A Platform-Independent API for Quality of Service Management

Günther Stattenberger Torsten Braun

Institute of Computer Science and Applied Mathematics
University of Bern
Neubrückstrasse 10
CH-3012 Bern

Marcus Brunner

C&C Research Laboratories
NEC Europe Ltd.
Adenauerplatz 6
D-69115 Heidelberg

Abstract— **The Differentiated Services approach to QoS in the Internet poses new challanges on the configuration and service provisioning side. The approach relies on an entity often refered to as a Bandwidth Broker. It configures the network elements so that guaranteed networking services are provided to customers. However, the DiffServ capabale routers have a large variety of hardware configurations and different configuration interfaces (e.g. SNMP, CORBA, CLI). Therefore, we propose in this paper a QoS management API which is used by Bandwidth Broker implementations in order to configure the underlying routers. In our prototype we implemented a proprietary configuration interface to our Linux-based DiffServ router implementation.**

I. Introduction

An important task for an ISP offering guaranteed services by using the Differentiated Services (DiffServ) approach to its customers is to configure the routers of its network according to the customers' requirements. The negotiation of technical service parameters between the customer and the ISP, also called a Service Level Specification (SLS), contains a description of the end-to-end Quality of Service the customer expects to be provided (even by routers that are not in the domain of the ISP). The resulting amount of configuration to own routers and communication to foreign networks is normally performed by an entity called a Bandwith Broker (BB).

In Figure 1 we show an example of a user requesting a certain service for a connection across two ISP's networks by sending a SLS to the first Bandwidth Broker. This broker interprets the SLS in its management section and forms a SLS to be forwarded to the neighbouring BB and a Traffic Conditioning Specification (TCS) for its own network. This TCS contains information about how each router has to be configured (on an abstract level) and will be forwarded to the configuration software that is able to create the corresponding configuration commands for each involved router.

A generic QoS management API is the scope of this paper. A bandwidth broker may use the API to configure different types of routers within its own domain. It will be an interface between the management software, which is not aware of the different types of routers within the underlying network and the configuration software, that is specially designed to optimally support each router type. The routers may differ in their capabilities or configuration interfaces. The main focus of this work is on the independence of the API from router hard- and software and on an easy extensibility to any kind of new routers.

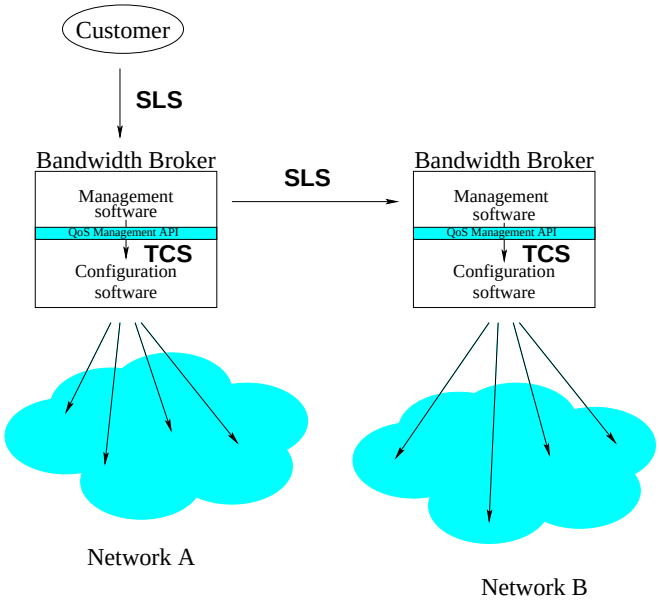

Fig. 1. Example of a general multi-broker scenario

It is not intended to propose any brokering algorithm or broker-to-broker communication interface, since this can be found elsewhere [4]. Nevertheless, we will show a broker application scenario as a first application of the developed API to the DiffServ implementation developed at our institute [3], [2].

This paper is organized as follows: in Section II we intro-

duce the design of the API and briefly describe each function seperately. Next, in Section III, we show how the abstract concept of the API can be applied to a real-world network consisting of Linux Routers. In Section IV we present a sample application using our API. A short conclusion is given at the end of the document.

II. Design of the QoS Management API

The independence of the API from router hardware or implementation details is best achieved using an object oriented design where the objects representing different implementations are derived from a common base class. The ideal solution for this polymorphism is to create abstract classes containing a certain set of virtual functions (methods) that have to be implemented by each derived class. This function set forms the interface by which the class can be accessed and conceals the different implementations from the application programmer.

