MMPP-based HTTP traffic generation with multiple emulated sources*

Grzegorz Hryn
Silesian University of Technology
Institute of Computer Science
grzegorz.hryn@polsl.pl

Zbigniew Jerzak
Dresden University of Technology
Systems Engineering Group
zbigniew.jerzak@inf.tu-dresden.de

Andrzej Chydzinski
Silesian University of Technology
Institute of Computer Science
andrzej.chydzinski@polsl.pl

3rd November 2004

Abstract

In this article we propose a new tool, named Raw Packet Sender (RPS), for testing the performance of WWW servers. Our solution allows for testing with arbitrary number of source IP addresses although the traffic originates from only one physical NIC. In order to better mimic the real life environment we implemented an HTTP session interarrival time generator based on Markov Modulated Poisson Process (MMPP), which can closely match auto-covariance and the marginal distribution of recorded web traffic traces.

1 Introduction

One of the most important tasks of web site administrators is to assure fast and reliable service to satisfy their clients. This goal can be accomplished by reduction of the "time to glass" factor – the delay between a request sent by a user and the subsequent delivery of the content and displaying it in a user's browser. In our previous work [5, 4] we presented a new approach to identification of incoming clients that suffer from a poor performance. Our solution, named *VisProxy*, is based on a proxy server forwarding client's request to

*This material is partly based upon work supported by the Polish Ministry of Scientific Research and Information Technology under Grant No. 3 T11C 014 26.

a Web Server storing a content that is appropriate for the client's connection capabilities. This mechanism allows for improvement of a user-perceived performance.

In order to test classification capabilities and performance of the *VisProxy* we have developed a tool, named *Raw Packet Sender (RPS)*, which allows us to generate a network traffic imitating a stream of HTTP requests originating from a population of web clients. The most important feature of *RPS* when testing *VisProxy* is the *RPS* ability to emulate traffic coming from numerous IP addresses using only a limited set of test computers. This in turn allows us to verify the correctness of measurements of sessions coming from different clients (i.e. computers having different IP addresses) and forwarded by *VisProxy*. Setting up such an environment would be impossible in a laboratory since most of the tests involves requests coming from hundreds of thousands of virtual IP addresses.

During the course of our tests we encountered a need to better emulate the real-life behaviour of the HTTP traffic, as the constant request rate generator does not mimic the real life traffic pattern. As it has been confirmed [14], there exists a variety of methods that emulate real network traffic in realistic way (to some extent), hence allowing for introduction of such characteristics as self-similarity, burstiness and long-range dependence [1, 7] in a test environment. We have decided to adopt these methods in our prototype of *RPS*.

2 Background and Related Work

As mentioned before during our work we have investigated many tools that were designed for the purpose of testing the HTTP servers. We have started with the *httperf* [8] and then extended it with the *autobench* [6] allowing for distributed tests. *httperf* suffers from one main problem – it allows only constant rate tests or tests with exponential distribution of sessions. Moreover, *httperf* gathers statistics only on the overall level – e.g. no detailed information concerning the single packet arrival times is given. There is also a number of testing solutions from other companies (*WebBench* and *WebStone*), however they all seem to suffer from the same problems as *httperf*. None of the solutions mentioned before allows for dynamic session generation based on MMPP. Recently there has also been developed an MMPP-based traffic generator [9], however its use has been limited only to the ns-2 network simulator environment and used to model the TCP traffic at the Internet edge routers.

3 System Architecture

In this chapter we describe the architecture of our software tool – *Raw Packet Sender (RPS)*, prototyped in C++. *RPS* is able connect to a web server mimicking clients with different IP address. With almost arbitrary number of such clients[1] we can stress the target system simulating connections from hundreds or thousands of network adapters with only one physical NIC. However, it has to be noted that such tests do not cover all

[1] limited only by number of available IP addresses in a C class IP network

the aspects of the normal network traffic since all the connecting artificial clients share the same connection parameters - time to live, bandwidth and round trip time. In order to enhance the measurement precision we decided to use two computers for the RPS tests – see Figure 1. First one - the *RPS Init*, initiates the HTTP sessions with the server. Second computer, the *RPS Stat*, takes over the sessions initialized by the *RPS Init* and collects the statistics for each of them. RPS can work in two modes:

