A Linux-Based Testbed for Multicast Sessions Set-up in Diff-Serv Networks*

Elena Pagani, Matteo Pelati, and Gian Paolo Rossi

Computer Science Dept., Università degli Studi di Milano
via Comelico 39, I-20135 Milano, Italy
pagani@dsi.unimi.it, matteo@dolce.it, rossi@dsi.unimi.it

Abstract. In this work, we describe the implementation of an architecture to perform admission control and traffic management for multicast sessions in Diff-Serv networks. The *Bandwidth Broker* functionalities are carried out by the *Call Admission Multicast Protocol* (CAMP). CAMP performs the set-up of a multicast RTP session. It supports dynamic changes in the group membership. The implementation has been performed in a testbed network based on the Linux platform; we discuss the measurement results obtained by performing experiments with the testbed.

1 Introduction

In the last few years, many distributed applications have been deployed, that require some sort of *Quality-of-Service* (QoS) to the network protocols for the transmission of their data, e.g. in terms of available bandwidth or limited end-to-end delay. Currently, those applications can only exploit the *best effort* service supported by the IP protocol, which is unable to provide adequate QoS guarantees. Many research efforts are carried out to design novel network protocols and devices that are *QoS-aware*, that is, that can appropriately manage different classes of traffic. The Diff-Serv framework [1] proposed by IETF is scalable and can support QoS with only limited changes to the current Internet structure, by charging the most part of the control overhead on the domains boundary routers. The Diff-Serv model includes *Bandwidth Broker* (BB) agents [2] that perform the functionalities of admission control, resource reservation and system configuration. A BB exists for each administrative domain; the BBs cooperate to support inter-domain sessions. The two approaches proposed in the literature to perform call admission control are either the *passive* measurement policy or the *active* measurement policy [3]. The active measurement approach allows a more accurate estimate of the resource availability than the passive approach, and it is more lightweight because routers are stateless. The solutions for admission control so far proposed in the literature [3,4,5,6] only consider unicast sessions,

* This work was supported by the MURST under Contract no.MM09265173 "Techniques for end-to-end Quality-of-Service control in multi-domain IP networks".

although many applications are multicast in nature. On the other hand, multicast support in the Diff-Serv model is difficult [7]. Moreover, the behaviours of the existing solutions in realistic environments are often not sufficiently analyzed.

In this work, we describe the implementation of an architecture to perform admission control and traffic management for multicast sessions in Diff-Serv networks. The admission control service is provided by the *Call Admission Multicast Protocol* (CAMP), that carries out the *Bandwidth Broker* functionalities. CAMP performs the set-up of a multicast RTP session, and it supports dynamic changes in the group membership. The system exploits an active measurement approach with dropping of the `probe` packets as the congestion signal. The implementation has been performed in a testbed network based on the Linux platform; we discuss the measurement results obtained by performing experiments with the testbed.

The paper is structured as follows: in Section 2, we describe the testbed architecture, and we give an overview of the admission control mechanism. In Section 3, we discuss in detail the testbed implementation on the Linux platform. In Section 4, we present the experimental results. Section 5 concludes the work.

2 Testbed Architecture

In figure 1, we show the architecture of the implemented system. At the highest level, the most part of the modules are implemented in both the source and the destinations; the shadowed modules only exist at the destinations. The modules at the network and the data link layers are instantiated in all the routers belonging to the multicast routing tree. At the server side, the traffic generator can either produce a dummy data stream having a Constant Bit Rate, or it can read the data (1) from a MPEG file. The traffic generator is linked (2) to the RTP/RTCP library [10], and the admission control module (CAMP). The RTP library is used to support the appropriate playback of the data traffic at the destinations (10). The RTCP library is exploited to exchange information about the profile of the traffic generated by the source, and the quality of the traffic received at the destinations. It may use either UDP or TCP; in section 2.1 we specify how the transport service is chosen. At the receiver side status information is maintained about the profile of the traffic generated by the source, to decide about the session acceptance. This information is available via the RTCP source report; it is used by the admission control module. To support multicast communications, we use IGMP [11] as a membership service, and PIM-SM [12] as a multicast routing protocol. They cooperate to maintain the multicast routing table (Multicast Forwarding Cache, MFC) (4). To notice the dynamic changes in the multicast group membership, thus performing the appropriate admission control over the new multicast tree branches, the CAMP part at the kernel level implements a hook (5) to PIM-SM. Both data (7) and probing (6) multicast traffic is routed by PIM-SM. At the data link layer, packets are managed according to a *priority* packet scheduling policy: the data packets belonging to sessions requiring QoS have the highest priority. The priority level is determined by the