Management software will likely be a critical part of an ISP's network management system. First, performance is an important issue. To be able to handle a large amount of user interactions together with the resulting configuration communication we have chosen C++ as the programming language offering OO concepts at high performance. But another issue is also very important: Since an ISP will certainly loose both revenue and customer goodwill if its network is temporarily unavailable for rebooting (e.g. to upgrade the system) this is undesireable yet normally inevitable. An effective way to meet this challenge is using dynamic code which ensures an up-to-date system running continously. The use of dynamic C++ classes [7] combines the advantages of the object oriented design and dynamic linking to a management system that provides permanent availability together with the possibility to integrate new hard- or software features to the system during runtime.

The QoS management API consists of three abstract base classes (represented in C++ notation in Figure 2): the `Router`, the `Interface` and the `TrafficConditioner` (`TC`). Additionally some data classes representing e.g. IP addresses or flows are used in the functions. The base classes provide a generic interface to the programmer by their virtual functions. An application programmer can build a virtual image of the network that is to be configured consisting of objects derived from an API-`Router`. A configuration application can send generic configuration commands to each `Router` object. The objects will then translate them to the hardware-specific commands that are finally sent to the router.

In the following subsections we describe the member functions of those base classes together with the implementation-independent data classes, that are not shown in Figure 2.

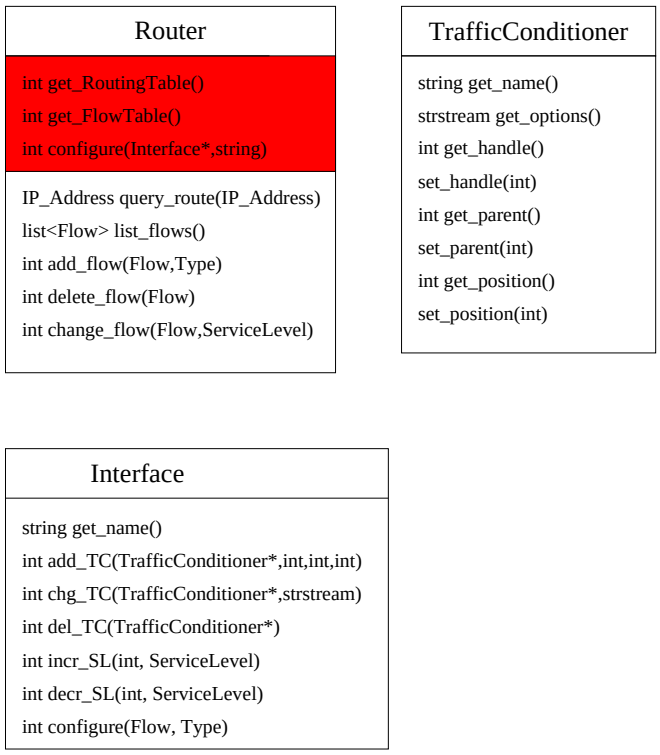

Fig. 2. Overview of the API classes

A. Data Classes for the API

The `IP_Address` class is the parent class providing the common functionalities of IPv4 and IPv6 addresses. Some of those functions are virtual, so that they can be redefined by the child classes (especially the output functions, providing the dotted decimal notation for IPv4 and the colon-separated byte notation for IPv6).

The `Route` class provides a generic routing information subset, containing the destination and gateway addresses, the netmask and an interface identifier. The routing table of a real-world router is mapped to a list of `Route` objects, representing the routing table in the generic `Router` class.

The `ServiceLevel` class is a generic service level for a DiffServ Router. It contains mainly a bandwidth value, that specifies the average amount of traffic of a flow. Depending on the service that is requested, additional parameters for excess bandwidth or maximum delay can be specified.

The `Flow` class is the basis for flow tables, that are lists of `Flow` instances. The `Flow` class contains a 6-Tuple with IP information (i.e. source- and destination address, source- and destination port, protocol ID, DSCP). Note that the IP addresses may be wildcarded, whereas the other parameters are single valued. Additionally, each `Flow` has a `ServiceLevel` object for the parameters concerning the traffic profile.

B. The `Router` Class

The `Router` class provides a set of functions that every vendor of a DiffServ router has to provide in order to be manageable by our software. Its child classes will contain all data

structures for interfaces and the routing- and flow tables. Those data structures can be the data classes presented in Section II-A, but also hardware - dependent classes can be used as long as they provide an interface to the API functions.

The set of member functions of the `Router` class can be devided into two parts (see Figure 2): the dark marked members are used for communication with the real world router and its interfaces and their use is restricted. Management applications can only use the other functions to manage the router.

The *get_type()* function returns some type information about the router that can be used by the application to develop device-specific code.

The *get_name()* function returns the hostname of the router. The IP address is fetched by usual DNS resolve methods.

