Dynamic Resource Allocation in Core Routers of a Diffserv Network

Rares Serban, Chadi Barakat, Walid Dabbous

PLANETE Project, INRIA Sophia Antipolis, France
{Rares.Serban,Chadi.Barakat,Walid.Dabbous}@sophia.inria.fr
http://www-sop.inria.fr/planete/index.html

Abstract. The Differentiated Services (DiffServ) architecture is receiving wide attention as a framework for providing different levels of service according to a Service Level Agreement (SLA) profile. The edge routers in a DiffServ network mark/shape/police flows based on their SLAs, and the core routers offer packets different treatments using the marks they carry. Core routers handle aggregates of flows instead of individual flows, which is known to considerably reduce the complexity of DiffServ. Tuning core routers is clearly an important issue to satisfy the needs of traffic marked at the edges. This tuning is actually characterized by extensive manual work, based on a trial-and-error process, which is often ineffective, time-consuming and costly to network managers. In this paper, we propose a dynamic, self-tuning mechanism for allocating resources in core routers among Diffserv services. Our mechanism is easy to implement, and does not require any particular signaling. It ensures that SLAs are respected, and allows at the same time an efficient utilization of network resources. We validate the performance of our mechanism by a campaign of experiments on a real network testbed.

1 Introduction

Network customers need guaranteed level of service from their Internet Service Providers (ISPs) to achieve their business objectives. Service Level Agreements (SLAs) between providers and customers define service level specifications [1] with traffic conditioning specification [2], monitoring service capabilities [3], service availability and the fees corresponding to each level. The traffic conditioning specification (TCS) defines service level parameters (bandwidth, packet loss, peak rate, etc.), offered traffic profiles and policies for excess traffic. Given the SLA of a stream of packets, the network has to allocate enough resources so that the service required by this stream is guaranteed. The allocation can be done in different ways, depending on the total amount of resources available in the network and the number of customers asking for a guaranteed service. One simple allocation is the one that uses the peak rate of traffic. This is the type of allocation supported by PSTNs (Public Switched Telephone Network). Another possible allocation is the one that uses the notion of effective bandwidth [4]. The effective bandwidth for traffic is the minimum bandwidth to be allocated in the network so that the probabilistic needs of the traffic (e.g., packet loss rate, tail of packet delay) are satisfied. In the effective bandwidth framework, the

statistical multiplexing of the different streams of packets is used to minimize the total amount of resources to be allocated. Note that for any resource allocation scheme, there is always a maximum limit on the number of customers the network can support. When the number of customers approaches this limit, the resources start to be rare and the Quality of Service (QoS) required by customers cannot by realized. Thus, some kind of Call Admission Control (CAC) [5] has to be implemented by the network, to protect already accepted customers from newly arriving ones.

Differentiated Services (Diffserv) is an IETF framework for classifying network traffic into classes, with different service level for classes [6]. The edge routers in a DiffServ network mark/shape/police flows based on their SLAs, and the core routers offer packets belonging to these flows different treatments using the marks they carry. A flow is a stream of packets belonging to the same SLA. Core routers handle aggregates of flows instead of individual flows, which is known to considerably reduce the complexity of DiffServ, compared to its counterpart IntServ [7], where core routers allocate resources on a per-flow basis. The treatment a core router gives to packets from one service class is called PHB (Per Hop Behavior). The PHB classes (or service classes) defined in DiffServ are: Best Effort (BE), Assured Forwarding (AF) [8][9] and Expedited Forwarding (EF) [10][11]. The EF class is designed to support real time flows with hard delay and jitter constraints. The AF class is designed for flows only asking for bandwidth, mainly TCP flows. The AF has four (sub)classes, each one with three drop precedence [6]. At the onset of congestion, core routers start drop AF packets with the highest drop precedence, then those with the medium drop precedence, and finally if congestion persists, packets with the lowest drop precedence are dropped. Packets within a AF class are served in core routers in order. The different AF classes may differ in the applications and transport mechanisms they support. For example, TCP traffic can be protected from non-responsive UDP traffic by separating both types of traffic in two different AF classes. How to distribute traffic on the different service classes of DiffServ is out of the scope of this paper.

The performance of a DiffServ network is strongly dependent on how well edge and core routers work. Edge routers are responsible of marking/shaping/policing traffic. Core routers give packets different treatments based on the marks they carry. For example, EF packets are queued in a separate buffer and are served before packets of the other classes. Packets of the different AF classes are queued in separate buffers and are served using the CBQ mechanism [12]. We focus in this work on the tuning of core routers for AF traffic. This operation reflects the tuning of the weights of the CBQ buffer. The weights of CBQ control the way with which the available bandwidth at the output interface of the core router is distributed among the AF classes and the BE class. The weights do not have any control on the EF class, since EF packets are usually served with a strict priority over the other types of packets (AF and BE). The tuning has to be done so that each class of traffic realizes its needs, and the resources of the network are efficiently utilized. In particular, the tuning has to ensure that the BE traffic is not penalized by the AF traffic, and at the same time, that the BE traffic does not consume more than its fair share of the available bandwidth.

