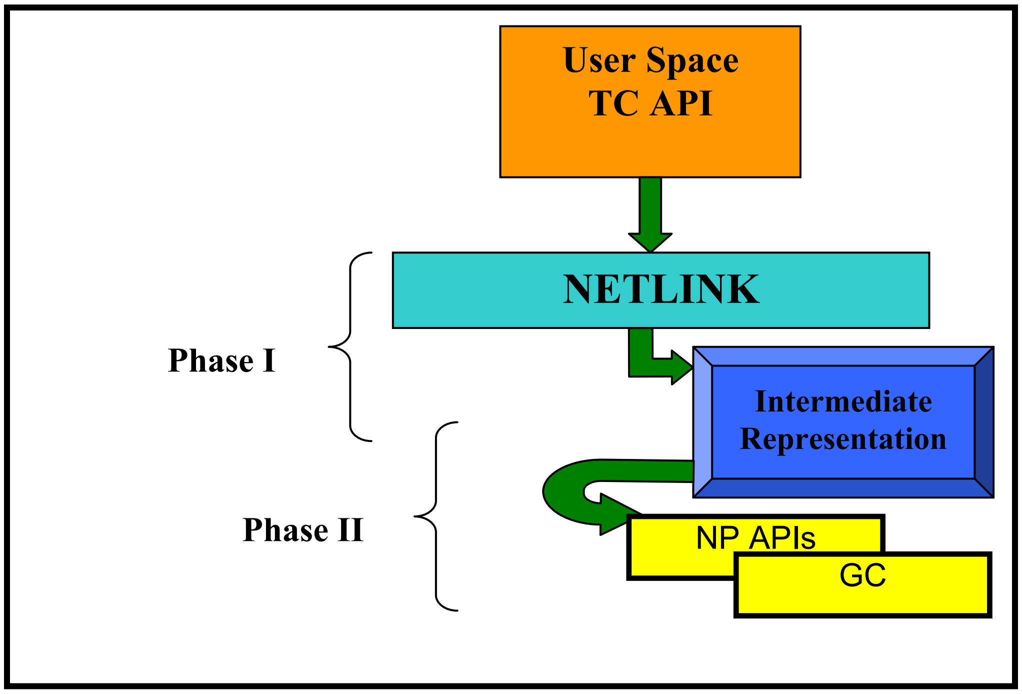

**Figure 12: Design overview.**

There are two phases of the mapping from the netlink messages to the NP API calls. The first phase converts the netlink messages into the XML-based intermediate representation. The process opens a netlink socket, parses all traffic control messages over the socket and writes the IR configuration file.

Ideally the Linux kernel should mirror the configuration on the NP, so a netlink message should only reach the kernel after an equivalent NP API call is successful. To achieve this a two-level adaptation is involved.

- TC sends netlink messages to the kernel module for processing with the NL_GROUP field set as 0, which in turn, after processing, put the message back on the socket with NL_GROUP = 8 (8 for traffic control messages). Hack the kernel module and intercept the message before they are processed. For messages that are not supported by the NP, send an error to the user and discard the message. This way the netlink message that does not contribute to the configuration on the NP is discarded before the kernel module processes it. This feature has not been incorporated in the present design.
- There are some messages that are supported by the NP but not all possible combinations of them are necessarily supported. The kernel module then processes these messages. Once processed, the messages are put back on the netlink socket with the correct NL_GROUP. The phase I process to convert netlink message to IR reads this message from the socket and extracts the relevant information to be written in the XML file. Once the configuration is complete, the phase II process tries to map the IR XML file into NP API calls. When an API call fails, this process immediately deletes the configuration from the kernel, hence maintaining perfect mirroring.

## 5.2 Implementation details

As stated earlier, the intermediate representation is the highlight of this design. The framework of the document-type definition of the IR is shown below.

