A Testbed Environment for the Performance Evaluation of Modular Network Architectures *

Dario Maggiorini, Elena Pagani, Gian Paolo Rossi
Dipartimento di Scienze dell'Informazione
Università degli Studi di Milano
via Comelico 39, I-20135 Milano, Italy
{dario,pagae,rossi}@dsi.unimi.it

Abstract: In this work we propose an original approach to evaluate the performance of network protocols and distributed applications. Our methodology is intermediate between the use of simulation techniques and the deployment of the code under test in a real system. It allows to test protocols in real environments. At the same time, it allows easy prototyping and modification of the protocols; code modules can be easily combined into different protocol stacks.
In this work we describe our preliminary implementation of the testbed environment we propose, and we discuss the experimental results obtained by using the framework to measure the performance of QoS protocols.
Keywords: network protocols, system architectures, performance evaluation, Quality-of-Service.

1 Introduction

When deploying network protocols, designers face the problem of testing the correctness and evaluating the performance of those protocols. The usually adopted approaches are either (i) the deployment of the protocol in the framework of a simulation environment, or (ii) the implementation in a real environment. Both approaches have pros and cons.

The use of simulators can be the only way to study new protocols, when a real network is not available to perform testing. Simulation techniques allow fast prototyping and modification of the protocols. The protocols can be tested under network conditions that are hard to reproduce in real environments. On the other hand, simulators are often not able to catch many interesting properties of the real systems. As an example, they usually do not allow to estimate the computational and memory overheads of a protocol. Moreover, it is difficult to characterize a real traffic pattern to use as an input to the simulators, to test the protocol behaviours in realistic conditions.

By contrast, the implementation of new protocols in a real system, although it can solve some of the above mentioned problems, has several drawbacks. The protocols must be implemented off-line. Only a few tools exist (e.g. `netperf` [1], `tcpdump` [2]) that allow to perform extensive testing and performance evaluation. Moreover, the result interpretation can be difficult because of the interferences with other concurrently running protocols and applications. The testing process is time-consuming because of the need of continuously performing the shutdown and reboot of the system. If testing has to be performed by working at the kernel level, security issues arise due to the side-effects that programming errors could have on both the network and the local host.

The greater is the number of functional modules constituting the architecture under test, the more the problems discussed above are emphasized. This is for instance the case of the proposed architectures to support Quality-of-Service (QoS) in the Internet [3, 4, 5]. Several possible policies have been proposed in the literature to realize the modules composing those architectures. The literature is fragmented and the understanding of the issues involved by QoS is at a preliminary stage. For the purpose of characterizing good standard protocols to efficiently provide QoS, the behaviour of those policies should be studied. But, in a real network it is hard to modularly add and substitute code pieces to perform experiments with different protocol stacks.

We are deploying a framework that supports a sort of *programming by components*: in this framework different functional modules can be incrementally added and modified, to observe the behaviours of the resulting architecture under different system conditions. The framework is built by copying the TCP/IP protocol stack to user level and by working on that copy. This will allow protocol designers to test and evaluate their protocols in a real environment, without dealing with the problems discussed so far. The framework allows to easily substitute code modules without halting the system. It has embedded measurement tools to compare different protocols using uniform metrics. The modules inserted into the framework can be either freeware or developed ad hoc.

* This work was supported by the MURST under Contract no.9809321920, "Transport of multicast packets with QoS guarantees".

In section 2, we give an overview of the framework structure. In section 3, we discuss our initial implementation, and we describe how we use the framework to evaluate the performance of QoS protocols. In section 4, we report some experimental results obtained with the framework. Concluding remarks and future deployments are reported in section 5.

2 Framework architecture

The framework exploits the duplication of the TCP/IP stack at the user level. The experiments with new protocols are performed by using that copy. The copy, in its turn, relies on the services of the system stack, as an usual user application. In figure 1 we show the components of our framework.

The stack in the user space is an infrastructure that encapsulates code modules implementing the protocols under test. It reproduces message passing through different software layers corresponding to the transport and network layers in the OSI model. Message passing amongst user stacks is realized by using UDP sockets. Test applications and application level protocols can be implemented that use the services of the modified TCP/IP stack. The behaviour of both these applications and the new modules embedded into the stack are evaluated by means of a set of measurement tools aimed at recording a real-time trace of the protocol activities that can be successively analyzed.

