A Soft Real-Time Measurement System for a Diffserv over MPLS Edge Router

Rosario G.Garroppo, Stefano Giordano, Fabio Mustacchio, Francesco Oppedisano and Gregorio Procissi

Dipartimento di Ingegneria dell'Informazione - Università di Pisa
Via Caruso 1, 56122 Pisa
Tel.: +39-050-2217511, Fax: +39-050-2217522
Email: {r.garroppo, s.giordano, g.procissi}@iet.unipi.it, {f.mustacchio, f.oppedisano}@netserv.iet.unipi.it

***Abstract*— The paper presents the architecture of a soft real time measurement system designed to reside into a DiffServ over MPLS Linux based router. The goal of this research is to develop an open source software product to be used for management purposes and to be integrated in the network control plane in order to automate resource allocation and admission control functionalities. The system is designed to be flexible, configurable and modular and each module has been implemented in order to minimize the impact of the system itself on the traffic dynamics. Several issues on the realization of the software modules are addressed and discussed. The output of the system is a sequence of measurements that report the amount of per-PHB and per-LSP traffic offered to the router over a configurable time window. Traffic data are sampled according to a customizable sampling frequency. The system has proven to be reliable as it passed several functional tests. Preliminary experimental results have shown that the output of the system accurately captures the traffic patterns offered to the router.**

I. Introduction

Emerging services such as Voice over IP, audio/video streaming, videoconferencing and Virtual Private Networks (VPNs) with Quality of Service have more stringent QoS requirements (low latency and losses, large throughput) than traditional "best effort" applications (e-mail, http, file transfer). This fact leads to the deployment of network architectures (e.g. DiffServ) which provide basic support to QoS oriented applications. Moreover a considerable effort has been dedicated over the last years to the performance evaluation and optimization of operational IP networks: the so called *Traffic Engineering*. In this context, the need for optimal resource utilization and traffic performance entails the adoption of dynamic allocation techniques and admission control mechanisms. The focus of this paper is on the design and implementation of a soft real-time measurement system to be integrated in a Multi Access Inter Domain architecture for QoS provisioning in MPLS/DiffServ networks. The measurement system has been designed both to be used for management purposes and to be integrated into the network control plane to automate dynamic resource allocation and admission control on the basis of traffic measurements and predictions. The activities described in this paper have been developed within the FIRB TANGO [1] and VICOM [2] projects.

Although the system design was strongly influenced by the test-bed architecture proposed in both projects and by its specific requirements, its flexible design philosophy based on:

- system overall modularity provided by socket interconnection among the different modules/functionalities
- use of open source code and tools
- standard compliancy
- inter-operability with commercial routers

makes the overall product readily portable on top of different network contexts.

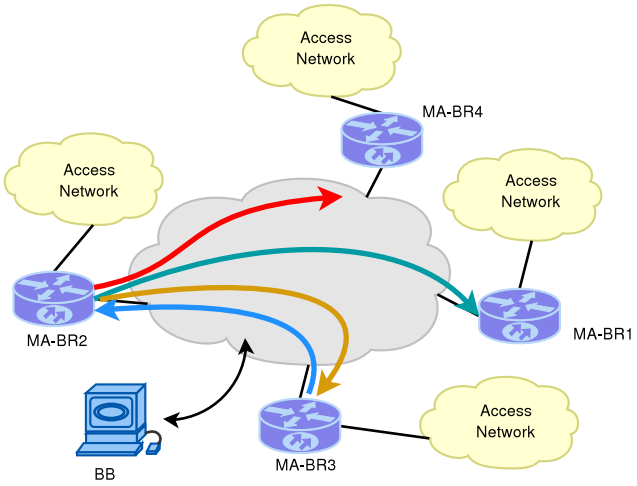

Fig. 1. MAID network architecture

The measurement system is integrated in the experimental test-bed developed under the TANGO project according to the Multi Access Inter-Domain architecture (MAID) [3]. A few comments on MAID are given next: a detailed description can be found in [3].

The MAID network architecture (see figure 1) is based onto two key elements, the Bandwidth Broker (BB) and the Multiple Access Border Router (MA-BR). The Bandwidth Broker acts as the centralized resource manager and is devoted to traffic engineering operations (e.g. path computation, LSP establishment and release), network element reconfiguration and inter-domain communications. The MA-BR, which provides the inter-working between the access network and the backbone network, merges the functionalities of an MPLS edge router and of a DiffServ border router by mapping the

traffic onto the correct pre-established LSP and handling the QoS parameters needed by each flow.

In the specific scenario, the measurement system has been designed to be integrated onto prototypal routers based on IA32 (PC) Linux OS platform. The complete set of functionalities provided by the measurement system is accessible via TCP/IP socket to enable the use of a remote centralized management system.

II. System Requirements

The above depicted network context in which the measurement system is supposed to be integrated, together with the objective of using such a system within a closed loop resource allocation mechanism, arise a number of requirements that will be shortly outlined in this section.