- constant rate load generation
- MMPP-based load generation

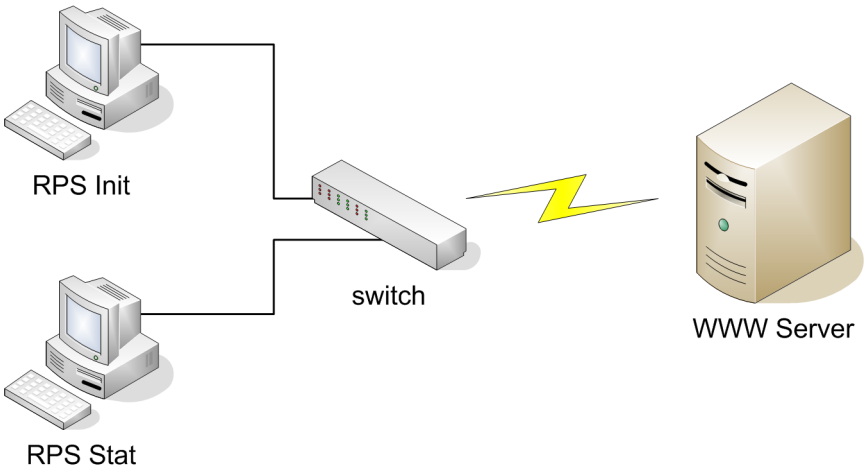

Figure 1: RPS System Architecture - the testbed overview

During constant rate load generation, HTTP sessions are initiated with an equal time intervals between them, whilst during MMPP-based load generation HTTP sessions are initiated with frequency obtained from the MMPP generator.

3.1 Networking Background

Networking part of our solution is based on the WinPcap Packet Capture Library [3, 12] a free, open source alternative to libpcap. In order to implement the simulated connections from one NIC with multiple source IP addresses we had to implement the ARP and IP spoofing mechanisms. These technologies allow a single NIC to pretend to be a source for packets with different source IP addresses and identical source hardware addresses - compare Figure 2.

ARP is a protocol used for pairing the TCP/IP stack IP addresses with the MAC addresses of network interfaces [11]. The ARP protocol bases on the simple mechanism of request and reply message exchange. Machine willing to send an Ethernet packet in TCP/IP network knows the destination machine's IP address and needs to obtain its MAC address so as to be able to create appropriate Ethernet packet header. In order

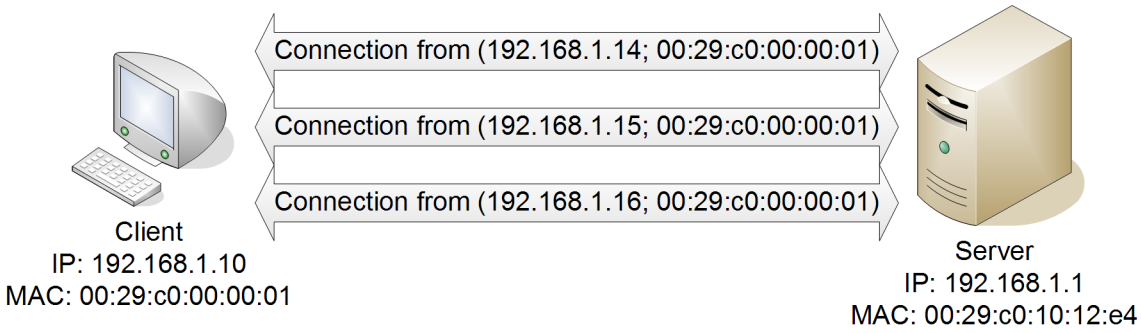

Figure 2: IP spoofing

to obtain this address it sends a multicast Ethernet packet containing the ARP request message asking for the MAC address of the known IP. Machine with the corresponding IP address sends a unicast ARP reply message with its MAC address to the querying host. In order to implement the IP spoofing mechanism we create custom IP packets and provide appropriate ARP replies for the server so that it can pair the virtual clients' IP addresses with the single MAC address of the physical sender. We have also implemented a TCP/IP stack allowing us to maintain complete spoofed TCP/IP sessions.

RPS internal architecture is split among the two test machines as follows. One process resides on the *RPS Init* machine. It spawns one thread which is responsible for creating and sending TCP SYN packets with the frequency obtained from the generator – either an MMPP-based or constant rate one. In order to synchronise the sending times of packets *RPS* uses the WinPcap kernel level mechanism which allows to prefill the send queue with the packet data and corresponding timestamp. When the queue is filled it is transferred into OS kernel where brute-force loops enforce the relative time dependencies between timestamps of concurrent packets. Such mechanism allows for achieving a granularity of 5 [μs]. Since synchronization is achieved by means of brute-force loops it causes large amounts of processor time to be consumed. Therefore this part of the testing environment has been placed on a separate machine.

Second test machine (*RPS Stat*) runs a process responsible for maintaining the HTTP sessions and storing the sessions' statistics. This process consists of one Manager thread and several Worker threads - see Figure 3, functions of components are explained in Table 1. The Manager thread is responsible for launching and administering all worker threads. First worker thread, the ARP Spoofing thread, is responsible for providing ARP Replies for the ARP Requests messages sent by the server. ARP Spoofing thread answers all server ARP requests that concern any of the IP's in the ARP Spoofing thread virtual IP address pool. The ARP Spoofing thread always replies with the MAC address of the machine it resides on. Second thread, the GET thread waits for the TCP SYN ACK message sent by the server. Upon detection of such message it sends the final ACK of the three way handshake along with the first GET HTTP message. This GET message refers to the base HTML page. The GET thread is also responsible for storing the time of sending the request in the shared memory. After receiving the requested base HTML