The tuning of core routers is actually characterized by extensive manual work, based on a trial-and-error process, which is often ineffective, time-consuming and costly to network managers. As we will explain later, a static tuning of core routers

may result in an inefficient utilization of network resources, and a bias against one or more DiffServ classes. We believe that the tuning of core routers has to be dynamic, so that the available bandwidth is efficiently utilized and fairly shared among the different DiffServ classes (more details in Section 4). Moreover, the tuning has to be automatic, self-configurable, easy to deploy and to manage, without any additional signaling. In this paper, we motivate the dynamic tuning of core routers and we present a mechanism that respects the above rules. We implement our mechanism in Linux and we validate its performance by a campaign of experiments on a real network testbed.

While presenting our mechanism, we will make the assumption that the network is over-provisioned, i.e., there are enough resources in the core of the network to support the high priority traffic marked at the edge. On the other hand, the resources of the network may not be enough to support the total amount of traffic coming from the edge, i.e., the high priority plus the low priority traffic. The decision on whether to accept a new SLA is done at the edge of the network, and core routers dynamically tune their parameters so as to absorb the marked traffic and to utilize efficiently the network resources. Our mechanism can be easily extended to the under-provisioning case, by making some default assumption on the distribution of the bandwidth among the DiffServ classes. For example, a core router may give the highest priority to EF packets, and set equally the weights of the CBQ buffer for AF classes, with zero or a minimum amount of bandwidth to the BE class. We believe that the under-provisioning case is undesirable for the ISP and the customers and has to be avoided, which can be done by implementing CAC at the edge, or by renegotiating the SLAs of active customers.

We consider the allocation of bandwidth based on the average rate of high priority traffic of each DiffServ class. For simplicity of the analysis, we omit the EF service and we focus on AF and BE. EF is usually handled by a priority queue, without any special tuning to be done by the core router. Without loss of generality, we focus on a AF service with two drop precedence per-class. This latter service has been first introduced in [11]. At the edge, compliant packets of a AF class are marked with high priority (called IN packets), and non-compliant packets are injected into the network with low priority (called OUT packets). IN and OUT packets are buffered in the same queue in core routers, but are dropped differently at the onset of congestion. The idea is to start dropping OUT packets while protecting IN packets. When all OUT packets are dropped and the congestion persists, IN packets start to be discarded. The mechanism proposed in [11] to support such a preferential dropping is called RIO (RED IN/OUT).

In summary, our mechanism measures the rate of IN packets, and sets the parameters of the CBQ buffer so as to absorb all IN packets and to distribute fairly the rest of the bandwidth (called excess) among OUT and BE packets. As a case study, we consider a max-min fairness for the allocation of the excess bandwidth [13][14].

The remainder of this paper is organized as follows. The problem of tuning core routers is explained in Section 2. Section 3 presents our mechanism, which dynamically tunes core routers to allocate efficiently the available resources among DiffServ classes. Section 4 illustrates on some examples the drawback of static tuning, and motivates the dynamic tuning of core routers. Section 5 explains our experimental

evaluation environment. Section 6 validates our mechanism with our experimental environment. Finally, Section 7 concludes the paper and gives some perspective on our future research activities in this direction.

2 Problem Description

The tuning of core routers in a Diffserv network is actually done in a static way by manual work based on a trial-and-error process. A static tuning is time-consuming and costly for network managers. Moreover, as we will see in Section 4, a static tuning may lead to an inefficient utilization of network resources and to an unfairness among the DiffServ classes. A dynamic tuning of core routers, or equivalently a dynamic allocation of resources in the core of the network, is needed to satisfy as much as possible all reservations of customers, and to fairly distribute the excess of bandwidth among them.

Consider a DiffServ network proposing to customers two AF services (AF1 and AF2), in addition to the classical BE service. This will be the type of networks we will consider throughout the paper. Our results hold in the case when more than two AF service classes are proposed. They also hold in the case when the network operator proposes the EF service to its customers.

A customer asks the network for some bandwidth of some class (AF1 or AF2). It may also ask the network for a simple BE service, with no bandwidth guarantee. As mentioned in Section 1, the difference between the two AF classes can be in the transport protocol used (TCP vs. UDP), in the number of customers authorized to join each class, in the policy applied to non-compliant packets, etc. The bandwidth allocated to a user of a AF class represents the maximum rate of IN packets the customer is allowed to inject into the network. All packets non compliant with the contract signed between the customer and the operator are marked as OUT at the edge. Some routers at the edge may choose to drop OUT packets instead of injecting them into the network. This is what we call shaping of flows.