From the implementation point of view, a stack is a sequence of simple, almost identical modules structured as shown in figure 2. Object code derived from either kernel modules, freeware code or code developed ad hoc, is inserted into an environment which will behave like the upper and lower layers such code is expecting. Each module in the user stack must involve procedures that perform logging and time coordination. We refer to these procedures as *layer tools*. When data are received in input to a module, they are managed by the companion layer tool before being delivered to the actual protocol implemented by that module.

The layer tools are accessible in two ways:

- from the upper level, through an interface exporting the same functionalities and behaviour of the corresponding kernel module;
- from the user, through a simple interface consisting of two pipelines: one for data input and one for data output. These two pipelines are used to inject simulated traffic in the system as a byte stream and to retrieve it at the destination.

The main issues to be dealt with in the framework deployment concern:

- the characterization of the performance parameters that the measurement tools must evaluate;
- the design of lightweight techniques that reduce the impact of the measurement process on the obtained results;
- the definition of an API which is common to all the modules implementing a given service (e.g., packet scheduling, multicast routing). Such a definition guarantees easy substitution of the modules for experimentation purposes.

3 Preliminary implementation

We have realized a preliminary implementation of the proposed framework for the testing of protocols for the support of QoS in best effort IP-based network infrastructures in the Internet [3, 4, 5]; we show in figure 3 the layout of the framework structure. The framework has been implemented on the `Linux` OS. It is so far composed of a traffic generator and of measurement tools to analyze network throughput, delay and jitter [6].

All the modules have been implemented in C language; the graphical interface has been realized in Java.

The offered traffic can have either a mathematically generated distribution (CBR, VBR or Poisson distributed) or it can be generated by sending out at a constant rate bytes read from a file (e.g. an MPEG or MP3 file). In the latter case, the data are sent as a feed to a multimedia application that will consume data through the network at a certain rate.

As a future extension we are planning to generate traffic starting from a previously recorded dump. This will allow to test different protocol versions under the same workload and to be able to measure effective performance enhancements.

The measurement tool is a unidirectional pipe with a buffer of size s_b, that can log the transfer of s_b bytes with a granularity of 1 millisecond. We are working to make it as lightweight as possible to achieve a sustained bandwidth of at least 200 Mbps for memory-to-memory copy on an entry level PC and to be able to perform measurements on a fully loaded fast ethernet connection.

The measurement tools produce dump files describing the observed traffic. They are integrated with a graphical log analyzer that allows an easy result interpretation. The events recorded in the dumps are both (i) data reception from an input pipe and recording in a buffer, and (ii) data transmission over a pipe or a socket.

When a module starts, it can create and use one or more dumps through an external library. The default log is 1024 seconds in length. For every second a kilobyte of information is produced; thus, the default log size is one megabyte. As the first step, an all-zeroes file is created on the local file system, then that file is mapped to memory using the `mmap` system call. This double step is done primarily for compatibility, since `Linux` cannot allocate anonymous memory mapped chunks. Getting the dump directly mapped to memory allows us to manipulate it as a linear array of characters (direct addressing). Since it is mapped from a file, measurements can be retrieved even if the application crashes. A global timer is initialized when the first dump is created,

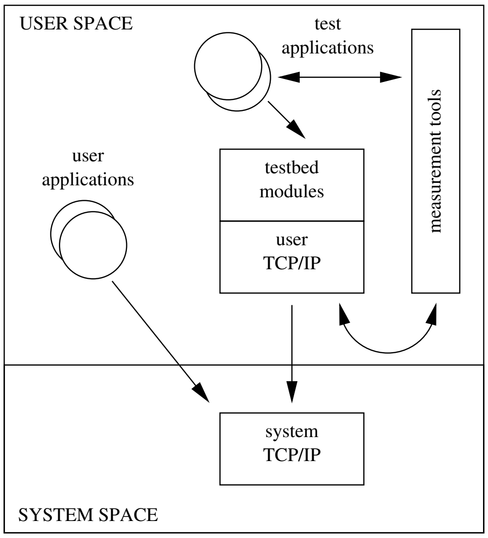

Figure 1: Layout of the experimental framework.