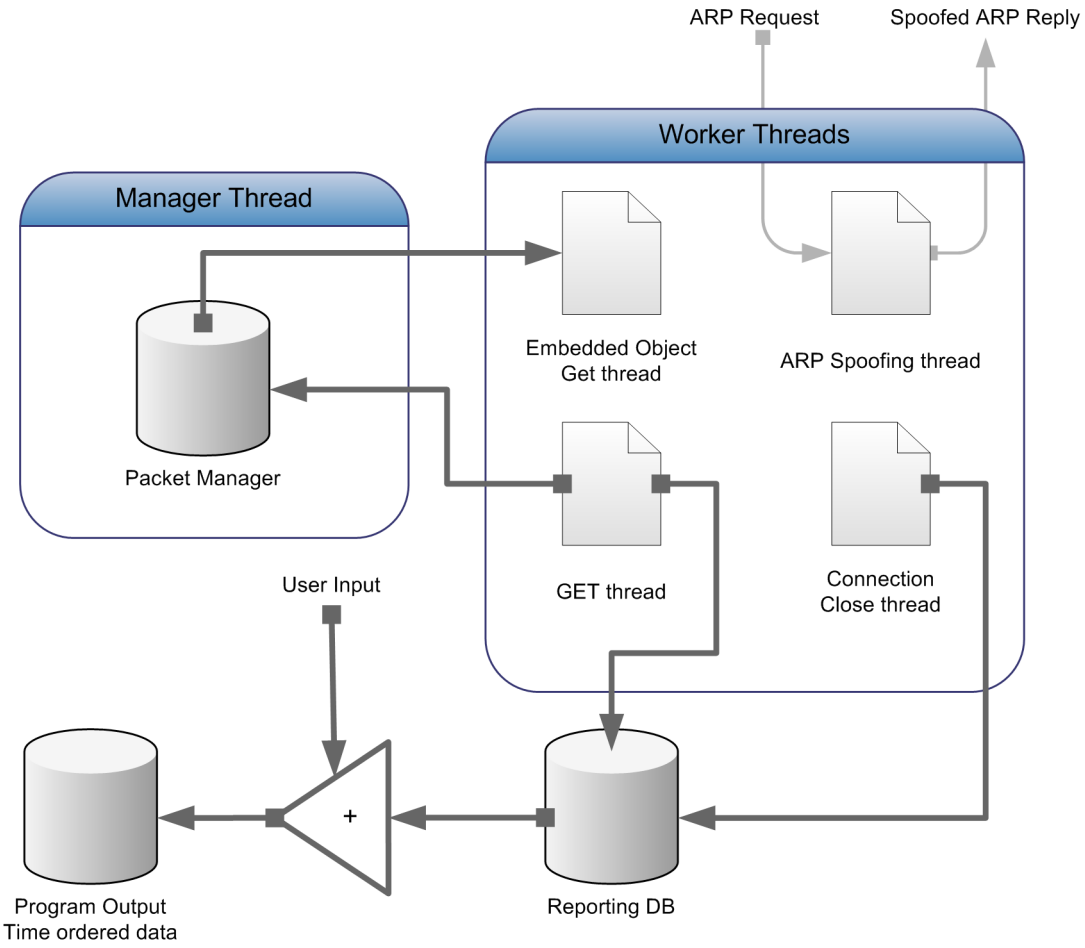

Figure 3: Thread diagram of the RPS spoofing process

page the Embedded Object Get thread confirms the received data and sends a request for the embedded object. When the server finishes transferring the data the TCP Connection Close thread closes the connection to the server and stores the connection close time in the shared memory. At any given moment during the test user can issue a command to the Manager thread which will parse the time statistics stored in the shared memory and will create a report in the textual format and store it on disk for further processing.

Name	**Function**
Packet Manager	Stores information about processed packets
Embedded Object Get thread	Waits for the server to deliver the HTML page and sends out request for the embedded object
ARP Spoofing thread	Responsible for providing spoofed answers to the IP MAC pairing requests
GET Thread	Confirms connection and send the first GET request for the base HTML page
Connection Close Thread	Terminates HTTP session and closes the connection after receiving the requested embedded object
Reporting DB	Stores session start and close times

Table 1: Object functions

3.2 Generating HTTP session traffic by means of MMPP

In order to imitate real network traffic we decided to implement a generator that produces load with the same statistical characteristics as in observed web traffic traces. Analysis of Internet traffic has shown that it is not Poisson but its correlation decays slowly with time and is visible over a few time-scales [10, 2]. As a result, peaks and dips have a tendency to group together and do not appear uniformly over the time scale, which can be easily observed in web server logs. Recent studies (see [16, 13, 15, 7]) have proven that a multi-state Markov Modulated Poisson Process (MMPP) is flexible enough to imitate various types of network traffic as it can closely match the observed autocovariance function and the marginal distribution. In particular, this property allows MMPP to simulate such phenomena as long-range dependence, burstiness and self-similarity.

