Communicating Process Architectures 2005
Jan Broenink, Herman Roebbers, Johan Sunter, Peter Welch, and David Wood (Eds.)
IOS Press, 2005

Assessing Application Performance in Degraded Network Environments: an FPGA-based Approach

Mihai IVANOVICI [a,1], Razvan BEURAN [a] and Neil DAVIES [b]
[a] *CERN, Geneva, Switzerland*
[b] *Predictable Network Solutions, Bristol, UK*

Abstract. Network emulation is a technique that allows real-application performance assessment under controllable and reproducible conditions. We designed and implemented a hardware network emulator on an FPGA-based custom-design PCI platform. Implementation was facilitated by the use of the Handel-C programming language, that allows rapid development and fast translation into hardware and has specific constructs for developing systems with concurrent processes. We report on tests performed with web-browsing applications.

Keywords. Application performance assessment, network emulation, FPGA, hardware implementation, Handel-C

Introduction

Network emulation is a technique that makes it possible to assess real-application performance under controllable and reproducible conditions. This hybrid technique combines the advantages of network simulation with those of tests in real networks [1]. Most of the existing network emulators are implemented in software, therefore the quality degradation they introduce is imprecise and unreproducible. Current hardware [4], [5], [6], [7] and software [2], [3] approaches exhibit an additional important drawback: they all introduce unrealistic degradation. The reason is twofold: packets in a flow are treated independently, and quality degradation effects are not correlated (e.g., packet loss and delay are independent).

We designed and implemented a hardware network emulator on an FPGA-based custom-design PCI platform [8], [12]. This ensures high-accuracy emulation, as well as high performance: the system supports packet rates up to 1.5 million packets per second in each direction. In addition, our implementation is a new approach to the emulation of quality degradation in networks, permitting reproducible experiments in realistic network scenarios. In this approach, network conditions are described in terms of network-element behavior models, which are aggregated into a single representation that is used effectively for emulation. Implementation was facilitated by the use of the Handel-C programming language [9], that allows rapid development and fast translation into hardware. This article focuses on the basic principles of our methodology and the architectural choices in the emulator design, including an assessment of the costs and benefits of the choices we made.

CERN [10] collaborates with Predictable Network Solutions [11] to develop the emulator as a tool permitting the quantitative evaluation of the influence of the experienced network quality degradation on distributed application performance. We studied several network ap-

[1]Corresponding Author: *Mihai Ivanovici, CERN, 1211 Geneva 23, Switzerland.* Tel.: +41 22 767 39 08; Fax: +41 22 767 39 00; E-mail: `Mihail.Ivanovici@cern.ch`.

plications used in domestic and specialized environments that would benefit from assurance of bounds on quality degradation. We report on the behavior of short-lived TCP transfers as occurring in the case of web-browsing applications. We conclude that our approach is suitable to the assessment of applications and approaches to help deliver network-based services that are predictable, a prerequisite for the support of safety-critical services.

1. Application Performance Assessment

A key issue in application performance assessment is the understanding of the fact that network environments perturb application behavior by delaying and dropping the application traffic. Networks are therefore degraded environments, and quality degradation in the network is reflected in the performance degradation at application level.

1.1. Principles

There are three steps to take in order to assess application performance: (i) observe the application behavior at the end-node level, (ii) accurately measure the quality degradation experienced by the application traffic and (iii) correlate the above. A general setup is depicted in Figure 1.

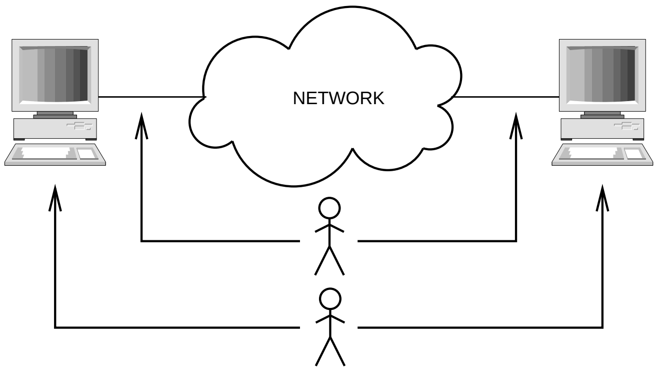

Figure 1. Observing end-to-end application performance and measuring the quality degradation in the network.