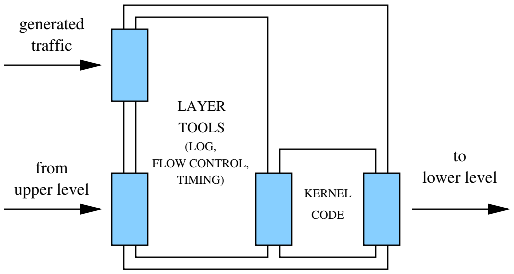

Figure 2: A single module detail.

to ensure that all the logs are synchronized during the measures. When the application needs to record a new event in the dump, it just calls a library API. Inside the library the `gettimeofday` system call is used to retrieve the actual date with microseconds granularity. The part relative to the seconds is used to address the kilobyte chunk inside the dump, while the microseconds fraction is used to address the byte inside the chunk. To get the best performances, microseconds are traslated to milliseconds with a ten bits shift to the right. The resulting number is used to address the memory mapped area. Since this conversion is not the result of a division we get a little time slack in the dump. A second is not made up of 1000 units but rather $1000000/1024$, which means 977 units. This can be used to explain why even measuring CBR traffic on the source small traffic drifts are observed.

Generated dumps are evaluated from an external program that downsamples the data to a required timeslice, (200 milliseconds by default), and generates data to be feeded to the `gnuplot` tool.

We are performing experiments with the described framework to verify that the impact of the measurement tools is reduced.

Our testbed network is currently constituted by a subsection of the department network. The network is totally switched using Cabletron 8600 devices. The switches implement a per-port routing giving ideally a personal subnet to every host. The connected hosts are PC's running either FreeBSD 3-4-STABLE or Linux Redhat 6.1; they all are equipped with 3Com 3C905B LAN NIC. The connection speed between the hosts and the nearest switch is 10 Mbps using UTP-45 whereas the connection between every two different switches is 1 Gbps using optical fiber .

3.1 Example of the framework usage

To realize a preliminary self-contained, working structure, our user stack is currently composed of a freeware implementation of RTP [7, 8] which is interfaced with the copy at user level of the UDP kernel implementation. This architecture has been used to evaluate the behaviour of the layer and measurement tools.

RTP is an end-to-end transport protocol for real-time traffic. The use of RTP allows us to achieve QoS in terms of the delivered jitter. The information provided by RTP can be used by a CBR application to guarantee a constant rate delivery

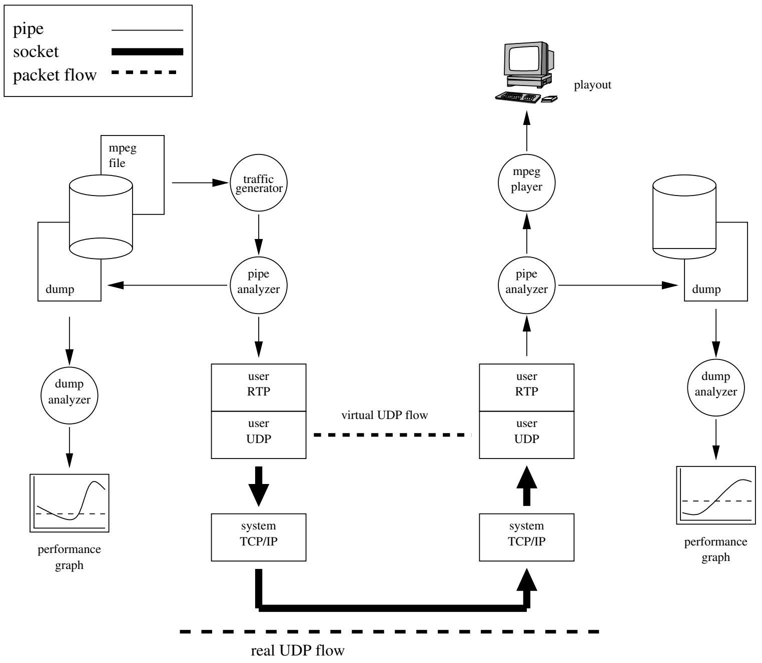

Figure 3: A measurement scenario.

of the data packets at the destination.

With reference to figure 3, the traffic generator is directly interfaced with RTP using the API of the layer tools. At the sender host, RTP marks the generated messages with a timestamp and a sequence number. At the receiver, the messages are stored in a *playback* buffer, according to the sequence number order. Sequence numbers are as well used to detect duplicates, which are discarded.