Let $R_{AF1.1}$ (resp. $R_{AF2.1}$) denote the rate of data carried by IN packets of class AF1 (resp. AF2), which arrive at a core router and are destined to the same output interface. Let C be the total bandwidth available at the output interface of the core router. We are working in an over-provisioning case, which means that there is enough resources to absorb the high priority IN traffic. This can be written as,

$$R_{AF1.1}+R_{AF2.1} \leq C . \tag{1}$$

Now, denote by R_{AF1} (resp. R_{AF2}) the total rate of data carried by AF1 packets (resp. AF2 data) (IN + OUT). Denote by R_{BE} the data rate of the BE traffic arriving at the same core router and destined to the same output interface as the AF1 and AF2 traffic. Dynamic tuning is only interesting in the case when the core router is congested. Core routers being work-conserving, the tuning of parameters does not impact the QoS perceived by customers in a non-congestion case. Our assumption is then,

$$R_{AF1} + R_{AF2} + R_{BE} \geq C. \tag{2}$$

We consider that a customer is satisfied if its IN packets get through the network without being dropped. We also consider that it is in the interest of customers and operator that the excess of bandwidth in the network is fairly distributed among the three classes: AF1, AF2, and BE. The excess of bandwidth is equal to $C-R_{AF1.1}-R_{AF2.1}$. Given the variability of the traffic and the change in SLAs, it is very likely that a static tuning of core routers does not realize the above objectives. The problems that can be caused by a static tuning of weights at the output interface of a core router, can be summarized as:

- Unfairness in the distribution of the excess bandwidth;
- Bias against the IN packets of one or more AF classes.

We will study these problems of static tuning in Section 4 on some real scenarios. We will show the gain that a dynamic tuning of weights can bring. The comparison will be made between a stating tuning scheme, and our mechanism that dynamically adapts the weights at the output interface of routers, based on the incoming IN traffic. Our mechanism is designed with the main objective to realize the above objectives, that is, to accept the high priority IN traffic of the two AF classes, and to distribute the excess bandwidth fairly among the three classes we are considering: AF1, AF2, and BE. We start by explaining our mechanism in the next Section. In Section 4, we compare it to a static tuning scheme, and in Section 6 we validate its performance using our experimental network testbed (explained in Section 5).

3 Dynamic Resource Allocation Algorithm

Our algorithm has three parts: monitoring, excess bandwidth distribution rule and CBQ scheduler programming.

In the monitoring part, we measure the average data rate carried by IN packets for each one of the two classes AF1 and AF2. The interval over which the data rate is averaged is an important parameter of our mechanism. It must not be too small, since the system may become unstable, especially when we have a transport protocol like TCP, which adapts its rate as a function of the reaction of the network. A small averaging interval also results in high computational overhead. The averaging interval must not be very large too, since this slows the reaction of the mechanism to changes in traffic and SLAs. We will come later to the impact of this averaging interval on the efficiency of our mechanism with some experimental results.

The excess bandwidth distribution rule part forms the core of the algorithm. With no loss of generality, we use the following rule: the excess bandwidth is split into three parts and equally distributed among the three classes. We call this rule *fair division*. Other rules are also possible. For example, one can distribute the excess bandwidth among classes using non equal weights. The objective might be to give AF service classes bandwidth advantages over BE, or the opposite.

After the distribution of excess bandwidth, the algorithm computes the desired rates for CBQ classes (the optimal weights for each class), and programs the CBQ mechanism with these rates. The algorithm has the following description:

```
while(){
    - Measure the throughput of IN-profile packets for
      both classes AF1 and AF2, and for each outgoing
      network interface: R_{AF1.1}, R_{AF2.1};
    - Compute the amount of excess bandwidth using
      measured values: Bw = C - (R_{AF1.1} + R_{AF2.1});
    - Distribute the excess bandwidth following some
      rule. Ex: w = Bw/3;
    - Program the CBQ scheduler using the computed op-
      timal rates: R_{AF1} = R_{AF1.1} + w, R_{AF2} = R_{AF2.1} + w,
      R_{BE} = w;
    - Wait x seconds;
}
```

Let us illustrate the operation of our algorithm with the following example, taken from Section 4.2.2. We have a scenario where the AF1 service is used by a UDP traffic sending data at a constant rate 4Mbps, and the AF2 and BE services are used by TCP traffic. The TCP traffic is generated by means of long-lived TCP connections. For both service classes AF1 and AF2, the rate of IN-profile data is set to: $R_{AF1.1}$=2.5Mbps and $R_{AF2.1}$=2.5Mbps. The available bandwidth at the output interface of the core router is set to C=6Mbps. We start from a case where the bandwidth C is equally divided among the three service classes: R_{AF1}= R_{AF2}= R_{BE}=2Mbps. After the first iteration, our mechanism sets: R_{AF1}= 3Mbps, R_{AF2}= 2.5Mbps and R_{BE}= 0.5Mbps. The next iteration measures $R_{AF1.1}$= 2.5Mbps and $R_{AF2.1}$= 2.5Mbps. The algorithm stabilizes then with the following rates for the CBQ buffer: R_{AF1}= 2.8Mbps, R_{AF2}= 2.8Mbps and R_{BE}=0.3Mbps. The convergence is achieved in two steps. Clearly, this allocation of bandwidth C satisfies the two objectives of our mechanism. Indeed, $R_{AF1.1}$ and $R_{AF2.1}$ are satisfied, and the excess bandwidth (C-$R_{AF1.1}$-$R_{AF2.1}$)=1Mbps is fairly divided among the three service classes.