Scientific method requires the use of objective metrics to perform both the network and application level performance assessments. In case of network quality degradation there is already a series of widely used metrics [15], [16]: one-way delay [17], one-way packet loss [18] and throughput. However, when application performance must be determined, each application class requires the definition of specific metrics that take into account the application nature. For example, for Voice over IP (VoIP) one can use the Perceptual Evaluation of Speech Quality score [19]. In case of file transfer, useful metrics are transfer time performance and goodput [20].

1.2. Our Approach

There is an issue with the setup in Figure 1: one has no control over the degradation introduced by the network. Other people's traffic and the loss and delay it induces are what they are at that moment. Consequently, any measurement only reflects the conditions at that particular time of day.

A much more practical approach is to use a network emulator instead of the real network. This allows for varying network conditions in a controllable and reproducible manner, hence effectively exploring application performance in a much wider range of conditions. Figure

2 shows the experimental setup we used. Quality degradation, denoted by ΔQ, is correlated with the User-Perceived Quality (UPQ) for the application under test.

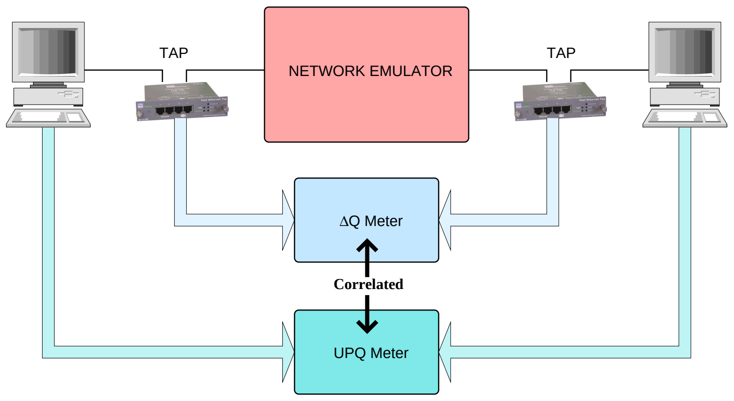

Figure 2. Experimental setup.

There are several emulators available at the moment, and we had hands-on experience with one of them—the results obtained with VoIP and file transfer applications were already reported in [1].

One of the problems of many network emulators, especially the software versions, is the lack of accuracy of the degradation they induce. Although it could seem that this is not essential from the point of view of delay (as long as the error is within reasonable bounds) it becomes important when packet loss is concerned. This is because packet loss is the result of a critical race for resources. The fact whether packet loss occurs or not at a certain moment depends on the relative timing of the packets. Accurate emulation of these effects is needed in order to get correct loss rates and distributions, a mandatory feature of an emulator since they are critical for the performance of most applications.

A hardware implementation, such as ours, ensures a correct behavior regarding timing. But current hardware emulators have another drawback: they all introduce unrealistic degradation. This comes from the approach to emulation that is generally taken. Firstly, packets in a flow are treated independently, which may lead to packet reordering within the same stream. This has a disastrous effect on TCP application performance, which is optimized for the normal case when packets arrive in order—we already encountered such problems in real networks during the performance measurements that were part of feasibility studies related to the ATLAS[1] Event Filter at CERN. Secondly, in current network emulators, quality degradation effects are not correlated (e.g., packet loss and delay are independent). By allowing the delay and loss distributions to be configured independently, the natural dependence that exists between these two parameters is destroyed.

Moreover, most of the existing network emulators are network-topology oriented. They use a node-by-node representation of the emulated network. However, this approach becomes unfeasible for large-scale networks. We believe that the network quality degradation should be emulated by using compact models of the network. These models are obtained by aggregating simple network elements into an object with equivalent observable properties. In this simple way we address the design shortcomings of the existing approaches and also achieve high accuracy through a hardware implementation.

We identified two basic elements that are the building blocks of any network system: the *wire* and the *queue*. The *wire* represents the transmission media, which can be considered, in a first approximation, error free. Therefore its main characteristic is the constant propagation

[1]ATLAS is one of the four experiments being built at CERN on the Large Hadron Collider accelerator.

delay. The *queue* is characterized by its length and service rate. It introduces variable delay and loss. This degradation is introduced by the intra-stream and inter-stream competition for resources: a packet competes both with other packets from the same traffic flow, as well as with packets from other streams. Hence the need arises to emulate the inter-stream competition, for which task we found two techniques. The first one is to use background traffic generation so as to artificially consume resources (queue space and service time). The second technique is to use the "server with vacations" paradigm, in which the queue server takes vacations that correspond to servicing the other streams.