**Table 6: Framework XML DTD of the intermediate representation.**

```
<?xml encoding="ISO-8859-1"?>
 <!ELEMENT IR      (Classifier)+>
 <!ELEMENT Classifier (Filter, Action)>
 <!ELEMENT Filter  (Ingress?, Egress?, SrcAdd?, DestAdd?, TOS?)>
 <!ELEMENT Action  (Policer | Qdisc | Marker | Dropper)+>
 <!ELEMENT Policer (PRate, PBurst, Marker)>
 <!ELEMENT Qdisc   (Queue, Scheduler?)>
 <!ELEMENT Marker  (Maction | Dropper)>
 <!ELEMENT Queue   (CBQ|PFIFO|CSZ|FIFO|TBF|RED|SFQ|GRED)>
 <!ELEMENT CBQ     (MinRate, MaxRate, MaxBurst)>
 <!ATTLIST FIFO    theshold CDATA #REQUIRED>
 <!ELEMENT Scheduler (WFQ|WRR)>
 <!ELEMENT WFQ     (Weight)>
 <!ELEMENT WRR     (Weight)>
```

- One XML file can have one or more classifier elements
- Each classifier element contains a filter and an action
- A filter element contains one ingress, egress, source address, destination address and DSCP
- An action element can be to police, flow control, mark DSCP or drop
- Policing is done with one or more meters with a specified rate and a burst
- Flow control is done using a qdisc and scheduler. Here the IR defines the scheduler for individual qdiscs, which is possible to have on TC but not NP
- Qdisc can be of a number types such as CBQ, PFIFO, etc. But currently only a mapping from a CBQ to a egress queue on NP has been successfully implemented
- Scheduler is represented by weighted round robin or a weighted fair queuing type and a weight associated with it
- For a complete DTD or a schema, refer to the source code of WideBridge

Phase I – Netlink to Intermediate Representation

This program creates from scratch a new DOM document in memory. The schema to verify of the document is described above. It opens a netlink socket to listen to traffic controller messages of the type RTMGRP_TC. Each netlink message is parsed for its useful information contents, which are stored as values in nodes attached to a DOM tree. After the first netlink message is received, the process starts a timer. We assume that one particular configuration would be complete in a specified period of time. Once the timer expires the process serializes the DOM tree, writes the XML-based IR file and initiates the Phase II process to configure the NP. The value of the timer has been set to 60 seconds but could be changed depending on the speed of the CPU. The timer has not been added in the present prototype.

The process parses the netlink traffic controller messages of type RTM_NEWTCLASS, RTM_NEWQDISC and RTM_NEWTFILTER for information about the TC class, qdisc and filter, respectively. The following example illustrates the working of this first phase script. The information in **red** from the TC messages is utilized to create the **green** nodes of the DOM tree.

-

**Table 7: Step-by-step generation of IR by TC API calls.**

| **TC command** | **Evolution of XML DOM tree** |
|---|---|
| TC qdisc add **dev reth3** handle ffff: ingress | <?xml version="1.0" encoding="UTF-8" standalone= "no" ?> <IR><Classifier><Filter> **<Ingress value="reth3"/>**</Filter> <Action></Action></Classifier></IR> |
| TC filter add dev reth3 parent ffff: protocol ip prio 1 u32 match ip **src10.20.3.12/32 police rate 1000 burst 90** continue **flowid :1** | <?xml version="1.0" encoding="UTF-8" standalone="no" ?> <IR><Classifier><Filter><Ingress value="reth3"/> **<SrcAdd><Value value="0c03140a" /><Mask value = "ffffffff" /> </SrcAdd>**</Filter> <Action>**<Policer><PRate value="1000"/> <PBurst value="2483" /> <Marker value = "1"/> </Policer>** </Action> </Classifier> </IR> |
| TC qdisc add **dev reth1** handle 2:0 root dsmark indices 64 default_index 4 | <?xml version="1.0" encoding="UTF-8" standalone= "no" ?> <IR><Classifier><Filter><Ingress value="reth3"/><SrcAdd><Value value="0c03140a"/><Mask value="ffffffff"/></SrcAdd> **<Egress value="reth1"/>** </Filter><Action><Policer><PRate value="1000"/><PBurst value="2483"/><Marker value="1"/></Action></Classifier></IR> |
| TC filter add dev reth1 parent 2:0 protocol ip prio 1 **handle 1** tcindex **classid 2:1** | <?xml version="1.0" encoding="UTF-8" standalone="no" ?> <IR><Classifier><Filter><Ingress value="reth3"/> <SrcAdd><Value value="0c03140a"/><Mask value="ffffffff"/> </SrcAdd><Egress value="reth1"/> </Filter> <Action><Policer> <PRate value="1000"/> <PBurst value="2483"/> **<Marker value="2:1"/>** </Policer></Action></Classifier></IR> |
| TC class change dev reth1 **classid 2:1** dsmark **mask 0x3 value 0x88** | <?xml version="1.0" encoding="UTF-8" standalone="no" ?> <IR><Classifier><Filter><Ingress value="reth3"/> <SrcAdd><Value value="0c03140a"/><Mask value="ffffffff"/></SrcAdd><Egress value="reth1"/> </Filter><Action><Policer><PRate value="1000"/> <PBurst value="2483"/> **<Marker value="22"/>** </Policer></Action></Classifier></IR> |