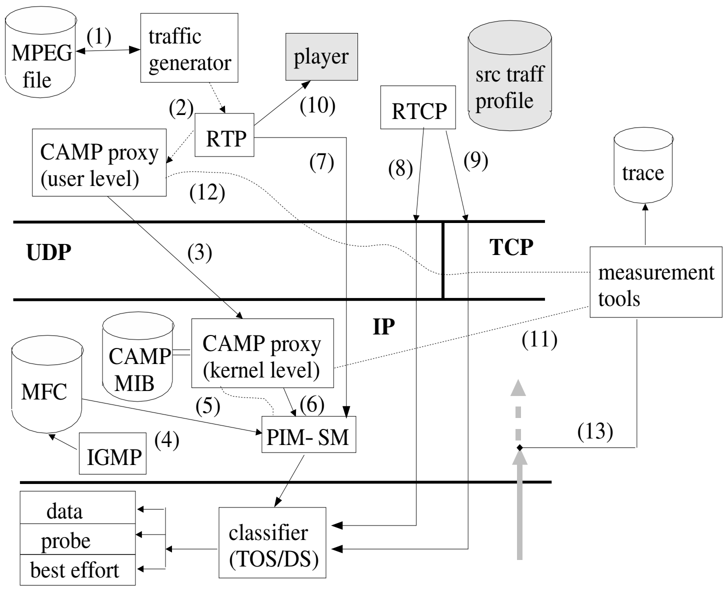


Fig. 1. Functional architecture of the multicast testbed.

value of the TOS/DS byte carried in the IP header [13]. We deployed ad hoc measurement tools that trace the traffic generation and delivery. At the source side, they are linked to the CAMP part at the kernel level, to monitor the packet transmission (11). At the destination side, they are linked to the CAMP part at the user level (12), that receives the packets; moreover, they trace the packet arrivals immediately above the network interface (13).

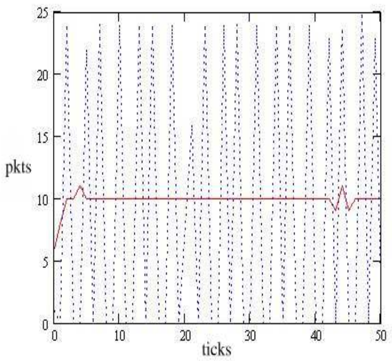
2.1 The CAMP Protocol

In this section, we outline the admission control mechanism; for more details, the interested readers may refer to [8,9]. The admission control procedure is *receiver-driven*: each receiver autonomously decides whether it can receive the data traffic with the adequate QoS level or not, that is, whether to participate in the multicast session or not. The *Call Admission Multicast Protocol* (CAMP) operates on-demand; it performs the admission control for multicast sessions requiring bandwidth guarantees, thus supporting the *Premium Service* (EF PHB [14]). CAMP adopts an active measurement approach, with dropping of the `probe` packets as the congestion signal. The `probe` packets are marked with a lower priority than the QoS data packets, but a higher priority than the best effort packets. This way, they can drain the available bandwidth at the expenses of the best effort traffic, while they do not affect the established QoS sessions. For the sake of simplicity, throughout this work we hypothesize that all the recipients have the same QoS requirements, and they receive the same set of microflows addressed to the group. Hence, they accept to participate in the session only if they can receive the `probe`s with the adequate bit rate; otherwise, they leave the group and prune from the multicast tree. We discuss this assumption in section 5. We deal with dynamic changes of the group membership by exploiting a *proxy* approach. CAMP proxies can be instantiated both in the source host and in the in-tree nodes. A proxy is created in a node as soon as a new downstream interface appears for a multicast tree. The proxy records in the CAMP MIB the status information concerning the downstream interfaces. Initially, all the downstream interfaces are in the *probing* state. By default, the CAMP proxy remarks all the packets received from the upstream interface in the multicast tree as `probe` packets, before forwarding them to the probing interface, while the packets are forwarded unchanged to the already existing interfaces. The recipient compares the QoS of the received probing traffic with the recorded source traffic profile, and decides whether to accept the session or not. In the latter case, it prunes from the tree. In both cases, it sends respectively an accepting or a refusing RTCP report to the upstream CAMP proxy. If the proxy receives an accepting report for a downstream interface, it stops the packet remarking for that interface, and it possibly forwards the acceptance to its upstream proxy if it is receiving remarked packets in turn. The interface state is changed from *probing* to *data*. When all the *probing* interfaces of a proxy either disappear as a consequence of a pruning recipient or are marked as *data* interfaces, the proxy turns off and all the status information it recorded in the CAMP MIB can be deleted. The proxy approach allows to perform the admission control for dynamically joining destinations, and to immediately forward the data to recipients joining during the transmission without negatively affecting other established sessions. It also allows to hide the membership changes to the source: in the initialization phase, the CAMP source generates `probe` packets until an accepting report is received. At that point, the source switches *without discontinuity* to the transmission of the data packets, while the proxies terminate the admission control procedure for the remaining destinations. To guarantee the termination of the session set-up phase, the decision reports cannot be lost. While the usual RTCP reports can be sent exploiting the UDP transport service (link 8 in fig.1), the decision RTCP reports are sent via TCP connections (link 9 in fig.1). The decision is included in an application-dependent field. In order to expedite the switch to the data transmission phase, the decision reports are generated using the *early RTCP feedback* mechanism [15].