Since any network emulator is in fact a system that introduces quality degradation in the network, from now on we will use the term "quality degrader" to refer to it.

2. Quality Degrader Architecture

Our quality degrader is implemented in hardware, in order to achieve high accuracy and packet rates up to 1.5 million packets per second. This section presents the hardware platform and provides details about the core of the emulator.

2.1. Hardware Platform

Our implementation is based on a custom-design PCI platform [8], [12]. This hardware platform uses one Altera Stratix FPGA (EP1S25F780C7) with 25 k logic elements, two Gigabit Ethernet RJ45 ports and memory (128 MB SDRAM and 1 MB SSRAM). The schematic is presented in Figure 3:

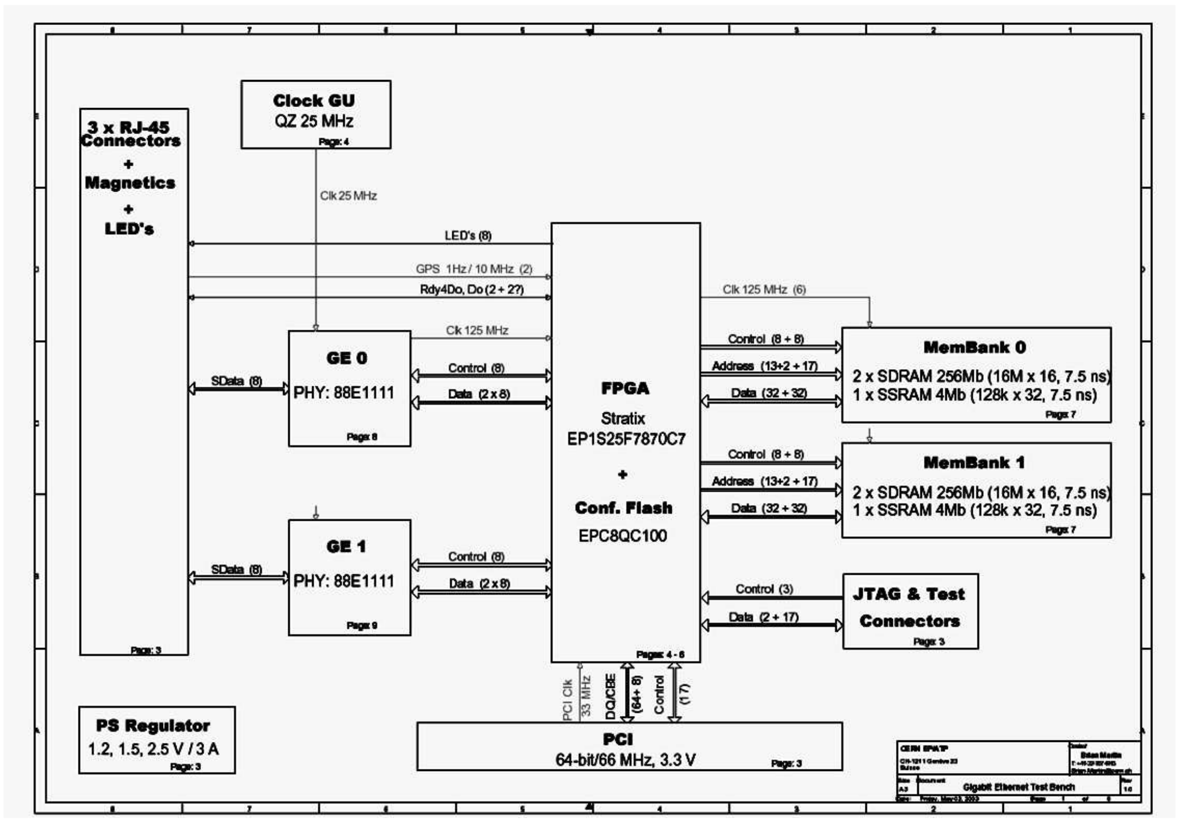

Figure 3. Schematic of the hardware platform.