-

The example above is an excerpt from the TC script to configure an edge route to police the incoming traffic and mark the DSCP. Once the timer expires, the DOM tree is serialized, IR configuration file is generated and the phase II process is initialized to configure the NP.

Phase II – Intermediate Representation to NP API calls

The phase II process parses the XML-based IR file either generated by the phase I process or generated by some other means and makes equivalent API calls to configure the NP.

This process is divided into two parts. One part extracts the filter information from the IR and utilizes it to form either the classifier rule or BA table entry depending upon the action field of the IR classifier entry. The following example illustrates the concept.

Here is an excerpt from an IR file generated in phase I to configure an edge router to police the incoming traffic and mark the DSCP:

-

**Table 8: Mapping IR to NP classifier API call.**

| Intermediate Representation | NP API Classifier call parameters |
|---|---|
| <Ingress value="reth3"/> | Cls_rule.ingress_cntx_mask<br>Cls_rule.ingress_cntx_value |
| <SrcAdd><br>    <Value value="0c03140a"/><br>    <Mask value="ffffffff"/><br></SrcAdd> | Cls_rule.sa_lower<br>Cls_rule.sa_upper |
|   <Egress value="reth1"/> | Cls_rule.egress_cntx_mask<br>Cls_rule.egress_cntx_value |
| <DstAdd><br>• NOT DEFINED<br></DstAdd> | Cls_rule.da_lower = 0x00000000<br>Cls_rule.da_upper = 0xffffffff |
| <TOS><br>• NOT DEFINED<br></TOS> | Cls_rule.tos_value = don’t care<br>• Cls_rule.tos_mask = don’t care |

The action information of the IR is mapped to one of the following default action types for the classifier rules:

- DIFFSERV_POLICE_ACTION – This action type is defined for policing the incoming traffic and marks them as green, yellow or red. The policer can be color-aware as well as color-blind and also single and two-rate. This configuration is typical to an edge route to police the incoming traffic and assigns color tokens to it based on the profile.

- DIFFSERV_REMARK_ACTION – This action type is defined for the DSCP-marked incoming traffic from some other DSCP domain. Owing to differences in the DSCPs used across the boundary of the networks, the DSCP value has to be rewritten to the locally recognized values. It uses the BA table with the incoming DSCP as the key to locate the new DSCP value.
- DIFFSERV_QDISC_ACTION – The configuration of a core router is different from that of an edge router. At the core router, traffic is assumed to be marked with a DSCP value and is thus only assigned an appropriate queue at the egress to yield the required behavior. The queues to give differentiated behavior to different traffic streams have to be configured at the egress. The IR has information about the sustainable and peak service rates, which are translated into NP flow queue API calls.
- DIFFSERV_FILTER_ACTION – This action type defines a filter to either permit or restrict packets. All packets that match the classifier rule having this action type set to restrict are dropped.

The action types could be extended beyond the four defined above to accommodate other types of requirements. This is a stand-alone process that connects to the wrapper remotely.

## 5.3 Software dependencies and requirements

The following are the requirements for compilation and running both phase I and phase II processes:

- Preferably kernel 2.4 or any other kernel with DiffServ enabled is required. Kernel 2.3 and above come with the various traffic controller kernel modules.
- Xerces C++ is required for creating and parsing XML DOM documents. The source code can be downloaded from http://xml.apache.org/xerces-c/
- Iproute2 package can be downloaded from http://snafu.freedom.org/linux2.2/iproute-notes.html
- The cabtools from the terp/tools/cabtool is required to write the DPT values to the CAB memory address.