3 Testbed Implementation

We deployed a prototype version of the described architecture in the framework of the NS-2 simulation package: the obtained performance measures were promising [8]. To evaluate the behaviours of our approach in a realistic environment, we implemented the architecture in a testbed network based on the Linux platform. In this section we supply the technical details on the implementation of the system modules shown in figure 1. As the IGMP and PIM-SM implementations, we used those available with the Linux kernel.

TRAFFIC GENERATOR. In order to deploy our streaming server, we evaluated the performance of two existing traffic generators: MGEN [16] and RUDE [17]. Our primary concern was the time granularity at which network packets were sent over the network. The obtained results suggested us to implement our streaming server using the same technique adopted by RUDE. We performed a test by generating bursts of 10 packets, of size 1024B each, every 10 ms, which is the tick value, that is, the minimum time granularity achievable on an x86 platform. As it is shown in Figure 2(a), we are able to transmit data with a packet interval of about 10 ms (continuous line). Hence, the performance measured at the clients (section 4) is not affected by burstiness in the generated traffic. This



(a)

```
<TSET ID="OS1">
   <GROUP VALUE="224.10.10.10"/>
   <DESTPORT VALUE="8000"/>
   <NAME VALUE="OS Course test transmission"/>
   <RTCPBW VALUE="5"/>
   <TTL VALUE="14"/>
   <STREAM ID="VIDEO" TYPE="MASTER">
      <NAME VALUE="Video feed"/>
      <PORTBASE VALUE="10000"/>
      <RELEVANCE VALUE="1"/>
      <BITRATE VALUE="1700"/>
      <PACKETSIZE VALUE="1024"/>
      <FILENAME VALUE="/home/videos/os1.mpeg"/>
      <FORMAT VALUE="mpeg"/>
   </STREAM>
   <STREAM ID="AUDIO" TYPE="CHILD">
      <GROUP VALUE="224.10.10.11"/>
      <NAME VALUE="Audio feed"/>
      <PORTBASE VALUE="11000"/>
      <RELEVANCE VALUE="2"/>
      <BITRATE VALUE="128"/>
      <PACKETSIZE VALUE="1024"/>
      <FILENAME VALUE="/home/videos/os1audio.mp3"/>
      <FORMAT VALUE="mp3"/>
   </STREAM>
</TSET>
```

(b)

Fig. 2. (a) Profiles of the packet flow produced by the traffic generator and by MGEN. (b) Example of the parameters specified in the sender-side configuration file.

granularity was obtained by using a busy-waiting technique instead of timers, and by scheduling every send operation using the `gettimeofday()` function. By contrast, the same traffic load generated by MGEN yields the transmission of bursts of 25 packets every 30 ms (dashed line), thus showing a sawtooth behaviour. The streaming server integrates the RTP/RTCP library and interfaces directly to the transport layer via sockets. The server can generate multiple sessions concurrently. Each session is composed by one or more streams, each one of them sent to a specific multicast group. With this approach, multiple channels can be included in a session (e.g., audio, video, slides), and compatibility is offered with future layered video codecs. The streams that are to be generated are described in a configuration file, which is interpreted by the traffic generator. A server-side configuration file is shown in figure 2(b): in that example, a two-streams transmission is set-up, involving a master stream of 1700 Kbps and a child audio stream of 128 Kbps. In the example, data are read from files. The generator can also produce synthetic CBR traffic.

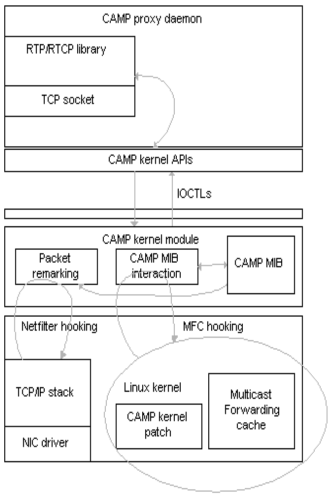
RTP/RTCP. As the implementation of RTP/RTCP, we use the `jrtplib` library [18], appropriately modified according to our needs. In particular, we modified the library so that the reports can be sent either multicast or unicast. Multicast reports are sent accordingly to the standard specification, that is, they are sent via UDP (datagram sockets) to all the multicast group members (link 8 in fig.1). The unicast addressing is used for the decision reports. Unicast reports are sent upon request of the CAMP proxy, and their content is specified by the proxy. The unicast decision reports are sent using TCP (stream sockets, link 9 in fig.1); each recipient sends the decision to the upstream proxy. A proxy receiving a unicast report processes it locally by exploiting the RTCP library, and it may as well update the report content and forward it to its own upstream proxy, if one exists. A further modification concerns the RTCP report content: the sender report contains, as application-dependent fields, the generated bit rate and the number of children streams, as well as the group and relevance of each child stream. This allows a client to automatically probe for children streams whenever additional bandwidth is available on the network. `jrtplib` is linked to the traffic generator and the CAMP code. The packets created by the traffic generator are labeled with the RTP information by handing them to the appropriate library procedure, and then they are possibily processed by the CAMP procedures. Then, they are sent using `datagram` sockets (link 7 in fig.1). At the receiver side, RTP forwards the data to the player at the appropriate rate via a `fifo` (link 10 in fig.1).