The FPGA manages all the resources and contains the IP[2] cores for the two Ethernet MACs (Medium Access Control) and the SDRAM and PCI controllers. The user-defined higher-level functionality is implemented on the same FPGA, using the Handel-C programming language. A low-level library that provides primitives for memory and PCI access was created. The dual-port on-chip RAM blocks provided by the Stratix FPGA are extensively used to implement queues between concurrent processes. The FIFO paradigm ensures that messages/data are processed in order and in the same time decouples through buffering the two processes that access the two ends of the queue. Of course data is lost if the queue fills, but this should not happen during normal operation and is an indication of a malfunction.

The SDRAM is used to store packet data temporarily. The SSRAM is used to store the configuration of the emulator. Packet data flows between the two Gigabit Ethernet ports, allowing for the seamless integration of the platform into a network, a prerequisite for a network emulator. To facilitate the deployment, the board can be hosted by ordinary PCs owing to its standard 3V3 PCI connector. The PCI is used to configure the application firmware and to collect statistics.

The whole system is driven at PC level by a Python-based [13] control system. This allows the creation of automated procedures for performing experiments in a very simple and flexible manner. The low-level communication with the hardware platform via PCI is ensured by a custom Linux kernel module [14].

2.2. Internal Architecture

The architecture of the network emulator is depicted in Figure 4. We used thick arrows to represent the data paths and thin arrows for the control paths. The architecture is based on modules, that are blocks of code that have a specific functionality, and which run concurrently. Each module consists of several parallel processes, and communicates with other modules and internally by means of channels.

The behavior of the emulator is briefly the following. The traffic is first classified in order to enable different degradation to be applied to it. Quality degradation is effectively introduced by means of a system of queues. The length of each queue can be specified. We chose the approach of background traffic generation, which was implemented as part of the Degradation Emulation Engine module described below. Packets are then serviced according to one of the two scheduling algorithms, Strict Priority (SP) or Weighted Round Robin (WRR). The weights for WRR are user configurable. Next we'll describe in more detail each of the modules.

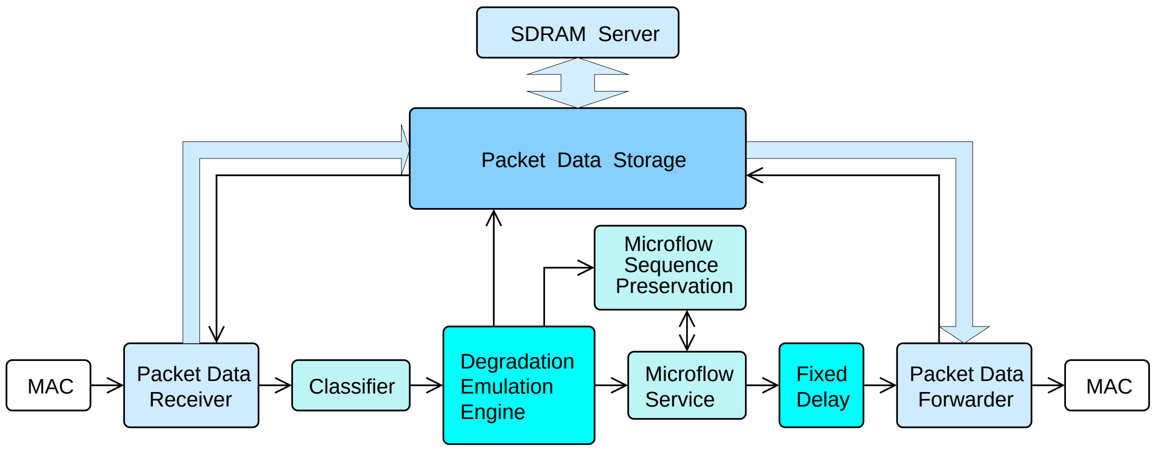

Figure 4. Quality degrader architecture.

[2]IP stands here for Intellectual Property.

The MAC Receiver module provides packet data to the Packet Data Receiver, which configures and controls it. The Packet Data Receiver checks with the MAC Receiver whether there was any error on reception, and discards the packet data if this is the case. Note that the MAC Receiver is an IP macro.

The Packet Data Receiver module manages the reception of complete packets from the MAC interface. The Packet Data Receiver maintains two queues—one for storing full packet data and another one for packet descriptors. If one packet is correctly received, a descriptor is placed in the descriptor queue.