Messages are extracted from the playback buffer at a constant rate to be delivered at the player process. We measure the traffic profile both when messages are received at the destination host and before being delivered at the player. Delivery starts when the number of messages in the playback buffer exceeds a high-water mark, and is performed according to the sequence numbers. The playback process makes use of a timer to estimate when the next message should be delivered. If the message with the appropriate sequence number is not in the buffer when the timer expires, the playback process delivers a `dummy` message to the player, increments the sequence number to be delivered next and restarts the timer. If a message m is received, whose sequence number is lower than that of the next message to deliver (i.e., m is a late message), m is discarded.

4 Experimental results

We performed measures over the subnet shown in figure 4, in the absence of background traffic (we estimated that the background traffic is around 3% of the total capacity of the network). `wagner` generates QoS traffic at a rate of 1.5 Mbps; packets are addressed to `netdev`. `mozart` produces noise traffic addressed to `giotto`, so that the traversed routers are congested. The noise generator gradually increases the produced traffic, starting from 0 Mbps, until a rate of 10 Mbps is reached. Router congestion is achieved within approximately 1000 sec. The traffic increase is obtained by sending noise packets at decreasing delay; the new intra-packet delay is recomputed every 60 seconds. Both data and noise packets have size 1KB, so that the overhead both in accessing the file and in the measurement tools operations is minimized. In figure 5 we show the profile of the traffic generated by `wagner`, while in figure 6 we show the profile of the generated noise traffic.

The results concerning the traffic profile received at `netdev` are shown in figures 7, 8 and 9. After 500 sec. links and switches are congested by noise traffic, that affects the regular profile of the CBR traffic by introducing irregularities in the QoS flow. QoS packets are received in late, and sometimes bursts of packets are received. The mean intra-packet delay increases because of noise packets that are forwarded interleaved with the QoS packets. Because of this interleaving, the spacing between subsequent QoS packets becomes variable, as shown by the increase in the jitter.

In the presence of network congestion, the playback process must anyway provide a regular profile of the traffic delivered to the player. In figures 10, 11 and 12 we show the throughput, intra-packet delay and jitter measures we achieved for the traffic outcoming from the playback process, by setting the high-water mark of the playback buffer equal to 1000.

While the received QoS flow is regular, every time the playback process awaken to deliver a message, it finds the expected message in the buffer. When QoS flow starts to

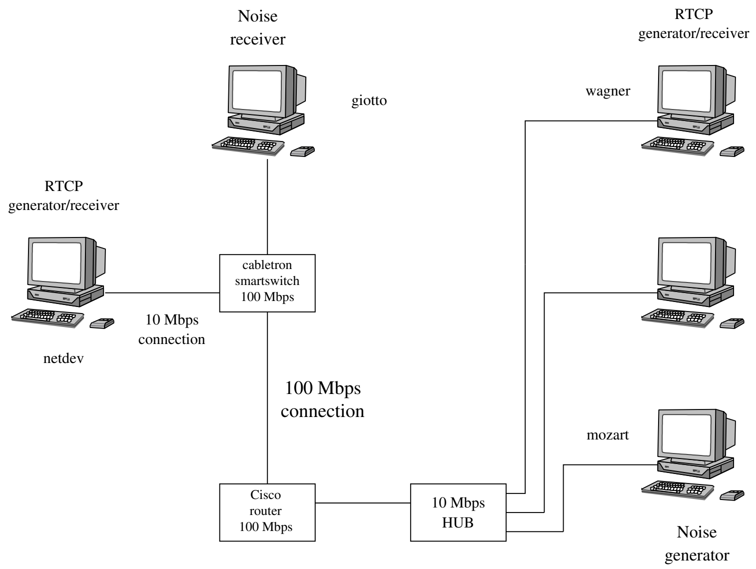

Figure 4: Topology used for the experiments.

be negatively affected by noise, the lenght of the playback buffer drops under 50 packets and shows a saw-tooth behaviour with peaks in correspondence of the reception of a packet burst. The playback process has to generate dummy packets at an increasing rate to guarantee CBR delivery in spite of the gaps in the buffer. In figure 13 we show the throughput of the data packets delivered to the player. The mean throughput decreases as the noise increases, because QoS packets are received at a rate lower than that adopted by the playback process. By contrast, the throughput of the dummy packets increases (figure 14).
We are currently performing experiments to compare different playback policies that guarantee better quality of the information flow delivered at the player. Those policies try to dinamically adapt to the changing network conditions. For instance, the packet delivery is suspended when either the number of packets in the playback buffer drops below a low-water mark or the end-to-end delay increases. Delivery is resumed either when the buffer length exceeds the high-water mark or after a time equal to the current end-to-end delay is passed.