CAMP. We implemented three distinct versions of CAMP: the CAMP server, the CAMP client and the CAMP proxy. They have been implemented in C++; their procedures are executed by independent threads. The server runs at the traffic source, while the client runs in each group member. The server involves both the traffic generator and the CAMP proxy, detailed below.

The CAMP client only records the multicast group it belongs to, and the address of the upstream proxy throughout the set-up phase. The client directly reads data from the network via datagram sockets and integrates with the RTP/RTCP library. The client is able to receive multiple RTP streams. RTP and RTCP data are automatically de-multiplexed by the `jrtplib` library: RTP streams are then forwarded to an external program (an `mpeg` player, for example) using `fifos`, while RTCP information is collected by the CAMP module of the client to create stream profiles. These profiles store information such as the expected and actual reception rate, jitter, delay, and other QoS metrics. The client uses this information whenever a probing session must be accepted or refused. In order to accept a probing session, the client invokes the `jrtplib` to create a new RTCP report and sends it to the upstream proxy using TCP. In addition, once a session has been accepted, if additional children streams exist, the client will try to join them in order of their relevance and, if enough bandwidth is available, accept them as well.

The CAMP proxy is executed in the in-tree routers. It maintains the status information of the currently controlled multicast output interfaces, recorded in the CAMP MIB shown in figure 1, and performs the packet remarking. Moreover, the CAMP proxy substitutes the IP source address of every remarked multicast packet with the address of its outgoing interface, so that the downstream recipient (or downstream proxy) can appropriately address the decision reports.

In figure 3(a), we show the modules implemented for the CAMP proxy; in figure 3(b), we represent the operations performed by the proxy. The modules




```
when (received a RTCP pkt requesting
      probe-to-data switch) do
{
   if (receiving probe from parent proxy)
   then forward the RTCP pkt to parent proxy;
   oif <- interface on which RTCP pkt arrived;
      /* lookup in the Kernel NICs table */
   if (sending probe to oif)
   then
      in the CAMP MIB: oif status <- "data";
}
```

(a) (b)

Fig. 3. (a) Layout of the CAMP proxy. (b) Pseudo-code of the CAMP proxy operations.

communicate via `ioctl` (link 3 in fig.1): the CAMP user space daemon handles the RTCP unicast messages, while the kernel level module manages both the reports and the data packets. We deployed a kernel patch that implements a hooking mechanism to monitor the MFC and receive a notification whenever a downstream multicast interface is added or removed; the interested readers may refer to [19] for technical details about the kernel hook implementation. Upon the reception of a decision report, the daemon must decide whether the report must be further forwarded upstream or not. If the packets received for the multicast group indicated in the report are `probes`, then the report is forwarded upstream after the appropriate update of the destination address. The daemon cooperates with the kernel module to retrieve the network interface from which the report has been received. The information for the report management are recorded in the CAMP MIB, which is composed by entries having the structure shown in figure 4. The NIC to which the report is addressed is compared against the `dev` field of the Camp MIB. For the matching entry, the `mark` field is changed from `probe` to `data`. To support the appropriate management of the decision reports,

```
typedef struct _PROBING_TRAFFIC_INFO {
    unsigned char mark; // downstream interface status
    unsigned long mcastAddress; // multicast IP address
    unsigned long srcIpAddress; // downstream interface IP address
    struct net_device *dev; // kernel list for attached NICs
} PROBING_TRAFFIC_INFO, *PPROBING_TRAFFIC_INFO;
```

Fig. 4. Element of the Camp MIB.

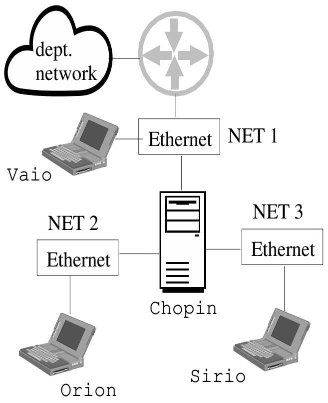
we exploit the functionalities of `netfilter` [20]: the Linux 2.4.x/2.5.x firewalling subsystem for packet filtering and processing. `netfilter` is also used to remark the packets on behalf of the proxy (link 6 in fig.1).