The packets are then sent to the Packet Data Storage module, which returns the memory address where the packet data is stored. The address of the next available memory slot is determined using a bitmap representation of the memory occupation. Memory is divided in slots of 2048 bytes; slots are seized when a packet is received and freed when the packet is either discarded internally or transmitted. The memory address of the current packet and its descriptor form a packet reference. The packet reference along with information extracted from the IP packet header are sent to the Classifier module. The following fields are retained: protocol number, Type Of Service (TOS), source and destination IP addresses.

The Classifier module classifies packets based on the retained fields forwarded by the Packet Data Receiver. Once packets have been classified, the corresponding packet reference and the identifier specifying the degradation queue are sent to the Degradation Emulation Engine module.

The Degradation Emulation Engine module induces quality degradation through the management of a system of queues. Upon receiving a packet reference and a degradation queue identifier, the process sends them to Microflow Sequence Preservation for enqueueing. Under certain conditions the decision to immediately discard the packet can be taken; in this case no enqueueing takes place. The next queue to be serviced is determined based on a scheduling algorithm (SP or WRR) and is indicated to Microflow Service. An essential feature of this module is the background traffic generator associated with each queue. These generators can be independently started/stopped and configured to transmit user-defined artificial traffic patterns. The patterns are uploaded onto the board in the on-chip RAM. So far we used CBR (Constant Bit Rate) and Poisson distributions for the inter-packet arrival time of the background traffic packets. The Degradation Emulation Engine module also implements the transmission-rate limiting mechanism.

The Microflow Sequence Preservation module stores and manages in a FIFO manner the packet references received from the Degradation Emulation engine, thus preventing packet reordering. The Microflow Sequence Preservation module uses eight queues, out of which one is assigned to the unclassified traffic. The size of each queue can be configured.

The Microflow Service module manages the retrieval of packet references from Microflow Sequence Preservation based on queue identifiers received from Degradation Emulation. The packet reference is subsequently sent to the Fixed Delay module.

The Fixed Delay module introduces a constant delay by enqueueing the descriptors in a queue. The constant delay represents the propagation delay and it is user configurable.

The Packet Data Forwarder module manages the transmission of the packets to the MAC interface. It maintains two queues—one for full packet data and another one for packet descriptors. When a packet reference is received, the corresponding packet is retrieved from the Packet Data Storage. Once a packet is transmitted, a “free reference” message is sent to Packet Data Storage.

The MAC Forwarder module receives packet data from the Packet Data Forwarder in chunks of 32-bit words. The MAC Forwarder is configured and controlled by the Packet Data Forwarder. As the MAC Receiver, the MAC Forwarder is an IP macro.

2.3. Implementation Philosophy

For a design of this complexity the obvious description language of choice would be VHDL; however the learning curve for non-hardware specialists was estimated as being too steep. We chose instead Handel-C, a language whose close affinity to C made it readily accessible to software engineers while at the same time offering constructs that allowed us to employ the natural parallelism of the hardware. Handel-C had been employed before in networking applications at CERN [21] but the magnitude and complexity of this design would present a real challenge.

The most obvious construct of use is the `par`, which allows for the parallel execution of statements or complete processes. In practice every module shown in Figure 4 runs as a process under a top-level `par`. There are limits with `par`, however, as was found when trying to parallelize the packet classifier. If the different classification rules were operating in parallel then each process would attempt to access the data at the same time, meaning significant FPGA routing problems and increased delay. In addition this solution doesn't scale to a large number of rules. Replicating the data to provide each process with its own copy was equally time consuming, so finally it proved better to execute classification rules sequentially.

Channel objects allow data to be communicated between processes; the transfer occurs only when both processes are ready and forces the synchronization of parallel processes. Their existence proved especially useful for several reasons. During the development phase, we could independently debug and validate individual modules comfortable in the knowledge that whatever logical or timing changes happened as a result, the module would still fit back into the full design and communicate with its adjacent modules as before. If, in the worst case of a design error, there is some channel mismatch then the whole system freezes and allows the debugger to retrieve state and correct the error. This independent development is especially powerful when one considers that for the full design a compile, place and route cycle is at least half an hour.