### 5.4 Capabilities

The XML-based generic intermediate representation makes this model versatile. It can easily be extended to any other messaging protocol to conFigure QoS parameters. It offers ease of verification of the configuration by means of the intermediate XML file. Also the same interface to convert netlink messages to IR can be reused for any other process, which converts IR to its specific API calls. The capabilities of both processes are easily extensible because of the standard representation.

### 5.5 Limitations

The limitations of the design are as follows:

- The system does not map netlink messages to API calls in real time.
- Minor changes in the TC model might lead to major changes in the API calls to the NP.
- Hierarchy of qdisc is not supported or mapped equivalently.

### 5.6 Testing and results

Many of the TC router configuration scripts were tested on the reference platform with aso version 132. IXIA was used as the traffic generator in all these cases.

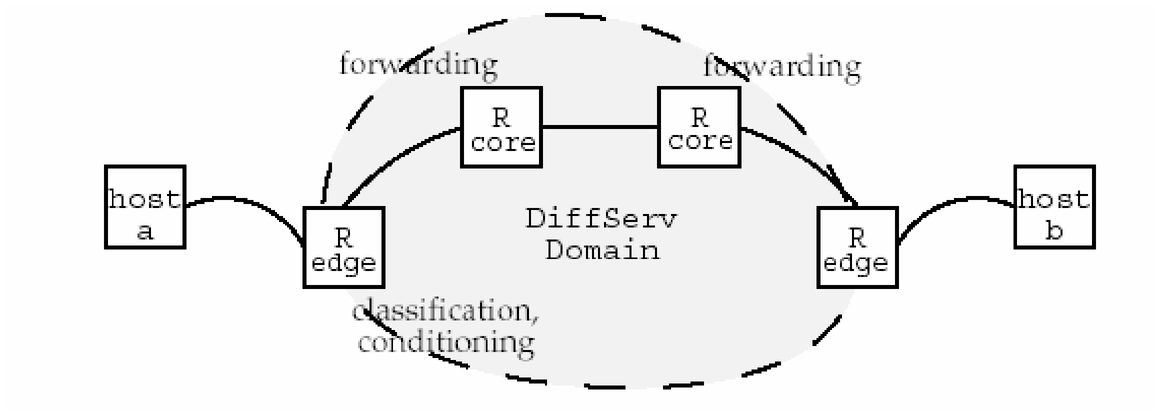

**Figure 13: Edge and core router.**

The different configurations tested are as follows:

- Edge router as a color blind policer to mark the DSCP,
- Edge router as a color aware policer to remark the DSCP,
- Edge router to remark the DSCP from other domain,
- Core router to give DiffServ behavior.

Linux uses the CBQ classes to give DiffServ behavior to the marked traffic. The qdisc attached to this class by default is FIFO, which does a tail drop after the queue length is exceeded. To imitate exactly the same behavior on the NP, flow queues with exactly the same parameters as the CBQ class is initialized at the egress. The tail drop FIFO is mapped by overwriting the drop probability table (DPT) of the NP using the cabtool with appropriate values.

# Chapter 6: WideBridge HOW-TO

Here are some of the problems faced during the development of WideBridge, as well as their solutions.

## 6.1 Problems

- In the case of kernel 2.4 there was a slight change in the method of ARP messaging. Out of the box aso (version 132 and 133) compilation do not work on kernel 2.4 until a device is enable to listen to ARP messages. This can be done by
    - Bash$ echo  1 > /proc/sys/net/ipv4/neigh/eth1/app_solicit .
- Use the debugged version of the cabtool present in the tools directory of the WideBridge source.
- As stated earlier there is a bug in the processing of the tcindex filter netlink messages. In order to read the original information from the tcindex messages the some changes in the TC source code are done. The nl_groups information in the /iproute2/lib/ll_rtnetlink.c rtnl_talk() function is changed to 8 for RTMGRP_TC. Now we receive two netlink messages, one original and the other processed by the kernel module. One of the two messages is used to extract the relevant information.
- There was a problem with the way the pico-code processed the packets in the NP.