- **Measurements** of the traffic offered to the edge router of a DiffServ/MPLS domain taken over arbitrary and configurable time windows in the past. This information should be available at *any* time upon request.
- **Low latency** introduced by the system itself to reduce the unavoidable impact of measurements onto network performance.
- **Sampling** of network traffic with accurate timing with low sampling jitter, with no use of busy wait.
- **Timestamping** of traffic samples in order to enable time series processing, including prediction.
- **Data storage** of per-flow traffic time series in a proper database.
- **Remotization** of the measurement system in order to enable router control through a separate control network.
- **Multiple Client Support** in order to enable simultaneous operations, such as traffic monitoring, estimation and prediction.
- **Quick** delivery of information to client/s to prevent that high processing delays impact the relevance of measurements.
- **Interworking** with other router subsystems such as *routing*, *MPLS*, *RSVP-TE* daemon, etc.

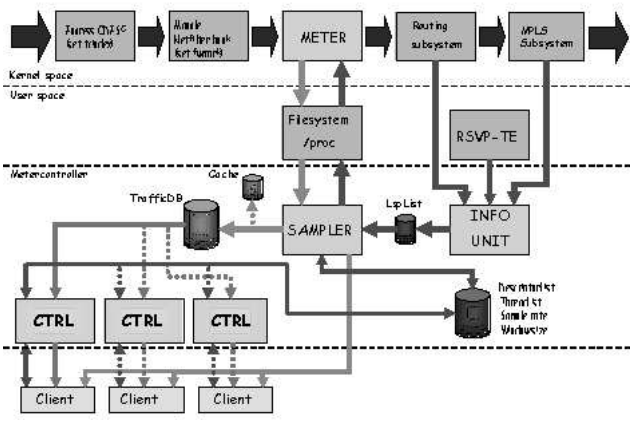

Fig. 2. Measurement system architecture

It is worth discussing a little more about the need for registering in a database the load condition of the edge router. Indeed, the system development would be much easier by just adopting a trivial measurement server which responds to queries from a remote client on the basis of the output of a suitable measurement subsystem. This approach would have reduced the implementation time at the cost of a reduced flexibility and versatility of the project which would suffer from several limitations, such as:

- *coarse accuracy of the sampling rate*, due to the latency introduced by the request network which sums up to the other many variable delay factors;
- weak capability of time series analysis/processing. Indeed, typical operations may need several minutes worth traffic data, which would not be available at the time of request;
- impossibility, for a network administrator, to have traffic load conditions available at any arbitrary times.

The use of a local database effectively overcomes the above listed issues: the price of this approach is, in turn, considerable in terms of design and development complexity.

Although the most part of the above mentioned requirements are met through the software modules presented in the present work, a few of them are to be refined and will be addressed in future works.

III. System Architecture: Modules and Functionalities

In this section, the whole system architecture will be presented with particular focus on the description of the modules, their functionalities and interworking operations.

Figure 2 shows the whole architecture of the system with the implemented software modules. In the following, the operation of all modules will be described in terms of design implementation and functionalities.

A. Meter

The most obvious requirement of a measurement system is, naturally, to monitor and register the traffic offered to the host in which it is installed (the edge router, in this case). In particular, it is required the measure of the total amount of traffic (in byte) offered to the router over time windows starting at given (but configurable) sample times with customizable duration in the past. More formally, let $A(x, y)$ be the amount of traffic offered to the router within the time instants x and y, $(x < y)$, let τ be the window length and T be the sampling interval. Then, the output of the measurement system should be the sequence:

$$X(nT) = A(nT - \tau, nT).$$

As it will be clear in the following, the system will be able to provide a more refined information in that it will give a matrix of measurement:

$$X_{i,j}(nT) = A_{i,j}(nT - \tau, nT),$$

i, j being the PHB' and LSP' indexes respectively.

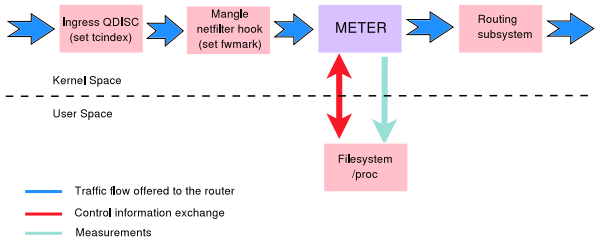

Fig. 3. Meter module location

In order to keep the system as "light" as possible, the core of the system has been implemented as a kernel module. The module, denoted as *meter*, intercepts and registers traffic data within the chain of processing of an IP packet in the Linux router. As shown in figure 3, the meter is placed in series to:

1) Ingress queueing discipline block, devoted to the marking of each packet with a parameter, the so-called `tcindex`, that specifies its PHB.
2) The mangle table of the *PREROUTING* hook of Linux netfilter, devoted to the marking of the `fwmark` parameter which specifies both the output physical interface (in other words, the MPLS virtual interface) on which the packet will be forwarded and which label will be written into the *shim header*.