Channels also resolve the problem of passing data across clock domain boundaries. The design can't avoid different domains since the PCI interface requires 33 MHz and the MAC interface requires 25 MHz. However having the facility to easily cross clock domains meant that we could choose something close to the optimum frequency for several different tasks. For example, the SDRAM server runs at 83.3 MHz and the main core at 62.5 MHz. Without this option we would not have been able to meet the design requirements of the project. Again however there are limits. The channel is a complex structure with up to a four-cycle overhead which becomes the limit for very high speed transfers. For the fastest transfer logic we needed to use an internal hardware feature, the dual-port RAM, as shared memory between two clock domains. The shared memory acts as a mailbox while the requests for transfer are still sent over channels. This came at the cost of having to ensure the synchronization with our own logic, an error-prone process that cost considerable debugging time.

Although ordered and synchronous operations have clear advantages, there are cases where data has to be retrieved as fast as possible—such as the ingress from the MAC interfaces which must be cleared irrespective of the state of the modules that will consume the data. We used queues in this case to accept asynchronously the incoming data.

The use of multiple channels in each module lead to yet another problem: the arbitration of the access to all these channels. Handel-C provides a solution by means of the `prialt` instruction. This instruction allows the channel that is first ready to perform a transfer and it even works between different clock domains. This instruction was used, for example, in the Packet Data Storage module, to which "free reference" messages are addressed from more sources (from Degradation Emulation Engine on packet discard, and from the Packet Data Forwarder module on packet transmission).

In retrospect the choice of language for this project was fully justified. It allowed for a formal approach to the design process and we found that with each iteration we moved further from the shared memory architecture and closer to channel based ones that facilitated both debugging and execution. As we became more competent with using channels and CSP, the code we wrote became smaller and simpler. This leads to the fact that maintenance and modifications are also easier.

3. Experimental Results

Using the network emulator we assessed the performance of several applications, such as web browsing, file transfer, VoIP, and video streaming. Web browsing is an HTTP-based application that is characterized by short-lived TCP transfers. The performance of such an application strongly depends on packet loss, hence we chose to present the results we obtained using it. The traffic of interest (HTTP) competes with the background traffic to occupy queue space—which induces loss—and for being serviced—which induces delay. We compare two cases, when the background traffic source has a CBR or a Poisson pattern. For all the tests the emulator was configured to introduce a fixed delay of 12.5 ms (equivalent to 25 ms RTT[3]). The available bandwidth was limited to 10 Mb/s and the size of the queue was 128 packets.

We used the setup presented in Figure 2. The end PCs run Linux with kernel 2.4.21, the HTTP server was Apache 2.0 (httpd-2.0.46) and the client was `wget` (wget-1.8.2), a non-interactive network retriever that allows for the automation of tests. The interconnect employed was Fast Ethernet, because the taps we currently use work only at 100 Mb/s. The emulator is however able to run at 1 Gb/s as well.

For the Apache server all the parameters had default values, including the *Timeout*[4] of 300s. *KeepAlive*[5] was "on" and "off" in turn. When "off", a new TCP connection is opened and closed for each file transfer. This represents the most inefficient case. When "on", the same TCP connection is used for up to *MaxKeepAliveRequests* = 100 transfers separated by no more than *KeepAliveTimeout* = 15 s.

We chose a representative web-page structure to use in our tests, that contains both images and text. The site consists of 499 files, with a total size of 1.6 MB. The average file size is approximately 3 kB, close to the average value of file sizes on the Web [22]. The results in Figures 5 and 6 show the dependency of site download duration on the offered background-traffic load, for *KeepAlive* "off" and "on", respectively. The site download duration is a measure of the user-perceived quality for web-browsing applications. The reference value is that obtained when the application has an exclusive use of the network (i.e., when the background traffic generator is disabled). The offered background-traffic load varies from 0 to 100%, being a measure of the congestion induced by the emulator.

Note in Figure 5 that CBR background traffic has almost no influence on the performance, since this case is equivalent to a constant diminution of the bandwidth available for the application. A constant amount of available bandwidth leads to a steady performance of TCP. Since web browsing only implies transfers of relatively small amount of data, the available bandwidth can be low without a significant impact on performance. When the background traffic load approaches 100%, the available bandwidth becomes insufficient. Subsequently there is a rapid increase of the download duration, followed by denial of service and leading to complete application failure.

[3]Round Trip Time, the time needed for a packet to travel back and forth between two end nodes on a particular network connection.

[4]*Timeout* is the number of seconds before the server receives and sends time out.