5 Concluding Remarks

In this work we propose a framework to perform the testing and evaluation of network protocols and distributed applications. We are initially implementing the framework focusing on protocols supporting QoS. As a long term objective, we plan to realize a comprehensive and general environment of code production and testing with integrated capabilities for fine-tuned measurements.
The proposed approach offers several advantages over the standard testing methods. It bases on moving the code to user level to reduce the development and testing difficulties. This way, the protocols can be evaluated in a realistic environment, while many problems related with the implementation in a real network are avoided. As the consequence, performance evaluation can be performed of the real code. At the same time, the produced code can be easily ported into the kernel, to make the protocols actually working.
We are currently devising a generalized interface for the stack modules. Such an interface should be then specialized for each layer. We are as well extending the described architecture with several packet scheduling modules. Many policies have beed proposed in the literature to perform packet scheduling and forwarding so that end-to-end QoS is guaranteed, such as `WFQ` [9],`WF2Q` [10], `RED` [11], `WRR` [12]. Experiments performed with simulation tools highlighted the impact of the different policies on the obtained QoS [3]. The implemented schedulers are added to the framework on top of the system UDP and below the user UDP. We re-order the packets accordingly to the scheduling policy under test, before sending them in the network, thus overriding the default (FIFO) IP packet scheduler.

Acknowledgement. We want to thank Mr. Jori Liesenborgs for his unvaluable support in the deployment of the RTP libraries.

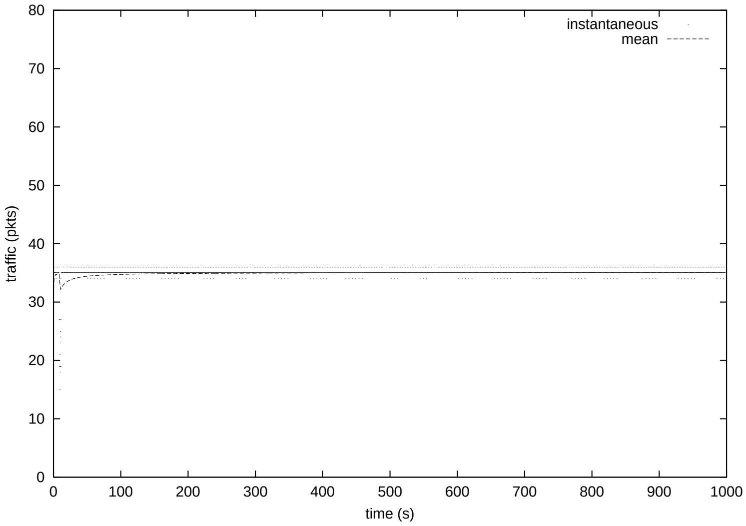

Figure 5: Data traffic generated by `wagner`.

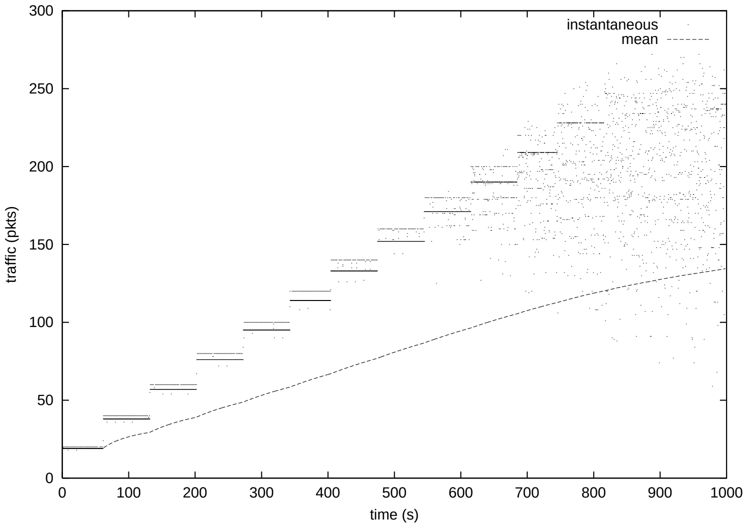

Figure 6: Noise traffic generated by `mozart`.