From the operational point of view, once launched, the meter adds an entry to the `/proc` filesystem of the host machine and immediately starts observing traffic and collecting measurements on `<length>`, `<fwmark>`, `<tcindex>` and `<timestamp>` of each intercepted packet.

From the user space, it is possible, at *any* time, to perform a measurement through the filesystem `/proc` by using the common system calls to access ordinary files. Specifically, the virtual file of interest is `/proc/meter/meter_timeinterval`.

The meter responds to the query with a report which specifies, for each fwmark and tcindex (namely LSP and PHB), the sampling time and the amount of traffic observed within the window horizon (figure 4). "Out-of-window" traffic data are flushed.

The timestamp information is given in terms of number of seconds and microseconds since January 1st 1970 (*wall clock time*). Notice that the choice of timestamping packets in kernel space is not accidental: indeed, that is the only way of getting the exact information on the time instant in which the measurement is taken.

Several extra control parameters are accessible through the filesystem `/proc` such as the window length currently in use, number of measured packets and the latency that the overall measurement system introduces with a precision of about a CPU clock cycle.

As a concluding remark, it should be clear that the meter module has been designed in order to meet two basic requirements: collection of traffic data and timestamping. The very limited functionalities implemented into the kernel space should not surprise: it is wise convention, indeed, to avoid

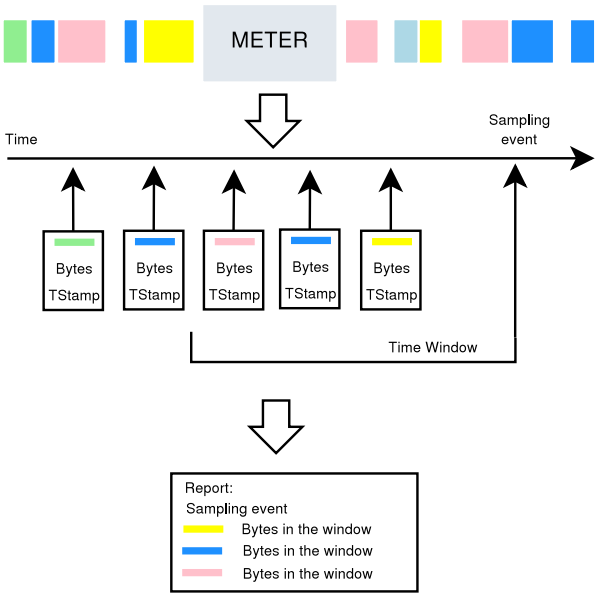

Fig. 4. Operational mechanism and traffic report of the *Meter* module

as much as possible modifications of any type to a so crucial component of a system such as the kernel. Indeed, the modules that will be described in the next sections are implemented into the user space and belong to a unique executable application, named *metercontroller*.

B. Info-unit

As shown in the previous section, the meter module provides traffic measurements for each pair *(fwmark,tcindex)* with no information on per-LSP and per-PHB traffic load. Although fairly sophisticated, this information is not directly usable by network administrators or automated modules for resource allocation which instead, need information on per-LSP and per-PHB traffic load. As well known, LSPs are identified by means of a LSP-ID, unique within the domain and known by the control plane through its RSVP-TE daemons which communicate among each other within the domain.

The *info-unit* has been implemented to sort out the issue of mapping the fwmark parameter into the proper LSP-ID. The translation of tcindex into a PHB identifier is, instead, straightforward as the correspondence is static; thus, this issue has been solved by just means of a table.

The information needed to perform the above detailed translation is retrieved over several system elements, some of them belonging to the kernel. As shown in figure 5, the info-unit should communicate with routing, MPLS and RSVP-TE daemon sub-systems. The interworking with the routing subsystem permits to achieve the information on which MPLS virtual interface each fwmark will be forwarded to. This information, in turn, is used to query the MPLS subsystem to retrieve the label which will be placed into the shim header of packets marked with the same fwmark. The knowledge of the label permits, finally, to find the LSP-ID by querying the RSVP-TE daemon.