4. Simple scenarios to compare static and dynamic bandwidth allocation schemes

The goal of this section is to compare static and dynamic bandwidth allocation schemes using simple scenarios. We show how a static scheme results very likely in an efficient utilization of network resources, and a bias against one or more service classes. As a dynamic scheme, we use our mechanism, that we described in Section 3.

4.1 Unfairness in the distribution of the excess bandwidth

Consider the case when we guarantee 2Mbps for the AF1 service, $R_{AF1.1}$ = 2Mbps, and nothing for the AF2 service, $R_{AF2.1}$ = 0. The CBQ scheduler in the core router statically limits each service class to 2Mbps. The link capacity is set to C=6Mbps. The traffic from one class cannot exceed 2Mbps, even if it is reserving much more bandwidth than the traffic in the other classes.

Suppose that the class AF1 wants to send more traffic than is guaranteed (R_{AF1} = 4Mbps), then this class will not obtain any share from the excess bandwidth, since the core router is limiting its maximum rate to 2Mbps. The other classes, which do not reserve any bandwidth, monopolize the excess bandwidth, which is completely unfair to class AF1. The weights of CBQ are clearly not adapted to this situation.
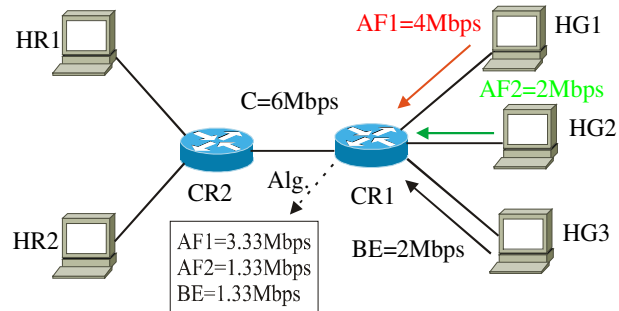

Fig.1. Unfairness in the distribution of the excess bandwidth case. HG - host generators and HR - host receivers. CR - core routers

Our mechanism solves this problem and allows a fair sharing of the excess bandwidth, which improves the AF1 service in this scenario. The rates allocated by our algorithm are shown in Fig.1. These rates are obtained with a fair division rule, which divides the excess bandwidth equally among the three service classes. Using this rule, we increase the AF1 bandwidth from 2Mbps to R_{AF1}=3.33Mbps, and we penalize the other services that do not ask for any guaranteed bandwidth, $R_{AF2}=R_{BE}$=1.33Mbps instead of 2Mbps in the static case.

4.2 Bias against the IN packets of one or more AF classes

A customer is satisfied when the IN-profile packets it injects into the network succeed to get through. We recall that we are considering an over-provisioning case, otherwise this condition on satisfaction is not feasible. A static tuning of core routers may violate this condition, by penalizing a class sending only IN-profile packets, while accepting OUT-profile packets from another class, or accepting packets from the BE class. We illustrate this misbehavior with the following two scenarios. Our mechanism ensures a complete protection of IN-profile packets in an over-provisioning case. It does this by adapting the rates of CBQ to the rates of IN packets.

4.2.1 One Reservation is Not Satisfied and One is Satisfied

Consider the scenario where the AF1 class generates a total data rate of 4Mbps, and the AF2 class generates a total data rate of 0.5Mps: R_{AF1}=4Mbps and R_{AF2}=0.5Mbps. All packets of the AF2 class are marked as IN, $R_{AF2.1}$ = 0.5Mbps. The rate of data carried by IN packets of class AF1 is equal to $R_{AF1.1}$ = 3Mbps. In the static tuning

case, the CBQ buffer is supposed to distribute the bandwidth C=6Mbps equally among the three classes. The BE traffic rate is set to 3Mbps.

In this scenario, class AF2 is satisfied since all its IN packets get through the network. However, class AF1 is not satisfied, since the CBQ buffer limits its rate to 2Mbps, whereas it is sending IN packets at a higher rate of 3Mbps. We are in a scenario where IN packets are dropped, and OUT/BE packets are served instead. The total rate of IN packets generated by the two classes is less than C, therefore it is possible to find another tuning that allows all IN packets to get through, and corrects the bias against class AF1. When running our mechanism, the rate allocated to AF1 increases to more than $R_{AF1.1}$, and the rate allocated to AF2 is kept larger than $R_{AF2.1}$ (Fig.2).
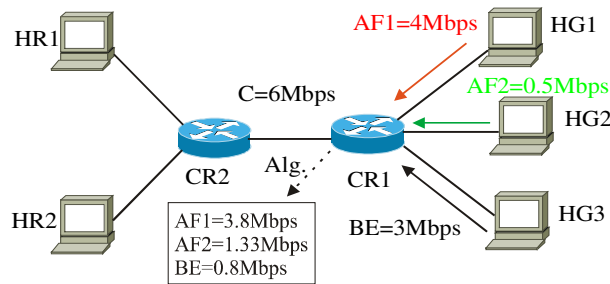

Fig.2. One reservation is satisfied from HG2. HG - Host Generator, HR - Host Receiver, CR - Core Router

Concerning the excess bandwidth (equal to 2.5Mbps in this scenario), our mechanism divides it equally among the three classes. So, the new distribution of the CBQ rates is: R_{AF1}= 3.8Mbps, R_{AF2}= 1.33Mbps and R_{BE}= 0.8Mbps.