[5]*KeepAlive* indicates whether or not to allow persistent connections (i.e., with more than one request per connection).

When the background traffic is Poisson (and therefore more realistic) noticeable performance degradation starts occurring from loads of 60%. At loads larger than 80% the degradation becomes significant and reaches values with more than one order of magnitude higher compared to the CBR case. The intrinsic burstiness of the Poisson traffic determines the larger deviations of the results.

One can observe in Figure 6 an improvement of the worst-case behavior of one order of magnitude when *KeepAlive* is "on", due to the reutilization of the same TCP connection for multiple transfers. This reduces the probability of losing connection establishment and termination packets; such loss is the main culprit for performance drop for TCP-based applications.

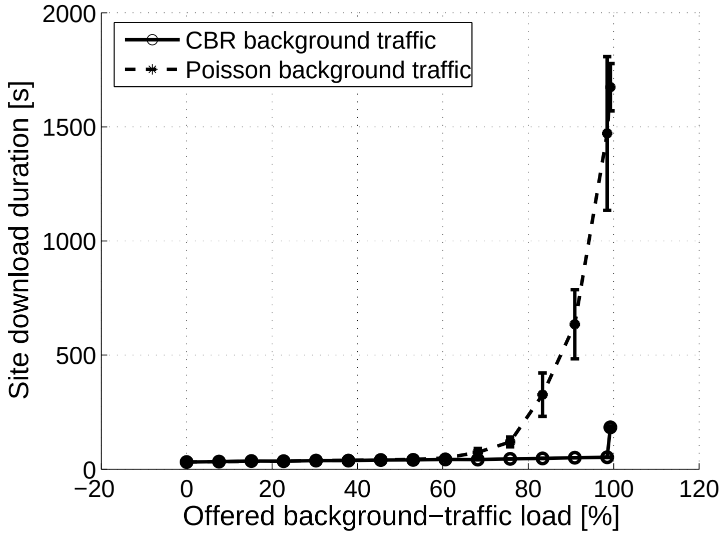

Figure 5. Site download duration versus offered background-traffic load (*KeepAlive* "off").

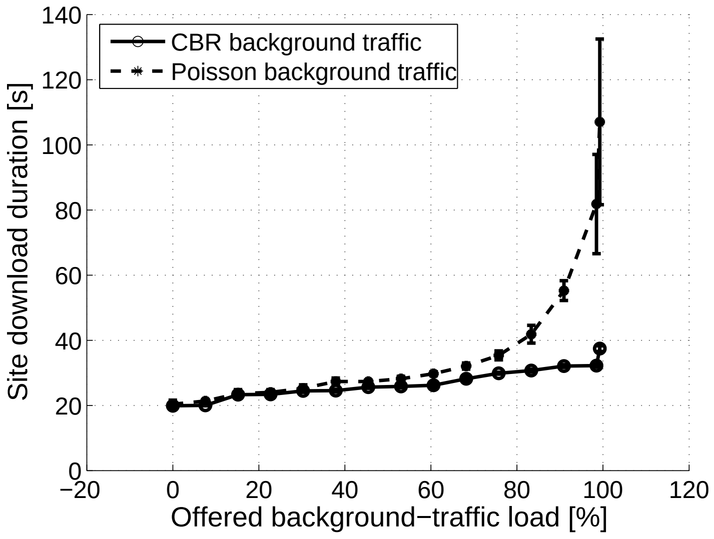

Figure 6. Site download duration versus offered background-traffic load (*KeepAlive* "on").

4. Conclusions

In this paper we present our approach to the emulation of quality degradation in networks. We implemented our concepts using an FPGA-based hardware platform. This proved to be the most appropriate solution for network emulation considering the requirements for accuracy, reproducibility and high-speed operation. The implementation was made easier by the use of Handel-C, a programming language that is rich in constructs that allow the development of concurrent process systems. We used in particular very intensively channels to synchronize the parallel processes in our implementation.

The core of the emulator is a system of queues which guarantees a realistic dependency between packet loss and delay. We identified two strategies for emulating the effects of other traffic flows on the traffic of interest: background traffic generation and the "server with vacations" paradigm. We have already implemented the first approach and we are currently comparing the two strategies. This will be useful for the next generation emulators, able to emulate large-scale networks through the aggregation of network element models.