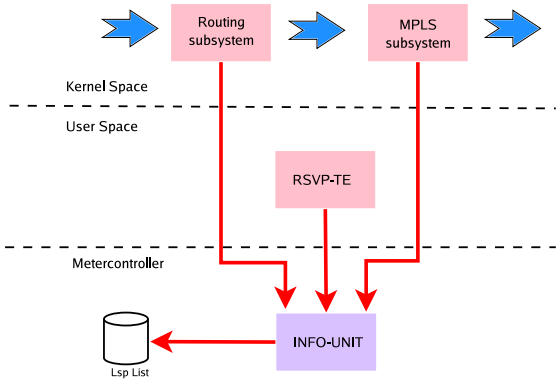

Fig. 5. The *Info-Unit* module

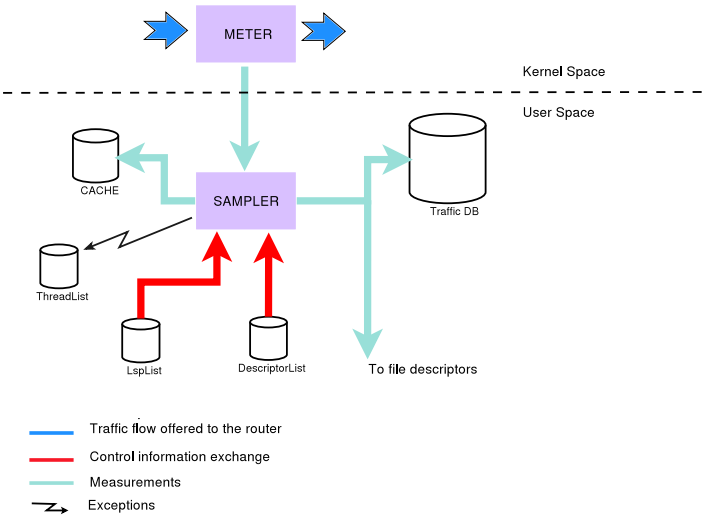

Fig. 6. The *Sampler* module architecture

Once the complete information is achieved, the info-unit generates a list of correspondence fwmark – LSP-ID: throughout this paper, this list will be referred to as *LspList*. As it will be described in the following sections, this table of correspondence will be used by the sampling and traffic data management system to convert the data acquired by the meter module into a format "readable" by the control plane.

The the Info-Unit sub-module in charge of retrieving information from the router subsystems has been implemented by properly modifying the application *tunnel* (from the *Tequila* project [4]) as well as several library functions in a multithread programming context.

At this stage of the presentation, it is worth introducing (it will be elaborated upon in details in the following section) a typical issue which arises naturally when dealing with databases, the problem of data consistency. Information consistency is crucial to assure the correct functionalities of the whole system; for this reason, the info-unit module is equipped with a security mechanism aimed at preserving the information stored in the database should some of the router subsystem delay with data responses. In other words, in case the information requested to the router subsystem does not become available by a given *timeout* (which depends on the sampling rate), the info-unit will not update at all with respect to the previous sampling period, and so the LspList.

As a final comment, the info-unit module acts as the interworking unit among the measurement system and the host machine operating system and is specifically designed to avoid unnecessary waiting times which could significantly deteriorate the traffic information relevance.

C. Sampler

As shown in figure 6, the *sampler* module somewhat represents the core of the whole architecture and is responsible for the most part of the system complexity.

Its target is to query the meter at a given sampling frequency and to report the results on the traffic database, from now on referred to as *TrafficDB*. The traffic database consists of a list of traffic array indexed by the LSP-ID. Each array is implemented as a *round robin database*, that is a static length database of length denoted as `HISTORY_LENGTH`, with a pointer to the oldest data. By replacing the oldest element with the newest one and by incrementing the pointer modulo HISTORY_LENGTH, an array reporting the most recent HISTORY_LENGTH traffic samples (or, in other words, a traffic time series for each LSP) is obtained.

In addition to registering data on the TrafficDB, the sampler is in charge of sending traffic information to the clients connected to the system by writing on the file descriptors which logically represent them.

Naturally, clients may be interested in different data: they may want to retrieve the whole database content, the information associated with some LSP, the very last traffic sample and so forth. File descriptors and the modality in which they can be selectively accessed are contained into an extra structure, the so-called *DescriptorList*), which is managed concurrently by the sampler and by the controller (*CTRL*) module, described in the next section.

TrafficDB is indexed with respect to LSP-ID; the sampler, after retrieving data from the meter, converts fwmarks into LSP-ID by looking up the LspList and only then fills the trafficDB and the file descriptors associated to the clients.

The whole system has been implemented through the so-called *Light Weighted Processes (LWP)*, also known as *thread*: the sampler is the main thread and, once launched, generates two more threads associated with the info-unit and with the *controller* respectively. After the initial setup of all the components, the sampler starts its main cycle which consists of:

- **Sampling** of traffic information from the meter;
- **Conversion** of the pairs $(fwmark, tcindex)$ into the pairs $(LSP-ID, PHB)$;
- **Registering** the retrieved information onto the *TrafficDB*;
- **Writing** the retrieved information onto the file descriptors listed in the DescriptorList;
- **Notifying** the coming-up "end of cycle" the all the system components to let them complete their operations as fast

as possible;

- **Triggering the process *sleeping* phase** whose duration is given by the sampling interval minus the time spent in the previous operations;
- **Returning** back to the beginning of the cycle.

It is worth noticing that the above described cycle is the steady state sequence of operation performed by the sampler, when it is granted with the exclusive access to the TrafficDB. However, this does not hold at all the times. Indeed, whenever a client connects to the measurement system, it may obtain the exclusive access to the database. Since the time needed for server–client information transfer is not, in general, a-priori predictable, the sampler may happen to be stalled waiting for the access to the database, which could result in a delayed sampling time. To get around to this issue, which has to be avoided, the sampler is equipped with a temporary *cache* where it can store data whenever TrafficDB is busy. The size of the cache should be reasonably limited in that its content has to be transferred to the client in less than a sampling interval time. Finally, to avoid that slow clients determine the cache overflow, a signalling mechanism triggers the immediate closing of data transfer to the client in case the cache content level hits a given "safeguard" threshold.