## 6.2 Kernel-2.4-HOW-TO

Support for DiffServ is already integrated into 2.4 kernels. In order to enable it, you may have to reconfigure and rebuild your kernel, or at least some modules.
The following kernel configuration options have to be enabled in the section **Networking options**:

- Kernel/User netlink socket (`CONFIG_NETLINK`)
- Network packet filtering (`CONFIG_NETFILTER`)
- QoS and/or fair queuing (`CONFIG_NET_SCHED`)

The following kernel configuration options should be enabled in the section **Networking options**, **QoS and/or fair queuing**:

- CBQ packet scheduler (`CONFIG_NET_SCH_CBQ`)
- The simplest PRIO pseudo-scheduler (`CONFIG_NET_SCH_PRIO`)
- RED queue (`CONFIG_NET_SCH_RED`)
- GRED queue (`CONFIG_NET_SCH_GRED`)
- DiffServ field marker (`CONFIG_NET_SCH_DSMARK`)
- Ingress Qdisc (`CONFIG_NET_SCH_INGRESS`)
- QoS support (`CONFIG_NET_QOS`)
- Packet classifier API (`CONFIG_NET_CLS`)
- TC index classifier (`CONFIG_NET_CLS_TCINDEX`)

- Firewall based classifier (`CONFIG_NET_CLS_FW`)
- U32 classifier (`CONFIG_NET_CLS_U32`)
- Traffic policing (`CONFIG_NET_CLS_POLICE`)

In some cases, additional elements may be needed. It is therefore recommended to enable all options in **QoS and/or fair queuing**.

Linux traffic control is configured with the utility **TC**. It is part of the iproute2 package. To build **TC** with DiffServ support, proceed as follows:

- Extract iproute2
- Edit `iproute2/Config` to enable DiffServ support:
  `TC_CONFIG_DIFFSERV=y`
- In `iproute2/`, run make

# Chapter 7: Conclusion

WideBridge is the first attempt to bridge the gap between the Linux traffic control model and the PowerNP model. At present only DiffServ has been implemented, but in the future other things such as NAT, firewall, tunneling, etc. might also be implemented along the same lines. Moreover, there is a need to extend the information base of the netlink messages so that designs could implement the netlink to NP API mapping in real time.

Finally the need for standardizing an implementation model needs to be emphasized. The multitude of implementation models leads to incompatibility between models and the focus thus shifts from translation to adaptation. Once standard implementation models are in place, the job of translation will be more accurate and reliable.

## References

- *Linux – Advanced Networking Overview Version 1*, Saravanan Radhakrishnan,
  http://qos.ittc.ukans.edu/howto/

- *Differentiated Services on Linux*, Werner Almsberger,
  http://diffserv.sourceforge.net/ (June 1999)

- *Linux Advanced Routing and Traffic Control HOWTO,* Bert Hubert, Gregory Maxwell (Section Author), Remco van Mook (Section Author), Martijn van Oosterhout (Section Author), Paul B Schroeder (Section Author), Jasper Spaans (Section Author), and Pedro Larroy (Section Author),
  http://lartc.org/howto/

- Linux netlink and rtnetlink manual pages

- *ManagementIinformation Base for the Differentiated Services Architecture*, IETF RFC 3289, F. Baker, K. Chan, and A. Smith,
  http://www.faqs.org/rfcs/rfc3289.html (May 2002)

- *Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers*, IETF RFC 2474, K. Nichols, S. Blake, F. Baker, and D. Black,
  http://www.faqs.org/rfcs/rfc2474.html (December 1998)

- *Differentiated Services Quality of Service Policy Information Base*, M. Fine, K. McCloghrie, J. Seligson, K. Chan, S. Hahn, A. Smith, and F. Reichmeyer,
  http://www.ietf.org/proceedings/00mar/slides/diffserv-pib-00mar.pdf
  (27 March 2000)

- *XMILE: An XML based Approach for Incremental Code Mobility and Update,* Cecilia Mascolo, Luca Zanolin, and Wolfgang Emmerich,
  http://www.cs.ucl.ac.uk/staff/c.mascolo/www/asemob.pdf

- *Netlink Sockets – Overview*, Gowri Dhandapani, and Anupama Sundaresan,
  http://qos.ittc.ukans.edu/netlink/html/ (March 1999)