TRAFFIC CONTROL. In order to effectively test our software modules, we had to simulate congested and low-speed network segments. To do so, the Linux TC tool [21] has been used. All the outgoing interfaces on the Linux routers were limited to a maximum bandwidth of 2.4 Mbps. In addition, we configured three different traffic type classes. The first one for `data` traffic is assigned the maximum priority and is identified by a DS field equal to 1E. The second one for `probe` traffic is assigned a medium priority and is identified by a DS field equal to 1C. The third class is for the best effort traffic and has the lowest priority.

MEASUREMENT TOOLS. We implemented two ad hoc measurement tools to control the profile of the generated packet flow at the server side and to test the reception quality at the client side. The former tool is a user level thread implemented into the client software. Instead of delivering all RTP packets to the external player program, the packets are just logged to an external file. The log file contains remote SSRCs, reception rate, jitter and intra-packet delay. The latter tool has been written to be integrated into the kernel. It is a kernel module that hooks all outgoing packets addressed to a particular multicast address. We used it to trace the packet generation process. We have been able to achieve a great precision (10 ms) by monitoring the system wide clock counter (the `jiffies` global variable of the Linux kernel) whenever network packets were about to hit the network wire. The trace files produced by the measurement tools can be used as input to graphic tools to plot the performance measures.

4 Experimental Results

Our test network is based on four machines running RedHat Linux with kernel 2.4.18. In figure 5(a), we show the layout of our network, while in figure 5(b), we report the network configurations of the hosts. `Chopin` acts as the router among



(a)

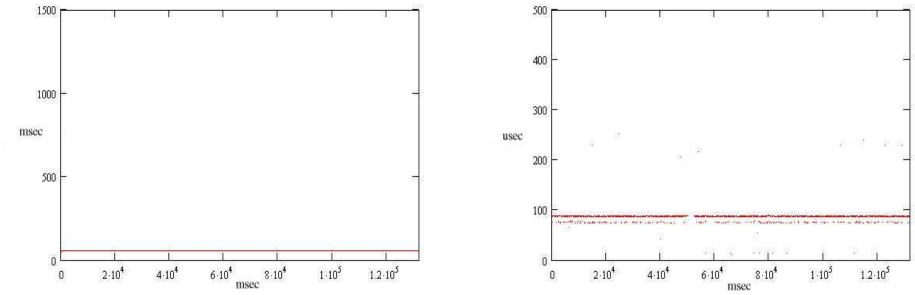
host	network interface(s)	gateway
Vaio	Eth0: 159.149.189.38/26	159.149.189.36
Sirio	Eth0: 159.149.189.98/27	159.149.189.126
Orion	Eth0: 159.149.189.65/27	159.149.189.94
Chopin	Eth0: 159.149.189.36/26	
	Eth1: 159.149.189.94/27	
	Eth2: 159.149.189.126/27	

(b)

Fig. 5. (a) Layout of the test network. (b) Host configuration.

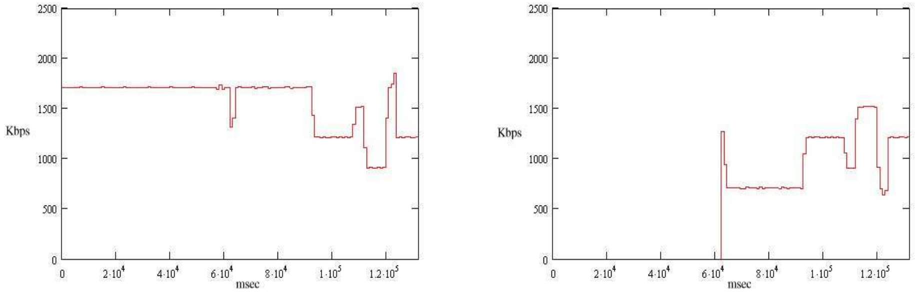
three different subnets. In order to perform multicast routing, PIMd has been installed on `Chopin` and multicast routing support has been enabled in the kernel. `Vaio` has been primarily used to generate traffic. `Sirio` and `Orion` have been used to receive the incoming data generated by `Vaio`. The client running on those two hosts has been deployed ad hoc to support CAMP. We performed experiments with the clients joining at different times during the data transmission: the proxy approach worked properly.