The rest of the section describes in details each of the above listed operations.

1) Sampling: The operation of reading data from the meter is pretty straightforward as it consists of simple reading access to a file. The sole trick that has been used is the introduction of a end-of-message marker to avoid retrieving useless and potentially dangerous information. Information is stored in a data structure called *TrafficList* which contains the number of measured bytes for each pair $(fwmark, tcindex)$. At this time, the sampler also registers the timestamp associated with the sampling time in the `HISTORY_LENGTH` vector.

2) fwmark Conversion: All the structures used in the system are created to let the processes to communicate with each other. In this light, the *LspList* acts as the way the sampler and the info-unit communicate.

The modularity of the whole system design and the processes'communication require, though, a controlled concurrent access to the data structures that permit the communication. As an example, as long as the LspList is in use by the sampler, it must not be accessed by the info-unit to avoid the whole system to collapse.

The access control to the structures is implemented by means of specific variables called *semaphores* for which the operating systems guarantees the *atomicity* of the operations that involve them. In other words, while a thread is about to modify a semaphore, no other processes can access it.

As previously detailed, the LspList contains the mapping fwmarks – Lsp-ID provided by the info-unit to be passed to the sampler. Moreover, since it is filled through data retrieved by the RSVP-TE daemon, it provides information on the number of LSPs installed on the router together with their identifiers. The knowledge of these data allows the sampler to consistently update the TrafficDB.

3) Information storing: The comparison between the trafficDB content and the LspList allows the sampler to build a new TrafficDB in which the LSPs teared down during the last sampling time are removed and the new LSPs established in the same time interval are added up. This way, the TrafficDB is always synchronized with the actual dynamic of the system which is perfectly known by the control plane. Naturally, the sampler also registers onto the TrafficDB the traffic information retrieved by the TrafficList.

4) Information transfer to clients: Clients are connected to the server through TCP connections. At the operating system level, they are represented as file descriptors, that is, as integer numbers.

File descriptors and clients'requirements are stored in the DescriptorList. The whole operation is carried out in parallel to the TrafficDB update in order to achieve a full synchronization of data registered in the database and presented to the clients.

5) End of cycle notification: As soon as the sampler has terminated its cycle, it communicates to the system that the structures are available by simply incrementing the semaphores involved in its previous operations.

6) Sleeping: The amount of time needed to complete the above described operations depends on many factors such as the number of installed LSPs, CPU processing capacity, etc.. Thus, the time which elapses between the end of the cycle and the next sample time is not a-priori predictable and, in fact, should be computed each time.

A possible way of implementing the waiting time by means of a program is to let a process count a number of CPU clock cycle which corresponds to the desired waiting time. This operation is known as *busy wait* in that the processor remains active by occupying 100% of the host machine CPU. This algorithm, though, does not guarantee any accuracy in the computation of the waiting time unless the process is given the highest priority. As a result, the router would get stuck in an essentially useless operation; even packet forwarding would be stalled.

For those reasons, the sampler features a closed loop sampling mechanism based on inactive waiting times that uses the *nanosleep()* library function. This function forces the system to a sleeping state in which the kernel does not assign the CPU to the sampler until a timeout expires.

The use of nanosleep() or its similar (sleep(), usleep(), etc.) arises the natural problem of the accuracy of the actual waiting time. Indeed, the above functions guarantee that the waiting time is *at least* the requested waiting time; an extra stochastic term which depends on many factors (CPU load, interrupt rate, etc.) has to be accounted for. Moreover, as the processing time the sampler needs to carry out sampling and database management is variable, the system should make up for as quickly as possible.

For those reasons, the use of a predetermined waiting time results in an intolerable derive of the sampling interval, and a more clever algorithm to compute the waiting time is needed. The proposed solution aims at tackling the two main causes of errors, namely:

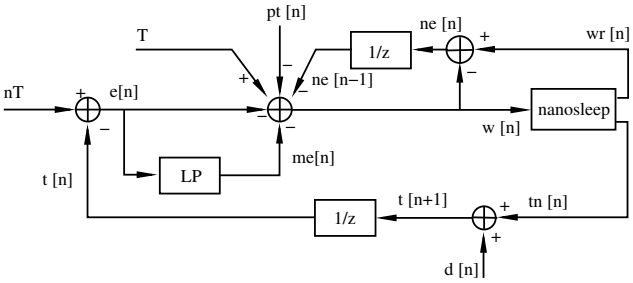

Fig. 7. Block diagram of the waiting time computation algorithm

- data processing delay of the sampler;
- intrinsic delay introduced by the nanosleep();

and is represented in figure 7. The algorithm is based upon the comparison between the actual and the ideal sampling times. The stochastic error sources are accounted for through the nanosleep block and through the "processing delay" input.