The *query_route(IP_Address)* function returns the IP address of the next hop for traffic to the given destination address. The translation of IP addresses to a pointer to the corresponding `Router` representation can easily be made by an external map.

The *get_RoutingTable()* function is used to fetch the routing table from the real world router it represents.

The *get_FlowTable()* function is designed for getting the flow table of the router. If there is more than one flow table in the router, they have to be mapped to a single one. Those flow table contains information about the flows that will be handled by the DiffServ implementation. The corresponding `FlowTable` data type is a list of `Flow` instances (see above).

The *configure(Interface*,string)* function is used for sending a router configuration command to the interface pointed to by the first argument. The interface related part is held in the `string` argument. The `Router` class can choose the way of communication and perhaps add the correct preamble to the command depending on the interface type of the first argument. This function is designed to be called by the `Interface` class after composing the correct command line.

The *list_flows()* function gives a complete list of all flows that are registered at this router.

The functions *add_flow(Flow, Type)*, *delete_flow(Flow)*, *change_flow(Flow, ServiceLevel)* are used to change the content of the router's flow table. The `Flow` data class is a router-independent flow description (see section II-A). The `Type` argument of the `add_flow` function specifies how to configure the router type for this flow. This parameter refers to the particular flow only. It denotes whether the router acts as an ingress, intermediate or egress router for that flow. Those functions can also call the `Interface` member functions to add the necessary traffic conditioners and/or adapt the service level.

C. The `Interface` Class

The `Interface` class is designed to represent the whole set of network interfaces that could be part of a router. `Interface` objects are most likely created by the child classes of the `Router` during initialisation. Each `Interface` provides member functions that are important for the QoS management:

The *get_name()* function returns an identifier of the interface. This identifier could e.g. be the name of the interface within the router.

The *add_TC(TC*,int,int,int)* function adds a traffic conditioner to the forwarding path of the interface. The location within the forwarding path is addressed by the three integer parameters: each traffic conditioner has its unique identification number called `handle` (the first parameter) and it is normally attached to some other traffic conditioner, whose identification number is given as the second parameter (the `parent`). If the conditioner is the root of the forwarding path this parameter is zero. Some traffic conditioners (e.g. classifiers) can have multiple conditioners attached. This can be indexed by the third parameter (called `position`). The `add_TC` function can create the necessary traffic conditioner together with the correct parameters, which can be calculated from the actual service level.

The functions *del_TC(TC*)* and *chg_TC(TC*,strstream)* are used to change the forwarding path of the interface. Since the information of the location of the traffic conditioner within the forwarding path resides in the conditioner itself there is no need for another parameter. The `strstream` parameter of the `chg_TC` function is a generic way of setting different numbers and types of parameters for different traffic conditioners.

The two functions *incr_SL(int,ServiceLevel)* and *decr_SL(int, ServiceLevel)* change the overall service level of a certain service at this interface. They increase or decrease the allocated bandwidth of the service and adapt the excess bandwidth or the delay limits.

The *configure(Flow,Type)* function arranges the correct traffic conditioners in the forwarding path of the interface for the given flow. The arrangement of the traffic conditioners depends on the `Type` parameter, that specifies, whether the interface should be configured as an ingress, an egress or any other type of interface.

D. The `TrafficConditioner` Class

The `TrafficConditioner` class represents the concept of modules that are used in the forwarding path of an interface in order to perform the necessary actions to comply with the DiffServ Per-Hop-Behaviours [6],[5]. For QoS management it is not necessary to simulate the exact behaviour of each conditioner. We need but one class for each "real-world" conditioner that knows the way how to address its counterpart (e.g. command line syntax). How to compose the traffic conditioners to support a certain service has to be programmed within the `Interface` class. Some common rules for boundary and core routers can be used to automate the setup of DiffServ routers.

The *get_name()* function returns the name of the traffic conditioner.

The *set_handle(int)*, *get_handle()*, *set_parent(int)*, *get_parent()*, *set_position(int)* and *get_position()* functions handle the location parameters of the traffic conditioner within the forwarding path of an interface. The `set` functions are used by the

`Interface::add_TC` member function to specify the three parameters. The `get` functions can be used by any application.

The *get_options()* function returns a `string` that contains the command-line options which have to be passed to a configuration command to set up this special traffic conditioner. As traffic conditioners of different types may have multiple and very different parameters (e.g. rate and depth for a token bucket filter or queue weights for a weigthed fair scheduler) it is the most generic way to support different types of conditioners.