In all the discussed experiments, we generated CBR data traffic with packet size 1024B. As the performance indexes we considered the bit rate received at the destinations, the intra-packet delay, estimated by the measurement tools as explained in section 3, and the jitter. The jitter is the variance of the packet inter-arrival time; it is a measure of the regularity with which the flow is received at the destinations. The jitter is evaluated by `jrtplib` according to the RTP standard. We tried to measure the overhead due to the packet processing performed by the CAMP proxy: the overhead is negligible, comparable to that involved by a firewall. In figure 6, we show the jitter and the delay measured by performing an experiment with one 1024 Kbps flow sent over a link of 2.4 Mbps, without any packet prioritization. The measures are reported against the measurement time; all the described experiments last around 120 sec. This experiment aims at validating our system. The received bit rate equals the sending rate, while the jitter is negligible and the intra-packet delay behaviour confirms that the traffic is generated with a regular profile. We initially performed experiments without enabling the admission control procedure, nor the packet prioritization. All the packets fairly share the available bandwidth, according to a stochastic fair queuing packet scheduling policy. We injected two flows of 1.7 Mbps over a channel having 2.4 Mbps capacity. The second flow starts around 60 sec. after the first flow. As it can be observed in figure 7, the average rate obtained by each flow is roughly half of the link capacity. As queuing delays cannot be predicted,



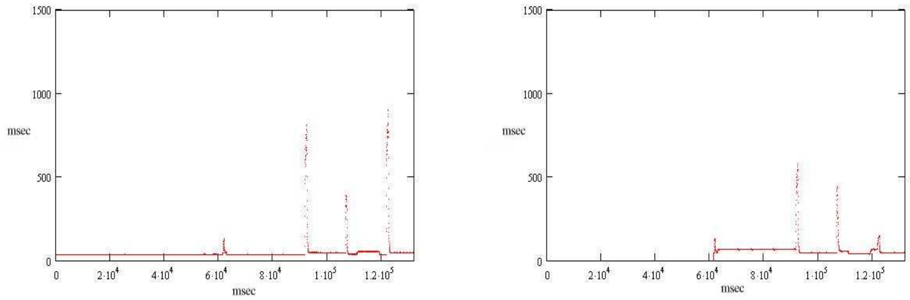
(a) (b)

Fig. 6. (a) Jitter and (b) delay of a 1024 Kbps best effort flow sent over an unloded network vs. measurement time.



(a) (b)

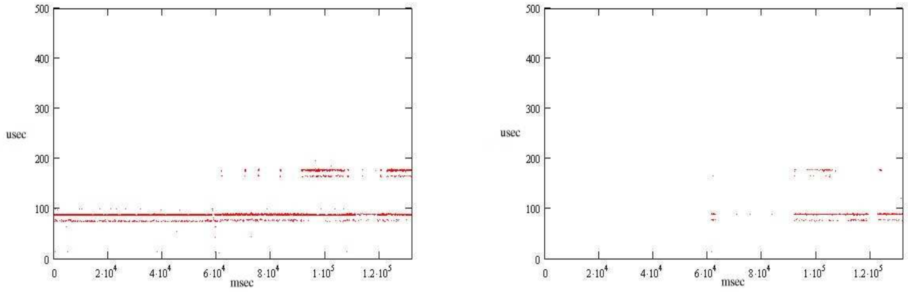
Fig. 7. Received bit rate for (a) the first and (b) the second flow, vs. measurement time.



(a) (b)

Fig. 8. Instantaneous jitter for (a) the first and (b) the second flow, vs. measurement time.

the traffic behaviour is not regular, and this negatively affects the measured jitter (figure 8) and intra-packet delay (figure 9). Under this system configuration, QoS sessions cannot receive any guarantee in terms of available bandwidth, jitter or delay. We tried to evaluate to what extent the traffic prioritization succeeds in protecting the QoS traffic at the expenses of the best effort traffic. We generated two flows of 1.7 Mbps; the former one is classified as best effort traffic, while the packets of the latter one are marked as `data` packets. As expected, the QoS data is received at the correct rate, and it shows a negligible jitter and a constant delay, comparable with the measures shown in figure 6. By contrast, the best effort traffic can only use the remaining bandwidth (figure 10(a)). Moreover, the best effort packets suffer transient and unpredictable queuing delays. As a



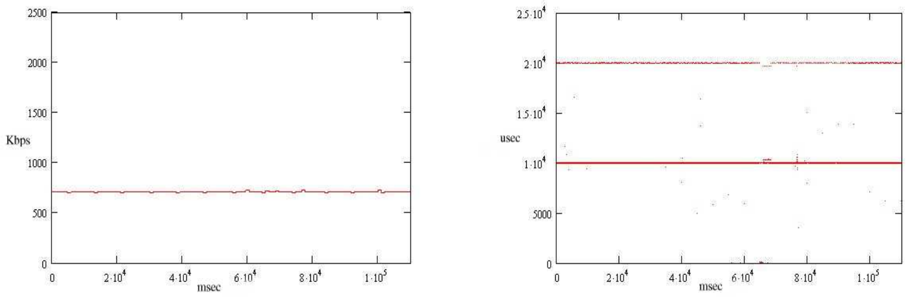
(a) (b)

Fig. 9. Instantaneous intra-packet delay for (a) the first and (b) the second flow, vs. measurement time.

consequence, its profile at the recipient is bursty (figure 10(b)). The jitter shows a sawtooth behaviour. Similar results are achieved starting the QoS flow 60 sec. after the best effort flow: the QoS packets gain the required bandwidth at the expenses of the best effort traffic, whose performance degrade.