The first sum block computes the time already spent within the sampling interval due to the above detailed causes and has to be deducted from the nominal waiting time T. This term represents the total sampling error which is averaged through a simple low pass filter. To obtain the input parameter of the nanosleep function, this quantity is deducted from the nominal sampling interval together with the processing time and a simple prediction of the error which will be introduced by the nanosleep block. Such a prediction consists of the error the nanosleep introduced the step before. The rationale of this approach relies upon the correlation of such an error, that strongly depends on the system load which, in turn, is supposed to be slightly different over back to back sampling intervals. The output of the system is the actual sleeping time.

Roughly speaking, the actual sequence of waiting time represents a *point process* with sampling epochs $t_n, n = 0, 1, 2, \ldots$ which ideally should be as close as possible to the sequence $T_n = nT, n = 0, 1, 2, \ldots$. The system shown in figure 7 is designed to force the actual sampling pace to meet the above requirement.

D. Controller

The *controller* module permits to external entities the access to the measured data. It is relevant to note that the module has a high importance, since the whole system is useful only if it is able to acquire traffic data and make them available to the external world. To release the acquired data, this module should manage a certain number of tasks, such as the support for the remote login, the access control through authentication, the support of different clients connected at the same time (and the related problems), the remote control of the job parameters of the system, the error management which should be friendly etc... The experimental software developed sofar does not include some of these functionalities, but it has been thought for an easy integration of them, trying to account for all the possible scenarios of the system utilization.

As the figure 8 illustrates, the system considers the presence of several controllers, one for every client. Each controller

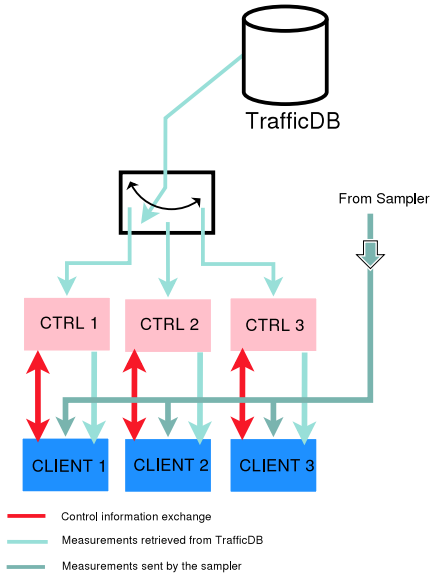

Fig. 8. The Controller module

takes care of the communication with a single client, of the access control to the *TrafficDB* and of the forward of the client requirements to the sampler, which then can send only the data required by the client. The controller module is realized like a *semi-concurrent server* that accepts connections on a TCP port. Upon a connection request arrival, it starts exchanging messages with the client in order to understand its requests; after that, it tries to obtain the exclusive use of the *TrafficDB*. The messages set used in this phase will be defined by means of the markup language, XML. When the system gives the permission to access to such a structure, the data transfer towards the client starts and, at the same time, the sampler begins writing data in the cache. The module *controller* is equipped with a signalling system able to stop the data transfer from the sampler to the client whenever the filling level of the cache exceeds a threshold that represents the limit beyond which the data integrity is not guaranteed. Once the client requests are satisfied, another copy of the controller is instantiated. This copy is used to manage connection requests provided by other clients, while the old controller instantiation remains connected to its client for further messages exchanges. At the time the client decides to close the connection, the controller carries out all the operations necessary to free the memory of the *DescriptorList* and then it stops. To date, the developed module implements the management functions of the signalling coming from the sampler, the access control of the different controllers to the *TrafficDB*, the communications towards the sampler of the client requests and the operations needed to free the memory when the client logoffs.

E. Client

The client to access to the described system information is not yet implemented, although some parts needed in the described system are already integrated. In particular, it is relevant to note that the client will have to maintain either

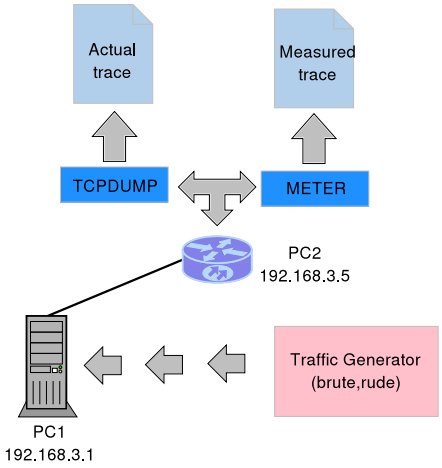

Fig. 9. Testbed used for the experimental analysis

a copy of the TrafficDB or a part of it in order to carry out monitoring, forecasting, management, etc... All the given structures and the related management routines are already implemented and tested and therefore the client developer should implement the communication and monitoring utilities only.

IV. Test and Measurements

This section reports the preliminary results on functional and performance tests carried out on the whole system. The focus is concentrated on evaluating the behavior of the meter module and on the overall effectiveness of the system in capturing actual traffic flows.

The testbed considered to test the system prototype is shown in figure 9; it is composed by two PCs connected by means of a Fast Ethernet switch. Each PC is equipped with a Fast Ethernet NIC based on Realteck 8139 chipset.