In order to obtain parameters for the generator we have used a multiscale fitting procedure proposed in [13]. Authors use there a discrete time MMPPs (dMMPPs) defined as a Markov random walk in which the increments in each instant have a Poisson distribution whose parameter is a function of the state of the modulator Markov chain. Formally, a two-dimensional Markov chain $(X, J) = \{(X_k, J_k), k = 0, 1, \ldots\}$ with state space $\mathbf{N}_0 \times S$ is considered dMMPP iff for $k = 0, 1, \ldots$

$$P(X_{k+1} = m, J_{k+1} = j | X_k = n, J_k = i) = \begin{cases} 0, & m < n \\ p_{ij} e^{-\lambda i} \frac{\lambda_i^{m-n}}{(m-n)!}, & m \geq n \end{cases} \quad (1)$$

for all $m, n \in \mathbf{N}_0$ and $i, j \in S$. λ_i and i are non-negative real constants ($\lambda_i, i \in \mathbf{R}$) and $\mathbf{P} = (p_{ij})$ is a irreducible stochastic matrix. Whenever (1) holds, we say that (X, J) is a dMMPP with set of modulating states S and parameters $\mathbf{P}$ and $\mathbf{\Lambda}$, and write

$$(X, J) \sim dMMPP_S(\mathbf{P}, \mathbf{\Lambda}) \tag{2}$$

$\mathbf{P}$ is a transition probability matrix of Markov chain J, $\mathbf{\Lambda}$ is a matrix of Poisson arrival rates. When $S = 1, 2 \ldots, r$ where $r \in \mathbf{N}$, then

$$\mathbf{P} = \begin{bmatrix} p_{11} & p_{12} & \ldots & p_{1r} \\ p_{21} & p_{22} & \ldots & p_{2r} \\ \ldots & \ldots & \ldots & \ldots \\ p_{r1} & p_{r2} & \ldots & p_{rr} \end{bmatrix} \tag{3}$$

and

$$\mathbf{\Lambda} = \begin{bmatrix} \lambda_1 & 0 & \ldots & 0 \\ 0 & \lambda_2 & \ldots & 0 \\ \ldots & \ldots & \ldots & \ldots \\ 0 & 0 & \ldots & \lambda_r \end{bmatrix} \tag{4}$$

and we say that (X, J) is a dMMPP of order r, which we denote as $(X, J) \sim dMMPP_r(\mathbf{P}, \mathbf{\Lambda})$.

The inference procedure matches both the autocovariance and marginal distribution of the counting process by constructing MMPP as a superposition of L 2-MMPPs and one M-MMPP. The 2-MMPPs (each one models another time-scale of data) are used to match the autocovariance and the M-MMPP to match the marginal distribution. The procedure can be presented in four steps (see [13]):

1. Approximation of the empirical autocovariance by a weighed sum of exponentials and identification of time scales,
2. Inference of the M-dMMPP probability function and of the 2-dMMPP parameters,
3. Inference of the M-dMMPP Poisson arrival rates and transition probabilities,
4. Calculation of the final M2^L-dMMPP parameters.

The final MMPP with M2^L states is obtained by superposing the L 2-MMPPs and the M-MMPP in the following way. Denoting

$$(X^{(l)}, J^{(l)}) \sim dMMPP_2(\mathbf{P}^{(l)}, \mathbf{\Lambda}^{(l)}), \text{ where } l = 0, 1, \ldots, L, \tag{5}$$

$$(X^{(L+1)}, J^{(L+1)}) \sim dMMPP_M(\mathbf{P}^{(L+1)}, \mathbf{\Lambda}^{(L+1)}), \tag{6}$$

the result of a superposition is the following process:

$$(X, J) = \left(\sum_{l=1}^{L+1} X^{(l)}, \left(J^{(1)}, J^{(2)}, \ldots, L^{(L+1)} \right) \right) \sim dMMPP_S(\mathbf{P}, \mathbf{\Lambda}) \tag{7}$$

where

$$S = \{1,2\}^L \times \{1,2,\ldots,M\},$$
$$\mathbf{P} = \mathbf{P}^{(1)} \otimes \mathbf{P}^{(2)} \otimes \ldots \otimes \mathbf{P}^{(L+1)},$$
$$\mathbf{\Lambda} = \mathbf{\Lambda}^{(1)} \oplus \mathbf{\Lambda}^{(2)} \oplus \ldots \oplus \mathbf{\Lambda}^{(L+1)}.$$

and operations $\oplus$, $\otimes$ represent the Kronecker sum and product respectively. They are defined as:

$$C \otimes D = \begin{bmatrix} c_{11}D & c_{12}D & \ldots & c_{1n}D \\ c_{21}D & c_{22}D & \ldots & c_{2n}D \\ \ldots & \ldots & \ldots & \ldots \\ c_{n1}D & n_{n2}D & \ldots & c_{nn}D \end{bmatrix},$$

$$A \oplus B = (A \otimes I_B) + (I_A \otimes B).$$

One of the important advantages of the inference procedure is that the number of states is not fixed a priori but is automatically adapted by the fitting procedure to the particular trace being modelled. The fitting procedure uses the number of arrivals in a predefined sampling (time) interval hence produces a fixed amount of data that is known in advance which in turn allows for recording of longer traces.