We used this system for application performance assessment. We report on web browsing, characterized by many short-lived HTTP transfers. We determined experimentally the dependence between site download duration and the offered load of the background traffic, i.e., between user perceived quality and the congestion level in the network.

As mentioned before, we already run test with various other applications, such as VoIP (Voice over IP) and video streaming using a software network emulator. Our future plans is to perform similar tests using the hardware network emulator we developed in order to obtain more accurate results and emphasize the differences between the two emulation approaches.

In addition the emulator will be used to perform tests in connection with the design of the ATLAS Data Collection system at CERN. Preliminary tests have already taken place and we will report on them in a future paper.

Acknowledgments

The work presented here was supported by PPARC PIPSS Project No: PPA/I/S/2002/00653. We would like to acknowledge Brian Martin and Jaroslav Pech for the design of the hardware platform. We are grateful to Matei Ciobotaru for the development of the low-level access libraries. We would also like to thank Brian Martin and the anonymous CPA-2005 referees for their pertinent comments and suggestions.

References

[1] R. Beuran, M. Ivanovici, B. Dobinson, N. Davies, P. Thompson, "Network Quality of Service Measurement System for Application Requirements Evaluation" , International Symposium on Performance Evaluation of Computer and Telecommunication Systems, July 20-24, 2003, Montreal, Canada, pp. 380-387.
[2] "Dummynet: a simple approach to the evaluation of network protocols", L. Rizzo.
[3] NISTNet Network Emulator, `http://www-x.antd.nist.gov/nistnet`
[4] Simena, `http://www.simena.net`
[5] Anue Systems, `http://www.anuesystems.com`
[6] Empirix, `http://www.empirix.com`
[7] Shunra, `http://www.shunra.com`
[8] M. Ciobotaru, M. Ivanovici, R. Beuran, S. Stancu, "Versatile FPGA-based Hardware Platform for Gigabit Ethernet Applications" , 6th Annual Postgraduate Symposium, Liverpool, UK, June 27-28, 2005.
[9] Celoxica, `http://www.celoxica.com`
[10] CERN, The European Organization for Particle Physics, `http://www.cern.ch`
[11] Predictable Network Solutions, `http://www.pnsol.com`

[12] M. Ciobotaru, S. Stancu, M. LeVine, B. Martin, “GETB—A Gigabit Ethernet Application Platform: its Use in the ATLAS TDAQ Network”, Real Time 2005, Stockholm, Sweden, June 10, 2005.
[13] The Python Programming language, `http://www.python.org`
[14] M. Joss, “IO_RCC—A package for user level access to I/O resources on PCs and compatible computers”, CERN, Technical report ATL-D-ES-0008, October, 2003.
[15] ITU-T Recommendation I.380, “Internet Protocol (IP) Data Communication Service—IP Packet Transfer and Availability Performance Parameters”, ITU-T, February, 1999.
[16] V. Paxson, G. Almes, J. Mahdavi, M. Mathis, “Framework for IP Performance Metrics”, IETF RFC 2330, May, 1998.
[17] G. Almes, S. Kalidindi, M. Zekauskas, “A One-way Delay Metric for IPPM”, IETF RFC 2679, September, 1999.
[18] G. Almes, S. Kalidindi, M. Zekauskas, “A One-way Packet Loss Metric for IPPM”, IETF RFC 2680, September, 1999.
[19] ITU-T Recommendation P.862, “Perceptual Evaluation of Speech Quality (PESQ), An Objective Method for End-to-end Speech Quality Assessment of Narrow-band Telephone Networks and Codecs”, ITU-T, February, 2001.
[20] R. Beuran, M. Ivanovici, V. Buzuloiu, “File Transfer Performance Evaluation”, Scientific Bulletin of University “POLITEHNICA” Bucharest, C Series (Electrical Engineering), vol. 66, no. 2-4, 2004, pp. 3-14.
[21] F. R. M. Barnes, R. Beuran, R.W. Dobinson, M.J. LeVine, B. Martin, J. Lokier, and C. Meirosu, “Testing Ethernet Networks for the ATLAS Data Collection System”, IEEE Trans. Nucl. Sci., Vol. 49, No. 2, April 2002, pp. 516-520.
[22] M. F. Arlitt, C. L. Williamson, “Web Server Workload Characterization: The Search for Invariants”, Proc. SIGMETRICS, Philadelphia, PA, USA, April, 1996.