The PCs have the following hardware features:

- **PC 1** AMD AthlonXP 2000+ , 512 MB RAM memory, O.S. Slackware Linux 9.0, kernel 2.4.20 (MPLS Patch), network address 192.168.3.1
- **PC 2** AMD K6 II 400 MHz, 196 MB RAM memory, O.S. RedHat Linux 8, kernel 2.4.20 (MPLS Patch), network address 192.168.3.5

The traffic loading the measurement system has been generated by means of the software traffic generator *rude* [5] version 1.62, which permits to generate packets starting from a file containing the inter-departure time and the size of each packet to transmit; this permits to generate VBR traffic starting from data acquired during measurement session carried out using software *sniffer* such as TCPDump. On the other hand, the CBR traffic has been generated using the application *brute* [6].

The experimental tests have been carried out by varying some parameters:

- **The bitrate** in order to verify the accuracy of the *meter* module under different traffic load conditions
- **The packet size** in order to heavy speed up the timing system integrated in the application *meterctrl* that, at least in its first versions, was blocked from an excessive number of interrupt requests. It is relevant to note that when short packet sizes are considered a fixed bitrate is achieved using a higher packet rate; this implies a higher number of interrupt requests in all nodes crosses by the packet flow.

A. Meter module performance

Given that the *meter* module is implemented as a hook of the netfilter of the O.S. Linux, it manages all the packets transferred from the NIC to the IP layer of the kernel. This observation permits to deduce that the module does not drop any packet since each time a packet enters in the kernel it is normally processed by every hook of the netfilter. On the contrary, if the kernel does not accept a packet, the module will not see it and, therefore, the information on this packet will be lost. It is worth considering the case where a packet could be accepted from the kernel, be measured from the *meter* and then discarded from the output scheduler. However this case is not considered in the proposed system, since it is planned to measure the traffic *offered to the router*, regardless that this traffic is forwarded in MPLS domain or not.

In the test of the system, it is interesting to evaluate the latency added from the module to the single packet. To quantify the latency, a quite objective performance parameter is the number of clock cycles that a packet spends in the module. This number is quite independent of the hardware features of the equipment hosting the measurement system as long as the O.S. kernel can manage the interrupts without the overlapping of the routines managing them. As an example of the problems deriving from this overlapping, it can be considered the case when a packet arrives at the NIC before the kernel has finished the processing of a previous packet. In this case, the routine managing the interrupt produced by the arrival of the new packet runs over the CPU, producing the stop of the active task. Packet processing will resume as soon as the routine finishes its operations. Hence, the permanence time (expressed in terms of clock cycles) of a packet in a hook can be much higher than the time necessary to the module to carry out the elaboration of the same packet.

Figure 10 shows this phenomenon observed in the module *meter* installed in the router *PC2*. The plot has been obtained by carrying out several experiments with CBR traffic at different packet rates and averaging the delay on the number of received packets that, considering a constant duration (equal to 30 seconds) of the experiments, varies with the packet rate. Furthermore, the plot reports the percentage of CPU occupation observed at different packet rate.

Also, figure 10 shows that the delay is nearly constant as long as the CPU occupation is below 100% while, thereafter (about 40000 pps in the specific case), the module propagation delay grows. Indeed, at full load the processor cannot effectively handles all the interrupts and the ordinary processing of

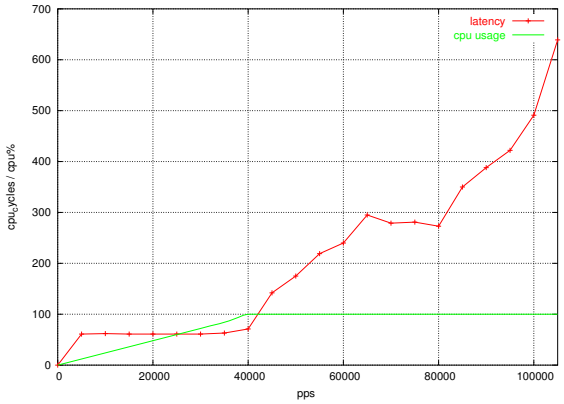

Fig. 10. Permanence time of the packet in the meter (clock cycles)

packets at the data plane level (forwarding, policying, marking etc...) is dramatically impaired.

An analogous phenomenon occurs even with more powerful processors: naturally, the threshold beyond which the delay increases is higher. The use of dynamic memory in the implementation of the *meter* does not modify the qualitative behavior above described: the latency, though, is more than ten times higher.

To summarize this discussion, when the router operates in normal conditions, the processing time of packets due to the *meter* module is of the order of hundreds of clock cycles. This results in a negligible latency which, moreover, decreases as the router CPU capacity increases.