In order to calculate the $\mathbf{P}$ and $\mathbf{\Lambda}$ parameters for our generator, we applied a fitting procedure implemented in a library (*dll*), provided by authors of [13]. We decided to use one of the network traces that are available on the Internet as a basis for inferring parameters for our generator. The chosen trace is an aggregated log from web servers hosting FIFA World Cup[2] (1998) web pages. As an input to the fitting procedure we used a data set consisting of a number of client arrivals in the predefined time interval – in our case it was 10 seconds. In order to differentiate between the concurrent session requests and new sessions we have decided to implement following algorithm. For each IP we browse the log backward from the occurrence of a request under question. If no prior requests from the same IP have been issued in a given amount of time, we assume that the request in question is a beginning of a new session. The input data set was generated from 48 hours observation time (starting from 10pm, 8th of June, 1998), which gave 17280 intervals. The total number of sessions observed was 562776 during that time.

The fitting procedure was applied to that set and the trace was fitted to a 14-dMMPP. We compared histogram of fitted model with histogram taken from trace data (Figures 4 and 5) and the autocovariance of the fitted model with the empirical autocovariance (Figure 6). All these figures shows that fitted model is able to reproduce very well the behavior of real session traffic taken from the server's log. For instance, average values for the distribution represented by Figs. 4 and 5 are 32.5699 and 32.5267 respectively. With regard to autocovariance the inequality coefficient defined as:

$$IC = \sqrt{\frac{\sum_{i=1}^{n}(x_i - y_i)^2}{\sum_{i=1}^{n} x_i^2}}$$

is equal to 2.35%, which is also a very good result.

[2]Soccer World Championships

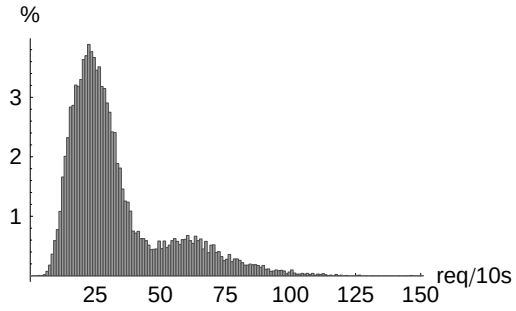

Figure 4: Trace data histogram

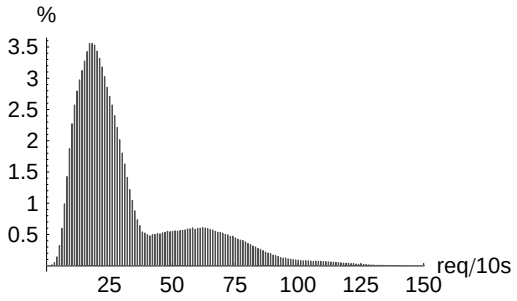

Figure 5: Fitted model histogram

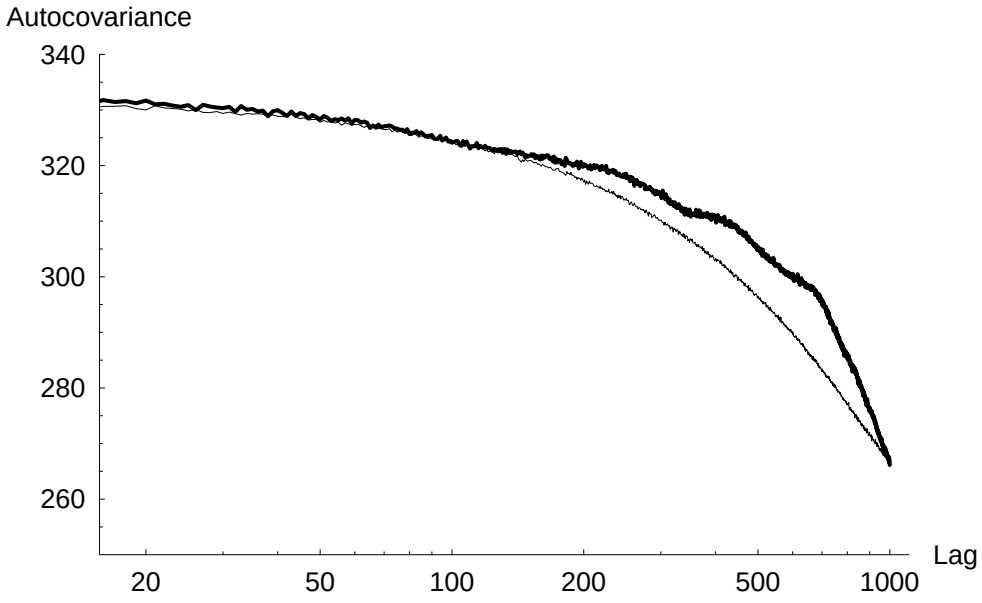

Figure 6: Autocovariance of fitted model (thin curve) versus empirical one (thick curve).