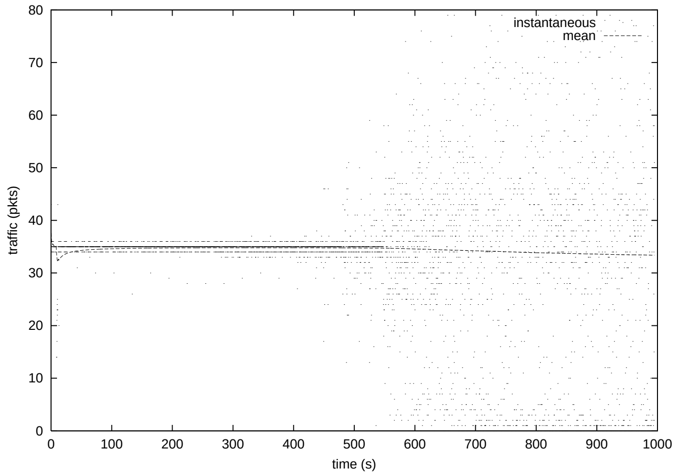

Figure 7: Incoming traffic at `netdev`.

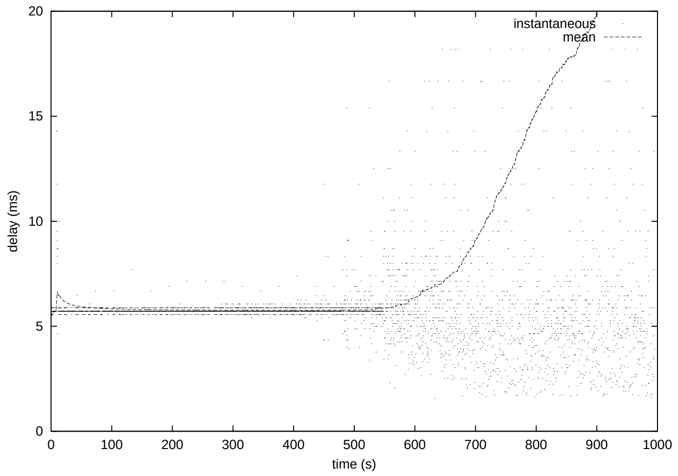

Figure 8: Intra-packet delay for incoming traffic at `netdev`.

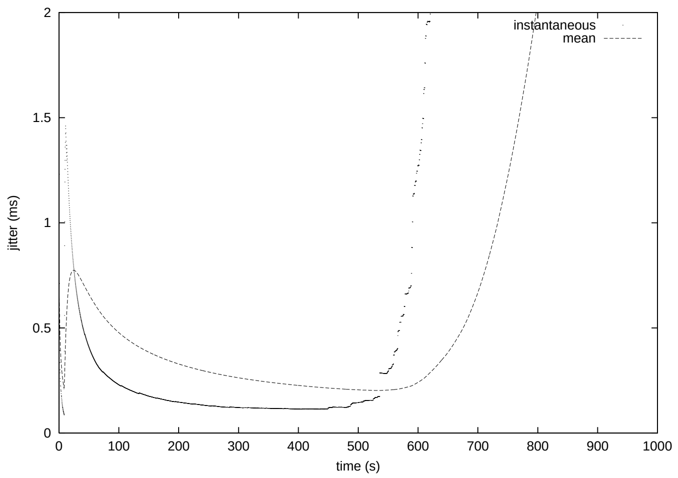

Figure 9: Jitter for incoming traffic at `netdev`.