4.2.2 All Reservations are Not Satisfied

The last case we consider is when all reservations are not satisfied, due to an appropriate static tuning of CBQ weights. Suppose AF1 has a IN-profile traffic $R_{AF1.1}$=2.5Mbps and AF2 a IN-profile traffic $R_{AF2.1}$=2.5Mbps. If the CBQ limits the maximum rate of each class to 2Mbps, both classes AF1 and AF2 will not be satisfied since some of their IN packets will be dropped. On contrary, clients of the BE class are favored, since they obtain more than their fair share of the excess bandwidth (2Mbps instead of 0.33Mbps).

We apply our algorithm and we measure each class throughput at the output interface of the core router. The rule to divide the excess bandwidth is the same as that in the above sections. After a while, the rates allocated in the CBQ buffer to the different classes change: AF1 and AF2 receive R_{AF1}=R_{AF2}=2.8Mbps, and BE receives R_{BE}=0.3Mbps. This is exactly the desired allocation, that satisfies customers who ask for bandwidth, and distributes the excess bandwidth equally among the three classes.
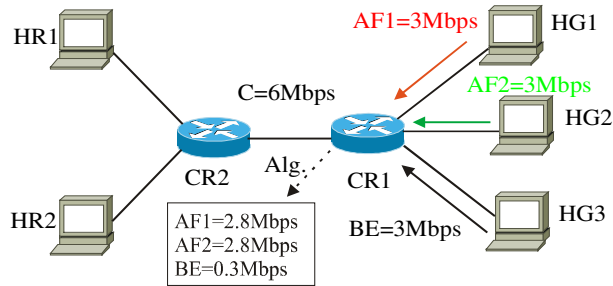
Fig.3. All reservations are not satisfied. HG - Host Generator, HR - Host Receiver, CR - Core Router

5 Evaluation Environment

The evaluation environment includes monitoring tools, traffic generator tools, traffic control tools and link emulator tools. The experimental results are based on cases presented in Section 4.

To validate our work, we use a test-bed network (Fig.4) with Linux Redhat 7.2 operating system. All the PC's are PIII with different clocks' speed processors and different network cards.

Our mechanism is implemented in the core router named Kisscool. Looking to Fig.4, each host at the right-hand side (Galak, Mnm, Raider) generates TCP or UDP traffic flows [15] to hosts at the left-hand side. The traffic is generated with the Iperf tool, and is marked at the output interfaces of the source hosts. The network bandwidth is 10Mbps. Using the NIST Net tool [16], we emulate a link with bandwidth 6Mbps between Kisscool and Kitkat routers.

The monitoring tool used is TCPSTAT [17]. It collects the throughput of each AF class at the incoming interfaces of the core router. To be effective, TCPSTAT works in Eth3, Eth2, Eth4 from Kisscool router.

We use the tc (traffic control) module implemented in the Linux Diffserv [18][19]. Traffic control can decide if packets are queued or if they are dropped (e.g., if the queue has reached some length limit or if the traffic exceeds some rate limit). It can decide in which order packets are sent (e.g. to give priority to certain flows). It can delay the sending of packets (e.g. to limit the rate of outbound traffic), etc. The traffic control code is implemented in Linux kernel and is configured with a command interface tc. The main components of Linux traffic control are filters, classes and queuing disciplines (e.g., qdisc).
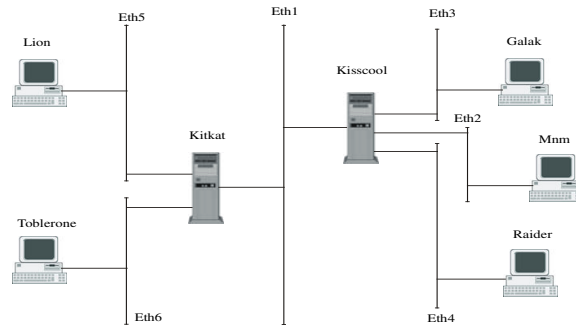
Fig. 4. Our test bed network. The core router is Kisscool host. Flow generators are instaled on Galak, Mnm and Raider.
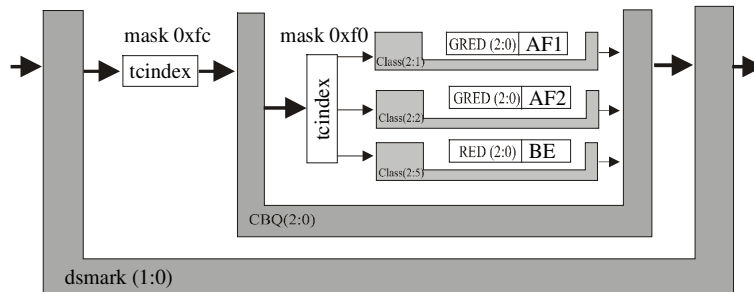

Fig. 5. Interior of core routers using dsmark, tcindex, CBQ, GRED and RED.