4 Evaluation and Testing

In order to evaluate our implementation we have conducted a series of tests in an isolated laboratory network environment. Testing environment consisted of two testing machines and one server machine. All computers were interconnected by a fast Ethernet switch. One of the testing machines was dedicated to run the *RPS INIT* process whilst the second one was hosting the *RPS STAT* process. The server machine (Dell Optiplex GX110 with 384 MB RAM and 800 MHz Intel Pentium 3) was running a Windows 2003 Server operating system and Internet Information Server 6.0. *RPS Init* computer is an IBM ThinkPad with 256 MB RAM and 1.4 GHz Intel Pentium M, *RPS Stat* computer is an IBM ThinkPad equipped with 256 MB RAM and 1.5 GHz Intel Pentium M, both laptops are running Windows XP Professional.

Our test suite consisted of two basic test sets. Each set covered test runs with different average count of request per second[3]. One of the test runs in each set was chosen so as to saturate the server and hence show how it behaves in overload condition. The two test sets are:

- tests with the *RPS* tool with MMPP option *disabled*
- tests with the *RPS* tool with MMPP option *enabled*

Such testing methodology allows us to compare the results obtained using constant rate testing with the varying source IP address and varying rate testing (MMPP distribution) with varying source IP addresses. To the best of our knowledge no tests with MMPP based distribution of the WWW sessions with varying source IP have been conducted so far.

4.1 Internet Information Server

The following table highlights the tests we have performed on the IIS 6.0 Web Server. All the time measurements are given in seconds. The measured response times are gathered at the client side and represent the time that has elapsed from issuing the request until receiving the complete response from the server. *min* denotes minimum observed response time, *max* denotes maximum observed response time, *avg* denotes mathematical average of the observed response times, *stdev* denotes standard deviation for *avg*, *med* denotes the median of the observed response times, *rate* denotes the request issue rate measured at the client side.

The following charts (presented on Figures 7, 8, 9 and 10) demonstrate the detailed results obtained during the tests. As it can be seen the MMPP tests stress the system accordingly with the expectations.

[3]In case of MMPP test it is not possible to determine the exact number of requests issued per second – hence the average estimation

test type	min [s]	max [s]	avg [s]	stdev [s]	med [s]	rate [1/s]
RPS (MMPP enabled)	0,0124	9,1170	0,0987	0,2652	0,0766	495
RPS (MMPP enabled)	0,0147	63,8051	4,4183	11,2706	0,1219	993
RPS (MMPP disabled)	0,0130	9,1196	0,1004	0,2618	0,0759	498
RPS (MMPP disabled)	0,0278	117,7428	12,1371	26,1527	0,1381	985