B. Overall system performance

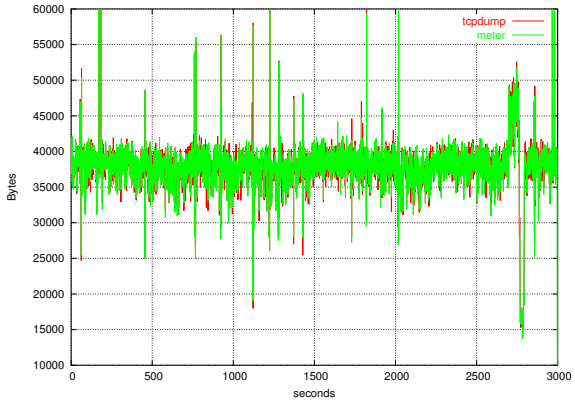

Fig. 11. Comparison with tcpdump: videoconference session

From a higher level point of view, the whole system performance was first assessed by using it to monitor a videoconference session traffic flow. Figures 11, 12 and 13 report the sequence of samples taken on-line by the system together with the ideal sequence of samples obtained by post-processing of data acquired by `tcpdump` [7]. The results confirm the ability of the measurement system in closely capturing the actual traffic dynamic even in case of very quick

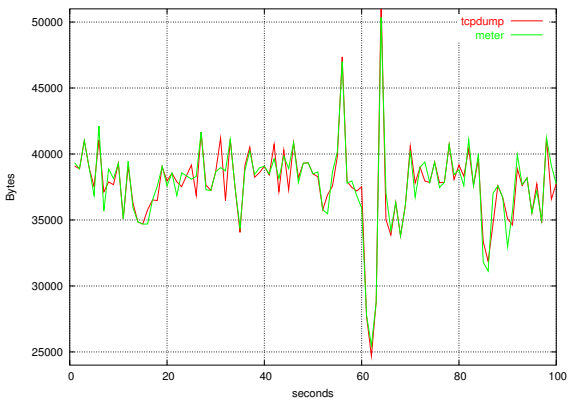

Fig. 12. Comparison with tcpdump: (videoconference session: 100 seconds detail

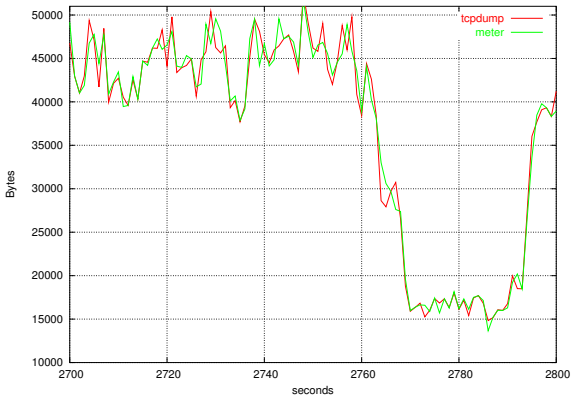

Fig. 13. Comparison with tcpdump (videoconference session: 100 seconds detail

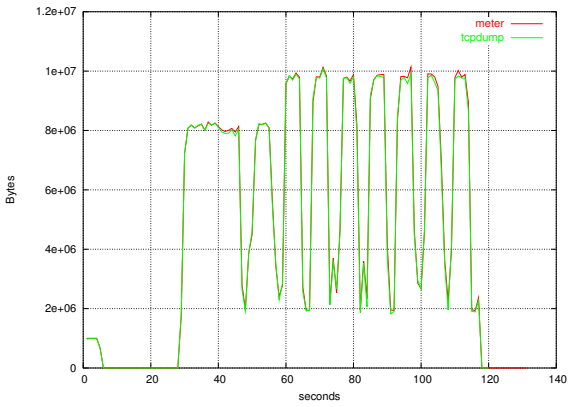

Fig. 14. Comparison with tcpdump on traffic measurements (FTP traffic)

changes of the traffic pattern as proven by 12 e 13 which zooms a 100 sec. slice of the overall traces of 11.

Analogous comments apply to figure 14 which shows the result of a similar test carried out to monitor the traffic produced by a file transfer session.

V. Conclusion

The paper describes in detail the architecture of a soft real time measurement system designed to resides into a DiffServ over MPLS Linux based router. The goal of this research was to develop an open source software product to be used for management purposes and to be integrated in the network control plane to automate resource allocation and admission control functionalities. The system is designed to be flexible, configurable and modular and each module has been implemented in order to minimize the impact of the system itself on the traffic dynamics. According to configurable sampling frequency, the system produces a sequence of per-PHB and per-LSP traffic measurements taken over a customizable time window. The system has passed several functional tests and preliminary experimental results show that it accurately captures the traffic patterns offered to the router.

Acknowledgment

This paper is partly supported through the MIUR FIRB projects TANGO and VICOM.

References

[1] http://tango.isti.cnr.it/
[2] http://www.vicom-project.it/
[3] G. Carrozzo, N. Ciulli, S. Giordano, *Multi Access Inter Domain architecture for QoS provisioning in MPLS/DiffServ networks* submitted to TANGO internal workshop, Madonna di Campiglio
[4] http://www.ist-tequila.org/
[5] http://rude.sourceforge.net/
[6] http://netgroup-serv.iet.unipi.it/brute
[7] http://www.tcpdump.org