Packets are selected by the filters in classes (u32, fw, route, tcindex). Each class uses one of queuing disciplines (tbf, pfifo, red, gred). Schedulers used by tc are: CBQ (Class Based Queuing), PRIO (Priority queuing), HTB (Hierarchical Token Bucket). There are hierarchies of classes and filters.

Fig.5 shows the configuration of traffic control inside core router Kisscool. The queuing disciplines and classes reflect the usual Linux notation of displaying tree-based traffic control framework.

In order to use the DSCP to classify packets into the correct queue and class [20], we use two levels from a hierarchy of filters. The first level of filters is for parent dsmark(1:0) and the filter obtains DSCP from each packet. The filter is of type tcindex and it masks the TOS field with 0xfc to extract the DS filed and shift two bits to the right to get the DSCP. The second level of filters is for CBQ(2:0) (Class Based Queueing) scheduler and extracts the second digit from the classid of the packets. After masking with 0xf0 and shifting 4 bits to the right, the handle now corresponds to the correct class under the CBQ scheduler.

Finally, the last level is internal in the GRED (Generalized RED) implementation as it masks the packets with 0xf to extract the third digit of the classid and put them into their correct virtual queues under GRED [21].

The available bandwidth C at the output interface of the core router is divided by CBQ, which uses three classes with id: 2:1, 2:2 and 2:5. To avoid borrowing bandwidth from the ancestors and to block the bandwidth sharing of the same parent, we

set the following parameters: isolated and bounded [16]. We need to set this parameters to control the sharing bandwidth with our policy sharing rules. Buffers associated to AF1 and AF2 in the core router implement the GRED mechanism with two drop precedence class (AF1.1 and AF1.2 for AF1, AF2.1 and AF2.2 for AF2).

6 Experimental Results

The proposed services are AF1 and AF2 with a positive guaranteed bandwidth and BE with a zero guaranteed bandwidth. We focus on the over-provisioning case. The link bandwidth between Kisscool and Kitkat is set to C = 6Mbps. In this section, we validate the scenarios studied in Section 4, where our mechanism is compared to a static bandwidth allocation scheme. We suppose that three customers located in the hosts at the right-hand side in Fig.4 ask the network for some guaranteed bandwidth. Table 1 gives an example on how much bandwidth each customer may reserve. In this example, the maximum rate at which IN packets will be injected into the network is equal to 2Mbps per AF class. We consider that before the activation of our mechanism, the rates in the CBQ buffer are set to 2Mbps (C/3).

Table 1. Guaranteed bandwidth

PC Name	AF1.1	AF2.1
Galak	650Kbit/s	550Kbit/s
Mnm	450Kbit/s	700Kbit/s
Raider	900Kbit/s	750Kbit/s
Total bw:	2000Kbit/s	2000Kbit/s

To simplify the experiment, we suppose that the traffic of each one of the three classes is generated by the same machine, e.g., Galak generates AF1 traffic, Mnm generates AF2 traffic, and Raider generates BE traffic. Then, if the traffic of a AF class is less than the guaranteed bandwidth, all packets of that class will be marked as IN-profile. The traffic of a AF class contains OUT-profile packets when its rate exceeds the guaranteed bandwidth for that class.

Figure 6 shows how our algorithm reacts to the unfairness in the distribution of the excess bandwidth (Section 4.1). We use UDP to generate the AF1 traffic (4Mbps) and multiple long-lived TCP flows to generate the AF2 and BE traffic. The AF2 traffic has only OUT-profile packets, which is realized by setting the guaranteed bandwidth of the AF2 class to 0.