Table 2: Client response time measurements for the IIS 6.0 Tests

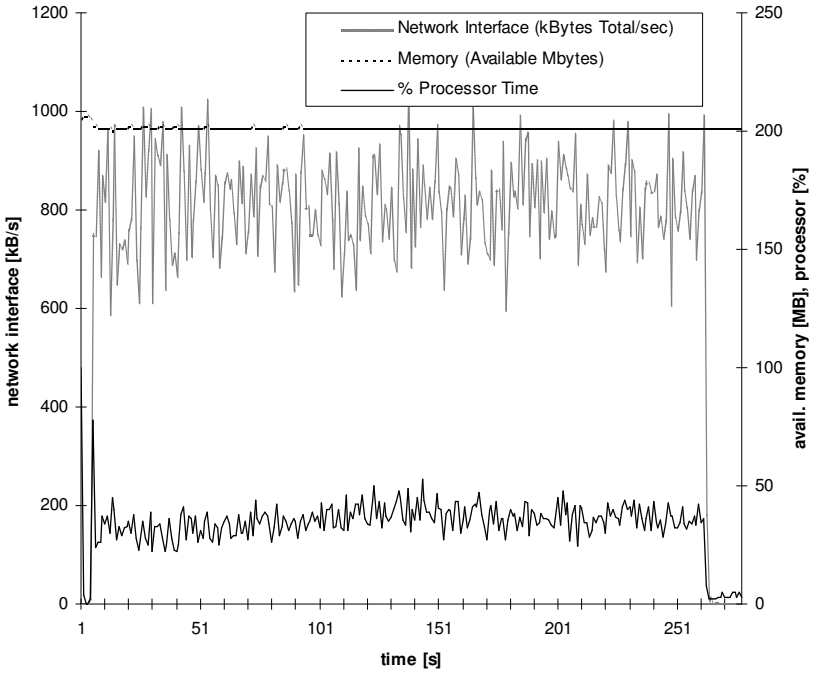

Figure 7: RPS test, MMPP enabled, 495 req/s

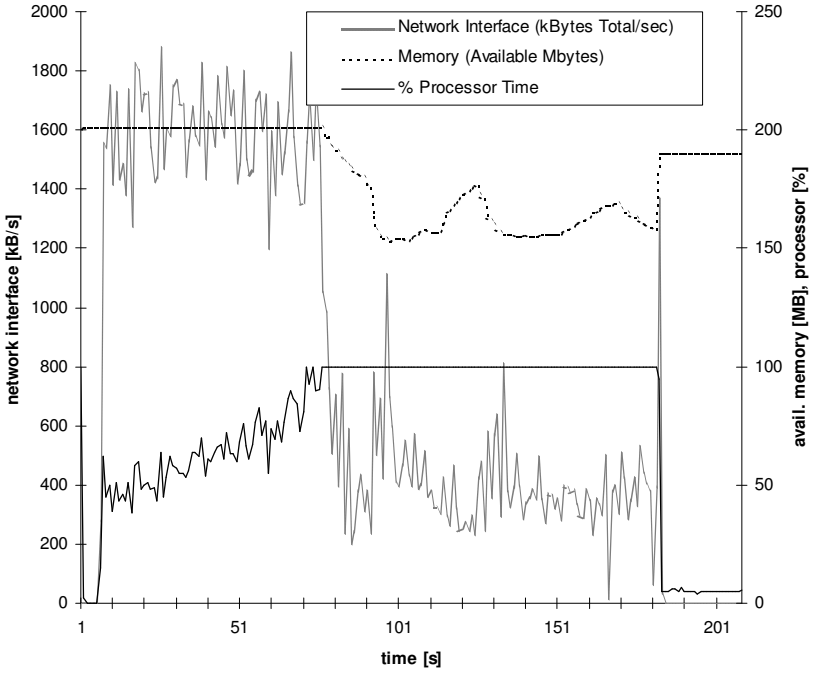

Figure 8: RPS test, MMPP enabled, 993 req/s

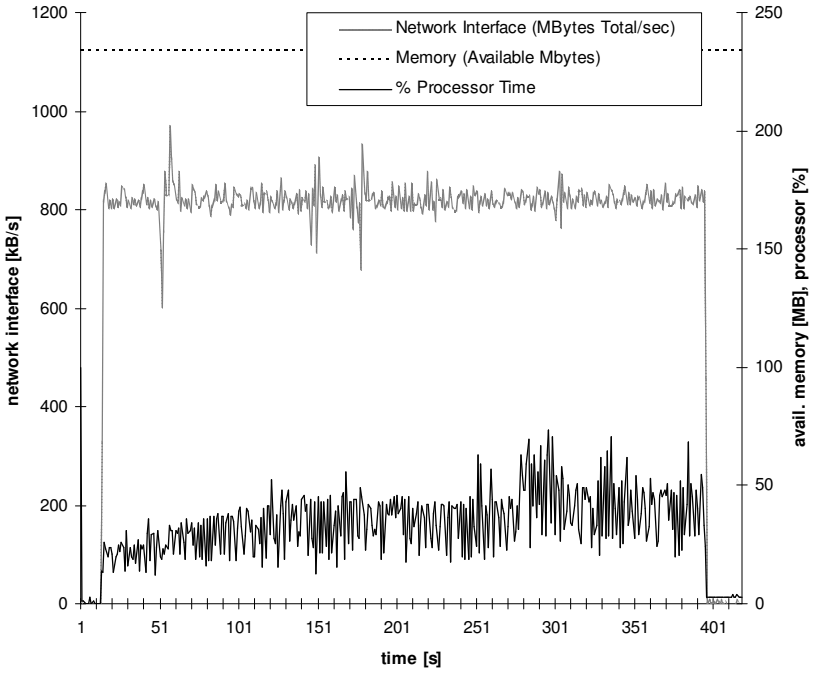

Figure 9: RPS test, MMPP disabled, 498 req/s

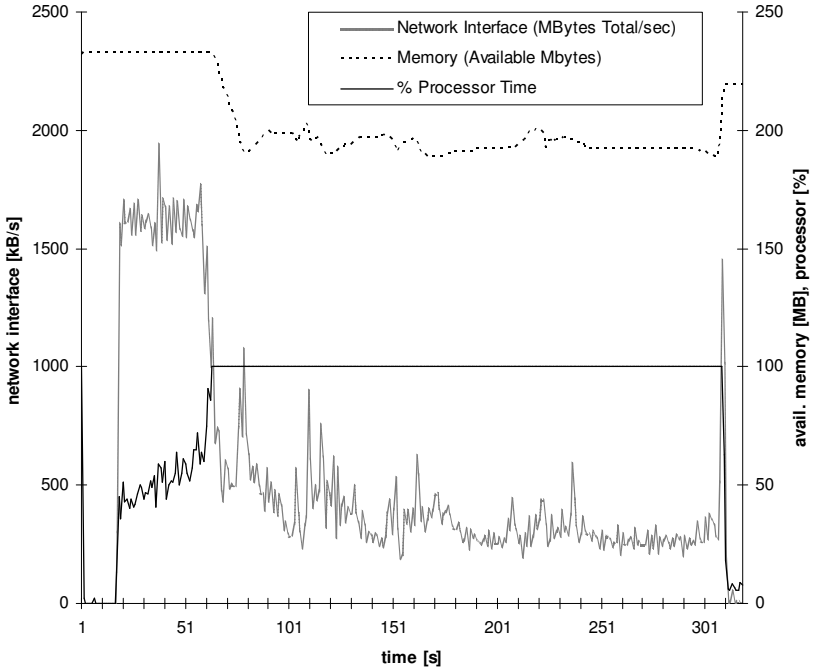

Figure 10: RPS test, MMPP disabled, 985 req/s

5 Conclusion and Future Work

Test results presented in this paper are promising and indicate that *RPS* with MMPP engine allows for stressing and obtaining performance characteristics from web servers. The MMPP workload resembles that of real life environment. We believe that MMPP tests are especially suited for stress testing servers and determining their behaviour under heavy load conditions. There is however, a number of improvements the *RPS* software could undergo – among them is the introduction of better distribution of tests – i.e. allowing for use of more than two PCs. During the course of our work with the *RPS* we have also noticed that a good management environment would be of an advantage for the administrator. Furthermore, releasing this software as open source would require more careful planning and envisioning of the potential extensibility at the early stages of design, which due to the nature of this software (experiment aiding tool) has been somewhat overlooked.

Current version of Raw Packet Sender is available online for download at the following address: `http://wwwse.inf.tu-dresden.de/rps/`

References

[1] Chen, X., Mohapatra, P., and Chen, H. An admission control scheme for predictable server response time for web accesses. In *10th World Wide Web Conference*

(2001), pp. 545–554. 1

[2] Crovella, M., and Bestavros, A. Self-Similarity in World Wide Web Traffic: Evidence and Possible Causes. In *Proceedings of SIGMETRICS'96: The ACM International Conference on Measurement and Modeling of Computer Systems.* (Philadelphia, Pennsylvania, May 1996). Also, in Performance evaluation review, May 1996, 24(1):160-169. 3.2

[3] Degioanni, L., Baldi, M., Risso, F., and Varenni, G. Profiling and optimization of software-based network-analysis applications. In *Proceedings of the 15th IEEE Symposium on Computer Architecture and High Performance Computing (SBAC-PAD 2003)* (November 2003). 3.1