Note that there is no function for setting the parameters, because this can be easily done by the traffic conditioner's constructor call. The command line string can then be composed from the constructor arguments. In order to change the parameters of already existing and installed traffic conditioners we can use the `Interface::chg_TC()` function.

We have chosen only one base class of traffic conditioners (traffic conditioning blocks in [1]). Conceptually, we could have derived different kinds of blocks from that base class, such as classifiers, meters, markers, packet schedulers, queues etc), and we could again derive from a packet scheduler a weighted packet scheduler and a priority scheduler, and so on. This would lead to an inflation of classes differing in details only. It is better to let the application programmer choose a suitable set of traffic conditioners to manage and configure the underlying network.

E. The Factory of Dynamic Classes

To ensure a common behaviour of different router types and for providing a generic configuration interface the approach of abstract base classes has been chosen. However, it is very important, that different derived classes implementing the functionality of e.g. new router types can be provided without recompiling and even without restarting the configuration software. For this purpose, the API provides three so-called factory maps — one for each abstract base class. Those factory maps provide the constructors of all derived classes, and the constructors can be indexed by the class name. Encountering a new class name the map tries to locate and open a dynamically loadable object file with that specific name. The programmer of that file has to ensure that it will register itself at the correct factory map at the time, when the dynamic object file is loaded by the linker. This can be done by instancing a proxy class which performs the registration within its constructor [7]. A `LinuxRouter` object can now be instanciated by

```
Router* LR=factory_Router["LinuxRouter"];
```

III. An Implementation of the API for Linux Routers

For each type of hard- or software element (i.e. routers and network cards from different vendors or traffic conditioners with different algorithms) that is used and managed within the network, an own class derived from one of the base classes must be provided, that implements the functions of the API according to the needs and the functionality of the element it represents. This set of classes must be dynamic, so that the management application can load them during run-time. This enables us to support network elements, that were unknown at the time when we were compiling the management software.

Another important issue is to specify a communication interface between the API classes and the routers. This depends on an implementation specific way to read and/or write certain variables of the router. For example it can be achieved by SNMP messages or CLI configuration for many commercial routers.

Using the dynamic classes the application programmer is now able to construct a logical image of the network he intends to manage by his application. All management is then performed by calling the API functions. Those calls are then translated to hardware / implementation dependent function calls and afterwards sent to the individual network elements.

A. API Child Classes for a Linux Router

Figure 3 illustrates how the API classes are used to derive classes for modelling a Linux DiffServ Router, so that the application can use it and communicate with it. Each of those child-classes must implement the API functions in a hardware-specific way. The generic data classes (see Section II-A) can be used to model the important parts of a router (e.g. the routing table, the flow table or some information about the network interfaces) in a way suitable for the API. Using the communication interface to connect to its hardware - counterpart, the child class can execute the necessary commands and fetch and filter the desired information, so that the generic data classes can be initialized.

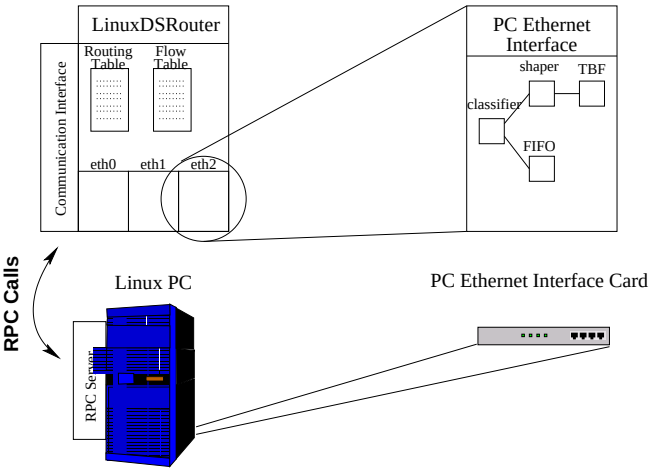

Fig. 3. An application example for the abstract API classes

B. The Communication Interface for Linux Routers

The communication interface for the Linux router is based on RPC calls, because the RPC API provides useful functions

like broadcast calls, callback functions, authentication and encryption. Since it is intended to enable a network administrator to set up, configure and monitor the QoS support of his network just as he would sit in front of each router's console, we have to provide a generic way to send configuration commands to a router. Those commands will be created by the classes representing the parts of the router that have to be configured. So we ensure, that each new network interface or traffic conditioner can be configured in a device-specific way. The command lines will not be parsed but simply be executed by the RPC server.

This shell-related part can be used to send configuration commands to the router, but it is not suitable to transport large amounts of data (e.g. routing tables). Fortunately, this happens only during the initialisation phase and is not likely to change, as it is the case for e.g. traffic conditioners. Therefore, a separate set of RPC calls is provided for initializing the class' data members. The third part of the RPC calls is used for advertising and solicitation messages. Those functions can be used by an application to detect the routers by broadcast calls, so no knowledge about the topology is required.