The guaranteed profile for AF1 is $R_{AF1.1}$= 2Mbps, but the router receives more than this amount (it receives 4Mbps). After one iteration we observe that AF1 has R_{AF1}=3.33Mbps, AF2 has R_{AF2}=1.33Mbps and BE has R_{BE}= 1.33Mbps. AF1 realizes its guaranteed bandwidth $R_{AF1.1}$= 2Mbps, and is then satisfied. The AF2 class is penalized because it does not ask for any bandwidth in this scenario, so it gets approximately the same throughput as BE.
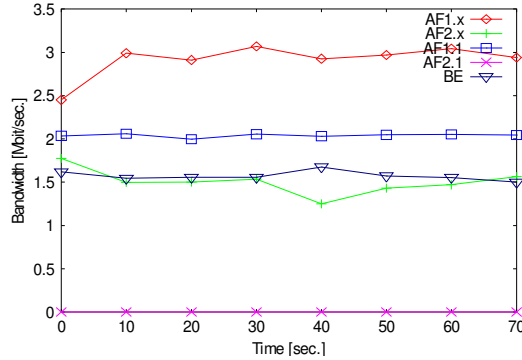
Fig. 6. Unfairness in the distribution of excess bandwidth (static case)

Table 2. Delay in AF1.1 service in the distribution of excess bandwidth case

	Min. [ms]	Avg. [ms]	Max.[ms]
Without alg.	2.108	23.799	68.891
With alg.	2.023	18.345	56.675

Table 2 shows the impact of our mechanism on the delay perceived by packets of class AF1. Our mechanism improves the quality perceived by this class by giving it a fair share of the excess bandwidth. This additional bandwidth is translated into a smaller end-to-end delay for packets of this class.

Figure 7 shows another case where one of the services (AF2) has low traffic, less than the guaranteed bandwidth, and another service (AF1) has more traffic than is guaranteed (Section 4.2.1). In this case, AF1 has UDP flows and AF2 and BE have long-lived TCP flows. The bandwidth guaranteed by the network to AF1 is set to 3Mbps, instead of 2Mbps as before (this follows a change in the contract according to a new agreement between customer and ISP). As expected, after two iterations AF1 gets R_{AF1}= 3.8Mbps, AF2 gets R_{AF2}= 1.33 and BE gets R_{BE}= 0.8Mbps. The priority traffic from each AF service class gets through the network. The two AF classes realize their desired rates. The BE service gets the smallest amount of bandwidth, since it has zero guarantee. In this case, the delay measured for AF2 has the same value with/without our algorithm (min.=1.772ms / avg.=19.708ms / max.=57.876ms).

The case when all service reservations are not satisfied is presented in Fig.8. We use for AF1 class UDP flows, and for AF2 and BE classes long-lived TCP flows. AF1 and AF2 classes ask the network for 2.5Mbps each (Section 4.2.2). In this case, we observe that our algorithm stabilizes the rates of the CBQ buffer after 20s.
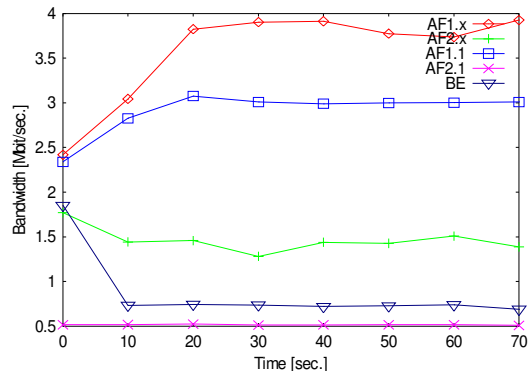
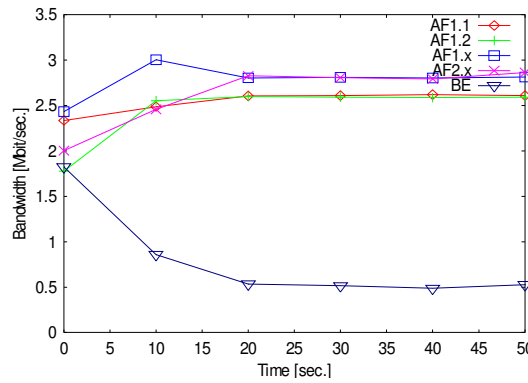**Fig. 7.** One service reservation is not satisfied, one is satisfied (static case).

Fig. 8. All service reservations are not satisfied (static case).

Fig.9 shows how fast our algorithm should run. The speed of the algorithm is determined by the averaging interval. Every averaging interval, our algorithm passes once by the loop `while` described in Section 3. If it runs faster than 5 seconds, the system oscillates. We found the optimum value of the averaging interval to be 10 seconds, as shown in Fig.10. For this experiment, we use long-lived TCP flows for all the services. The dynamics of TCP congestion control is the reason for these oscillations of the system, when the averaging interval is small. One should expect that the instability of the system for small averaging interval does not exist in case of constant rate non-reactive UDP flows.

7 Conclusions and Future Work

Our mechanism is easy to implement and does not require any particular signaling. The bandwidth allocation is taken according to local decision rules. It ensures that SLAs are respected and allows at the same time an efficient utilization of network resources. It is flexible because it can use other division rules for excess bandwidth.