The priority packet scheduling allows to protect the QoS traffic from the best effort traffic, but, it cannot guarantee the required QoS in the case the network is congested by QoS sessions. We performed an experiment by generating two 1.7 Mbps flows both having a `probe` priority; the flows start at the same time. Although our network was underutilized during the experiments, some background traffic was possibly present. With the packet prioritization, the effects of that background traffic are negligible and the two flows have a regular pattern (figure 11). But, they compete for the available bandwidth, that is fairly shared among them, so that the recipients observe a degradation in the quality of both flows. As a consequence of the contention for the use of the available resources, the packets of both flows suffer queuing delay, that in our measures is around 10 msec, yielding a highly variable jitter.

A priority packet scheduling policy alone cannot guarantee that the multimedia sessions receive the needed amount of resources. To this purpose, the network congestion must be avoided by allowing that in the network is only injected the traffic that can be appropriately handled. We generated two flows with bit rate 1.7 Mbps as before, this time activating the CAMP mechanisms. The second flow starts 45 sec. after the first one. The probing phase lasts about



(*a*) (*b*)

Fig. 10. (*a*) Bandwidth and (*b*) instantaneous intra-packet delay of the best effort flow in the case of packet prioritization and without admission control.

30 seconds: this is a very long set-up phase with respect to the lower bound estimated in [9], but allowed us to highlight the traffic behaviours. The first flow does not compete with any other `probe` or QoS traffic, and as a consequence it succeeds to complete the admission control and the packets are transmitted as `data`, with the highest priority. When the second flow starts, the transmission of the first flow is already ongoing. Thanks to the higher priority, the traffic profile of the first flow is not negatively affected by the second flow (figure 12(*a*)). The `probe` packets of the second flow can drain at most 0.7 Mbps of available bandwidth (figure 12(*b*)); they are enqueued and suffer a queuing delay of about 10 msec. As a consequence, the recipients refuse the second session. By activating the CAMP procedures and repeating the same experiment with two flows that start contemporarily, the recipients refuse both the flows because their arrival bit rates are unacceptable. The measured performance is comparable with that shown in figure 11. Hence, the admission policy is conservative, and it can lead to poor network utilization. Anyway, in those cases, the traffic sources can wait a random time before re-trying to perform the admission control, so that a greater success probability is guaranteed.

5 Concluding Remarks

In this paper we describe the implementation of a Linux-based testbed for the set-up and management of multicast sessions requiring QoS in a Diff-Serv environments; the source code of the testbed is available [22]. The CAMP protocol has been exploited to implement the Bandwidth Broker functionalities. Experiments have been performed to evaluate the system behaviour under different

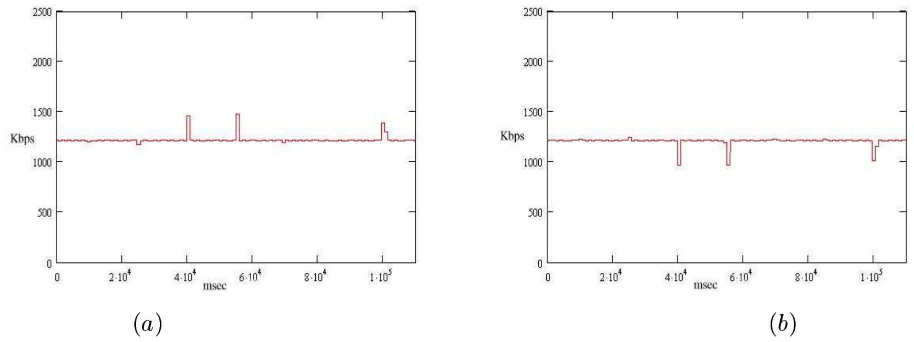


Fig. 11. Received bit rate for (a) the first and (b) the second QoS flow, vs. measurement time.

traffic and network conditions. The implemented architecture is able to provide bandwidth guarantees. The established sessions are not affected by the concurrent best effort traffic or the probing traffic for the set-up of new sessions. The admission control mechanism is effective, although possibly conservative in the case several sessions start simultaneously. The adopted approach does not impose drastical changes to the current network architecture, and charges a low processing overhead on the routers. As a consequence, it can effectively represent a first step in the deployment of the QoS-aware Internet structure. As a future work, we plan to extend our testbed with mechanisms for traffic policing, to support multiple service levels by guaranteeing the separation among different classes of traffic. Moreover, we are designing mechanisms to support heterogeneous receivers with different QoS requirements. This could be easily achieved by