Figure 4 shows the RPC functions of the server at the Linux DiffServ router. Of course each block can be arbitrarily extended if new functionality is required.

QoS Management RPC Server		
Initialisation	*Command Interpreter*	*Autoconfinguration*
get_RoutingTable() get_FlowTable() init_FT(list<Flow>)	execute(string)	reply() solicit()

Fig. 4. The RPC server at the Linux DiffServ Router

IV. An Application Example

A possible application scenario is shown in Figure 5.

1. The application sends a broadcast call to all RPC servers on the network during initialisation.
2. Each RPC server answers the broadcast call by invoking its `reply()` function. Now some information about each router is sent back to the application (e.g. the hostname, router type, ...).
3. The application creates a `Router` object of the according type for each reply.
4. Each router object invokes the `get_RoutingTable()` function and fetches the routing information available at the corresponding router.
5. According to the information of the routing table several `Interface` objects are created and attached to each router.

Now the ISP's network is mapped to a logical image running within a management application on a remote workstation (see Figure 5). This image consists of several objects of derived API classes. The administrator configures this logical image via the API functions using for example some kind of command-line or HTML interface. Those commands are in turn translated to RPC calls to the servers running on each Linux Router.

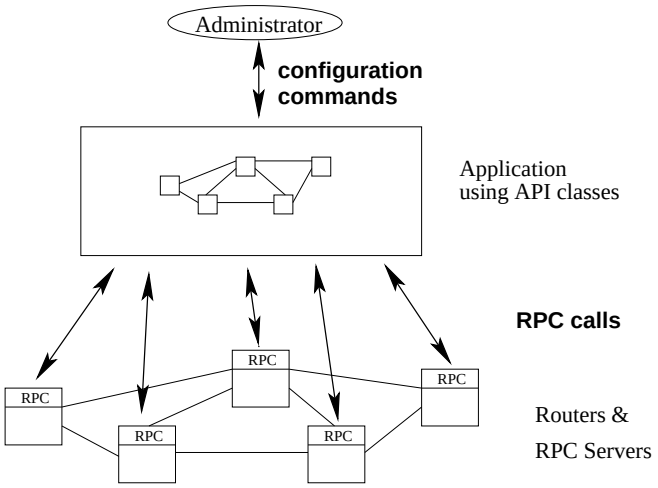

Fig. 5. A typical application scenario

V. Conclusion

We proposed and implemented an object oriented Quality of Service Management System, that is independent of the router hard- and software. A heterogenous network can be managed by building a homogenous virtual network consisting of instances of the `Router` class. We also implemented a sample configuration-application that uses derived API classes adapted to our implementation of a Linux DiffServ router. An enhanced version of such a configuration / brokering software is subject of future work.

References

[1] Y. Bernet, A. Smith, S. Blake, and D. Grossman. A conceptual model for DiffServ routers. Internet Draft, March 2000. work in progress.

[2] T. Braun, A. Dasen, M. Scheidegger, K. Jonas, and H. Stüttgen. Implementation of differentiated services over ATM. In *Proceedings of the IEEE Conference on High Performance Switching and Routing HPSR 2000*, pages 317 – 322. IEEE Computer Society, June 2000.

[3] T. Braun, M. Scheidegger, H. Einsiedler, G. Stattenberger, and K. Jonas. A linux implementation of a differentiated services router. In Sathya Rao and Kaare Ingar Sletta, editors, *Next Generation Networks — Networks and Services for the Information Society*, volume 1938 of *Lecture Notes in Computer Science*, pages 302 – 315, October 2000.

[4] M. Günter and T. Braun. Evaluation of bandwidth broker signaling. In *Proceedings of the International Conference on Network Protocols ICNP'99*, pages 145 – 152. IEEE Computer Society, November 1999.

[5] J. Heinanen, F. Baker, W. Weiss, and J. Wroclawski. Assured forwarding PHB group. RFC 2597, June 1999.

[6] V. Jacobson, K. Nichols, and K. Poduri. An expedited forwarding PHB. RFC 2598, June 1999.

[7] J. Norton. Dynamic class loading for C++ on linux. *Linux Journal*, (73), March 2000. http://www2.linuxjournal.com/lj-issues/issue73/3687.html.

[8] F. Reichmeyer, L.Ong, A. Terzis, L. Zhang, and R.Yavatkar. A two-tier resource management model for differentiated services networks. Internet Draft, November 1998. work in progress.