Each ISP can define his own rules. The future work will be to test and to compare multiple excess bandwidth sharing rules. Another future work will be to test our mechanism with real applications like FTP, RealPlayer, and Web generators. In this paper, we only focused on constant rate UDP traffic and long-lived TCP flows.
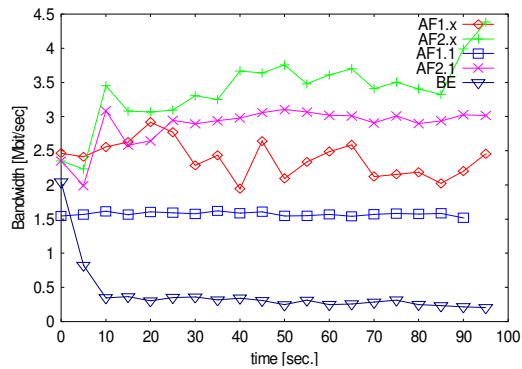

Fig. 9. The algorithm runs with a period of 5 seconds. In this case the TCP flows from AF services have no time to adapt their windows. So, this creates oscillations.
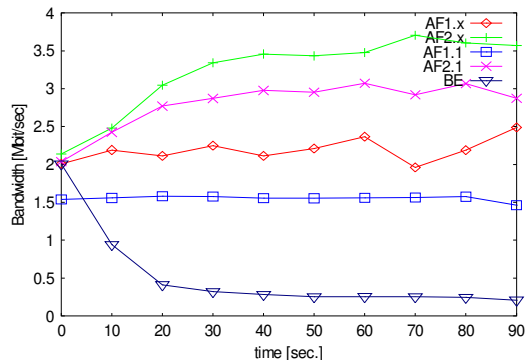

Fig. 10. The algorithm runs with a period of 10 seconds. In this case the TCP flows from AF services have time to adapt their windows. So, there are no oscillations.

References

1. R.J.Gibbens, S.K.Sargood, F.P.Kelly: An Approach to Service Level Agreements for IP networks with Differentiated Services, article submitted to Royal Engineering Society, January (2000)
2. D. Grossman: New Terminology and Clarifications for Diffserv, RFC 3260, April (2002)
3. A. Asgari, P. Trimintzios: A Monitoring and Measurement Architecture for Traffic Engineered IP Networks, IEEE/IFIP/IEE IST2001, Teheran, Iran, September (2001)

4. Jean Warland, G. Kesielis: Effective bandwidth for multiclass Markov fluids and other ATM sources, IEEE/ACM Tran. Networking, Vol.1, (1993) 424-428

5. H.G. Perros, K.M.Elsayed: Call Admission Control Schemes: A Review, IEEE Communications Mag., Vol. 34, No. 11, November (1996), 82-91.

6. S. Blake, D. Black, M. Carlson: An Architecture for Differentiated Services, RFC 2475, December (1998)

7. J. Wroclawski: The use of RSVP with IETF Integrated Services, RFC 2210, September (1997)

8. J. Heinanen, F. Baker, W. Weiss, J. Wroclawski: An Assured Forwarding PHB Group, RFC 2597, June (1999)

9. K. Nichols, V. Jacobson, L. Zhang: A Two-bit Differentiated Services Architecture for the Internet, April (1999)

10. V. Jacobson, K. Nichols, K. Poduri: An Expedited Forwarding PHB, RFC 2598, June (1999)

11. David D. Clark, Wenjia Fang: Explicit Allocation of Best-Effort Packet Delivery Servicc, IEEE/ACM Transactions on Networking, Vol. 6. No. 4, August (1998) 415–438

12. S. Floyd V. Jacobson: Link-sharing and Resource Management Models for Packet Networks, IEEE/ACM Transactions on Networking, Vol.3, No.4 August (1995)

13. L. Massoulie, J. Roberts: Bandwidth sharing: objectives and algorithms, IEEE Infocom'99, New York, March (1999)

14. Veselin Rakocevic, John M. Griffiths: Physical Separation of Bandwidth Resources for Differentiated TCP/IP Traffic, IEEE Infocom'99, New York, March (1999)

15. Ajay Tirumala, Feng Qin, Jon Dugan, Jim Ferguson: Iperf - a tool to measure maximum TCP bandwidth, allowing the tuning of various parameters and UDP characteristics, http://dast.nlanr.net/Projects/Iperf/

16. National Institute of Standards and Technology: NIST Net emulator, http://www.antd.nist.gov/nistnet

17. Paul Herman: tcpstat-Network Interface Statistics, http://www.frenchfries.net/paul/projects.html

18. Bert Hubert, Gregory Maxwell, Linux Advanced Routing & Traffic Control HOWTO, v0.9.0, January 10, (2002)

19. Bert Hubert, Gregory Maxwell, Stef Coene: Linux Advanced Routing&Traffic Control, http://lartc.org

20. Rui Pedro de Magalhaes Claro Prior: Qualidade de Servico em Redes des Comutacao de Pacotes, Faculdade de Engeharia Da Universidade Do Porto, March (2001)

21. W. Almesberger: Linux Traffic Control – Implemetation Overview, EPFL ICA Swiss, February (2001)

22. Randal L. Schwartz: Learning Perl, O'Reilly&Associates Inc. (1994)

23. Jerry Peek, Tim O'Reilly, Mike Loukides: Unix Power Tools, O'Reilly&Associates Inc. (1994)