[4] Hryń, G., and Jerzak, Z. Adaptive bandwidth driven content delivery for WWW clients. In *XII Konferencja Sieci i Systemy Informatyczne* (Łódź, Poland, 2004). 1

[5] Hryń, G., and Jerzak, Z. Możliwości poprawy postrzeganej jakości połączenia klientów WWW z wykorzystaniem pośredniczącego serwera proxy. In *Współczesne problemy sieci komputerowych - zastosowanie i bezpieczeństwo* (2004), WNT, pp. 465–474. 1

[6] Midgley, J. T. J. Autobench. 2

[7] Morris, R., and Lin, D. Variance of aggregated web traffic. In *INFOCOM (1)* (2000), pp. 360–366. 1, 3.2

[8] Mosberger, D., and Jin, T. httperf—a tool for measuring web server performance. In *Proceedings of the First Workshop on Internet Server Performance* (June 1998). 2

[9] Muscariello, L., Mellia, M., Meo, M., Cigno, R. L., and Marsan, M. A. A simple markovian approach to model internet traffic at edge routers. Tech. rep., Politechnico di Torino, May 2003. 2

[10] Paxson, V., and Floyd, S. Wide area traffic: the failure of Poisson modeling. *IEEE/ACM Transactions on Networking 3*, 3 (1995), 226–244. 3.2

[11] Plummer, D. An ethernet address resolution protocol. RFC 826, Internet Engineering Task Force, 1982. 3.1

[12] Risso, F., and Degioanni, L. An architecture for high performance network analysis. In *Proceedings of the 6th IEEE Symposium on Computers and Communications (ISCC 2001)* (July 2001). 3.1

[13] Salvador, P., Valadas, R., and Pacheco, A. Multiscale fitting procedure using Markov Modulated Poisson Processes. *Telecommunication Systems 23*, 1–2 (2003), 123–148. 3.2, 3.2, 3.2

[14] Scott, S. L., and Smyth, P. The Markov Modulated Poisson Process and Markov Poisson Cascade with applications to web traffic data. In *Bayesian Statistics 7* (2003), Oxford University Press, pp. 671–680. 1

[15] Shah-Heydari, S., and Le-Ngoc, T. MMPP models for multimedia traffic. *Telecommunication Systems 15*, 3-4 (2000), 273–293. 3.2

[16] Yoshihara, T., Kasahara, S., and Takahashi, Y. Practical time-scale fitting of self-similar traffic with Markov-Modulated Poisson Process. *Telecommunication Systems 17*, 1-2 (2001), 185–211. 3.2