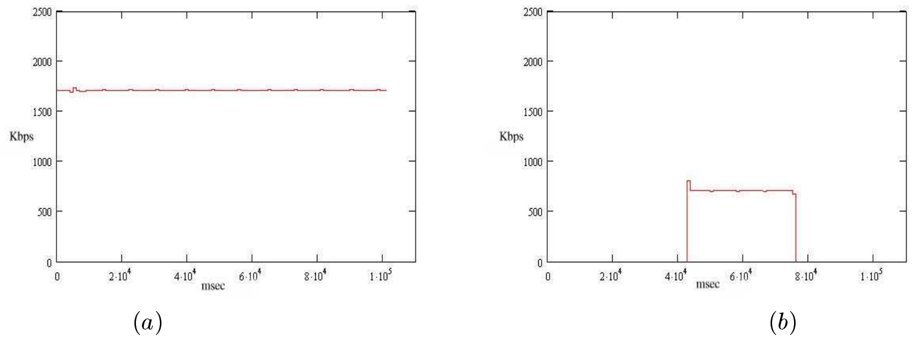


Fig. 12. Received bit rate for (a) the first and (b) the second QoS flow, vs. measurement time.

adopting layered encoding schemes for the data. So far, however, those schemes are not standardized.

References

1. Blake S., Black D., Carlson M., Davies E., Wang Z., Weiss W.: *"An Architecture for Differentiated Services"*. RFC 2475 (Dec. 1998). Work in progress.
2. Nichols K., Jacobson V., Zhang L.: *"A Two-bit Differentiated Services Architecture for the Internet"*. Internet Draft draft-nichols-diff-svc-arch-00 (Nov. 1997). Work in progress.
3. Breslau L., Knightly E., Shenker S., Stoica I., Zhang H.: *"Endpoint Admission Control: Architectural Issues and Performance"*. Proc. SIGCOMM'00 (2000) 57-69.
4. Räisänen V.: *"Measurement-Based IP Transport Resource Manager Demonstrator"*. Proc. International Conference on Networking, LNCS Vol. 2094. Springer (Jul. 2001) 127-136.
5. Lai K., Baker M.: *"Measuring Link Bandwidths Using a Deterministic Model of Packet Delay"*. Proc. SIGCOMM'00 (2000) 283-294.
6. Elek V., Karlsson G., Rönngren R.: *"Admission Control Based on End-to-End Measurements"* Proc. INFOCOM'00 (2000).
7. Bless R., Wehrle K.: *"IP Multicast in Differentiated Services Networks"*. Internet Draft draft-bless-diffserv-multicast-01 (Nov. 2000). Work in progress.
8. Pagani E., Rossi G.P.: *"Measurement-Based Admission Control for Dynamic Multicast Groups in Diff-Serv Networks"*. Proc. 2nd Intl. IFIP Networking Conference, Lecture Notes in Computer Science, Vol. 2345. Springer (May 2002) 1184-1189.
9. Pagani E., Rossi G.P.: *"Distributed Bandwidth Broker for QoS Multicast Traffic"*. Proc. 22nd IEEE Intl. Conf. on Distributed Computing Systems - ICDCS'02 (Jul. 2002) 319-326.
10. Schulzrinne H., Casner S., Frederick R., Jacobson V.: *"RTP: A Transport Protocol for Real-Time Applications"*. RFC 1889 (Jan. 1996). Work in progress.
11. Cain B., Deering S., Thyagarajan A.: *"Internet Group Management Protocol, Version 3"*. Internet Draft draft-ietf-idmr-igmp-v3-01 (Feb. 1999). Work in progress.
12. Estrin D. et al.: *"Protocol Independent Multicast - Sparse Mode (PIM-SM): Protocol Specification"*. RFC 2362 (Jun. 1998). Work in progress.
13. Nichols K., Blake S., Baker F., Black D.: *"Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers"*. Internet Draft draft-ietf-diffserv-header-04 (Oct. 1998). Work in progress.
14. Jacobson V., Nichols K., Poduri K.: *"An Expedited Forwarding PHB"*. RFC 2598 (Jun. 1999). Work in progress.
15. Wenger S., Ott J.: *"RTCP-based Feedback: Concepts and Message Timing Rules"*. Internet Draft draft-wenger-avt-rtcp-feedback-02 (Mar. 2001). Work in progress.
16. Naval Research Laboratory: *"The MGEN Toolset"*. `http://manimac.itd.nrl.navy.mil/MGEN/`
17. Laine J., Saaristo S., Prior R.: *"Real-time UDP Data Emitter (RUDE)"*. `http://rude.sourceforge.net/`
18. Schulzrinne H.: *"RTP: About RTP and the Audio-Video Transport Working Group"*. `http://www.cs.columbia.edu/~hgs/rtp/`
19. Pelati M.: *"Multicast Routing Code in the Linux Kernel"*. Linux Journal. SSC Publications (Nov. 2002) 16-21.
20. Netfilter Team: *"The netfilter/iptables project"*. `http://www.netfilter.org/`
21. *"Differentiated Services on Linux"*. `http://diffserv.sourceforge.net/`
22. `http://homes.dsi.unimi.it/~pagae/NPTLab/Software`.