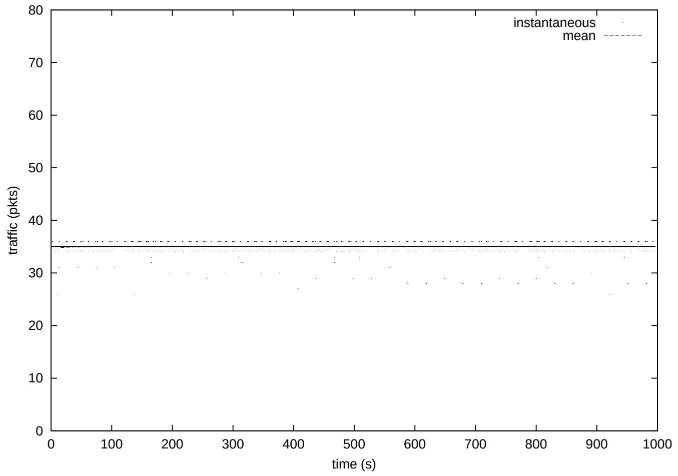

Figure 10: Outgoing traffic to the application.

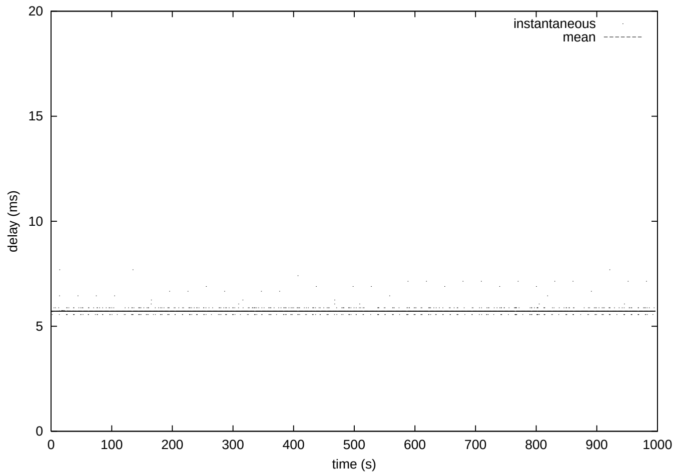

Figure 11: Intra-packet delay for outgoint traffic to the application.

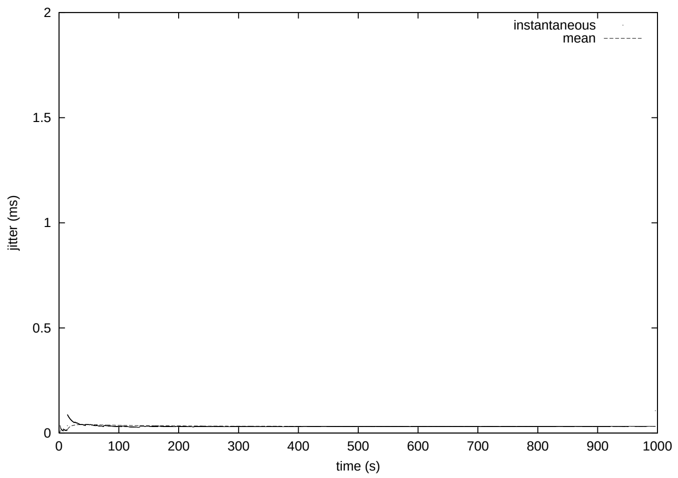

Figure 12: Jitter for outgoing traffic to the application.

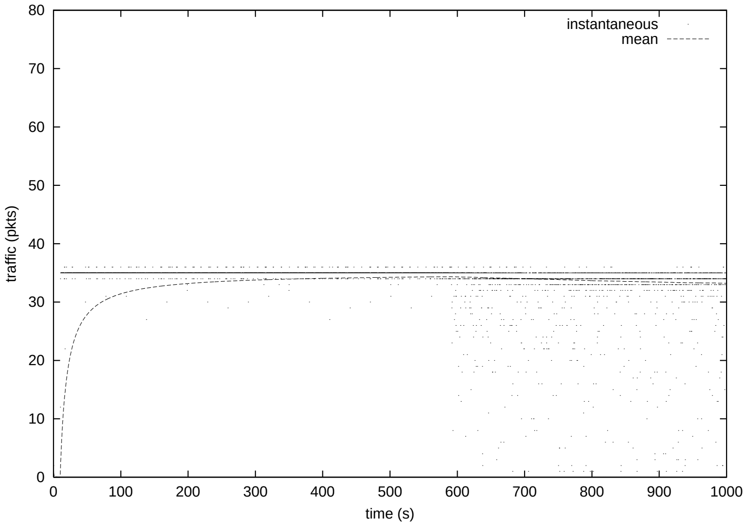

Figure 13: Throughput of the data packets delivered to the application.

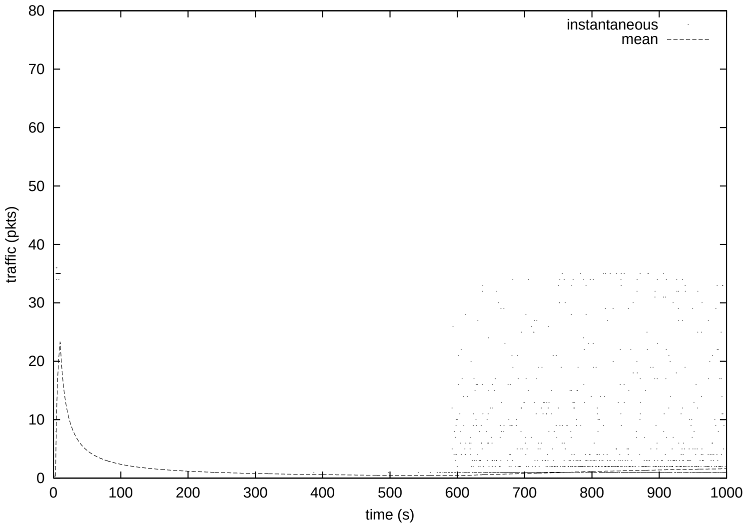

Figure 14: Throughput of the dummy packets delivered to the application.

References

[1] Jones R., Choy K., Shield D., ""Netperf"." HP Information Networks Division, Networking Performance Team, `http://www.netperf.org/`.

[2] Jacobson V., Leres C., Mc Canne S., ""TCPdump"." Lawrence Berkeley Laboratory, University of California at Berkeley, `ftp://ftp.ee.lbl.gov/tcpdump.tar.Z`.

[3] Pagani E., Rossi G. P., ""Architectural Requirements for the Transport of Multicast Packets with Maximum Delay Delivery Guarantees"," *Technical Report 244-99, Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano*, Nov. 1999.

[4] Blake S., Black D., Carlson M., Davies E., Wang Z., Weiss W., ""An Architecture for Differentiated Services"," *Internet Draft, draft-ietf-diffserv-arch-02.txt*, Oct. 1998. Work in progress.

[5] Braden R., Clark D., Shenker S., ""Integrated Services in the Internet Architecture: an Overview"," *RFC 1633*, Jun. 1994. Work in progress.

[6] Bagnall P., Briscoe R., Poppitt A., ""Taxonomy of Communication Requirements for Large-scale Multicast Applications"," *Internet Draft, draft-ietf-lsma-requirements-02.txt*, Nov. 1998. Work in progress.

[7] Schulzrinne H., Casner S., Frederick R., Jacobson V., ""RTP: A Transport Protocol for Real-Time Applications"," *RFC 1889*, Jan. 1996. Work in progress.

[8] Schulzrinne H., ""RTP: an overview"." `http://www.cs.columbia.edu/~hgs/rtp/`.

[9] Shreedhar M., Varghese G., ""Efficient Fair Queuing Using Deficit Round Robin"," *IEEE/ACM Transactions on Networking, Vol. 4, No. 3*, pp. 375–385, Jun. 1996.

[10] Bennet J.C.R., Zhang H., ""WF^{2}Q: Worst-case Fair Weighted Fair Queueing"," *Proc. IEEE Infocom'96*, pp. 120–128, Mar. 1996.

[11] Floyd S., Jacobson V., ""Random Early Detection Gateways for Congestion Avoidance"," *IEEE/ACM Transactions on Networking, Vol. 1, No. 4*, p. 397, 1993.

[12] Katevenis M., Sidiropoulos S., Courcoubetis C., ""Weighted Round-Robin Cell Multiplexing in a General-Purpose ATM Switch Chip"," *IEEE Journal on Selected Areas in Communications, Vol. 9, No. 8*, p. 1265, 1991.

[13] Mills D. L., ""Network Time Protocol (Version 3) Specification, Implementation and Analysis"," *RFC 1305*, Mar. 1992. Work in progress.

[14] Fall K., Varadhan K., "ns Notes and Documentation". The VINT Project, Jul. 1999.