11

HTTP PERFORMANCE EVALUATION WITH TTCN

Roland Gecse, Péter Krémer, János Zoltán Szabó
Ericsson Research
Conformance Center, Ericsson Ltd.
H-1037 Budapest, Laborc u. 1., Hungary
E-mail: { Roland.Gecse, Peter.Kremer, Szabo.Janos } @eth.ericsson.se

Abstract This paper presents an approach, an implementation and an example for using TTCN in performance tests. The idea of using TTCN in performance test originates from PerfTTCN. The concept of PerfTTCN is excellent. The realization, however, does not seem to get acceptance as – to our knowledge – no vendor supports its non-standard tables. This is the point where the implementation described herein differs. The extra performance related tables are replaced by services abstracted in ASPs. The functionality behind them is then imported from or implemented as part of the MoT. This approach results in greater flexibility and wider range of applicability. An example is also shown using HTTP/1.0 and HTTP/1.1. Performance measurement results of these versions of the protocol are also presented. Finally we draw a conclusion.

Keywords: Performance test, TTCN, HTTP, Internet

1. INTRODUCTION

The performance aspects in today's telecommunication networks became as important as the functional correctness. In order to get repeatable and comparable results we need formalized and standardized methods not only for conformance tests but for performance measurements as well. A proposal to this is the PerfTTCN [1], an extension of TTCN (Tree and Tabular Combined Notation) [4], [5].

The basic idea behind conformance testing is to verify a protocol implementation – Implementation Under Test (IUT) – whether it fulfills the requirements described in its specification or not. TTCN is a standardized test description language used for black box testing of different protocols that are based on the OSI reference model, and it is mostly used for conformance testing of tele-

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35516-0_20

H. Ural et al. (eds.), *Testing of Communicating Systems*

com protocols and applications. The usage of TTCN seems to be an evident candidate since performance testing presumes the functional correctness (i.e. conformance) of the implementation and the conformance testing methodology is also applicable for IP-based protocols as it was shown in [17], [12].

PerfTTCN adds some extra functions to the base TTCN system. It defines a performance test configuration, which consists of one or more IUTs, Foreground Test Components (FTCs) communicating with IUTs and Background Test Components (BTCs). BTCs communicate with each other according to given traffic models in order to emulate the real load on the network, and they can monitor continuously the state of the network, too. PerfTTCN supports measurement and analysis as well: FTCs can measure for example the response times of IUT.

We chose the Hypertext Transfer Protocol (HTTP) for the evaluation of PerfTTCN, mainly because of its simplicity and widespread use. We know that there are plenty of well-established research studies related to the performance aspects of different HTTP versions (1.0 and 1.1) and we did not want to produce a "yapa" (yet another paper about ...). Our aim was to demonstrate the capabilites and to prove the usefulness of PerfTTCN. We think that the most effective way is to show that this approach gives comparable results.

This paper is organized as follows. The next section presents former work on HTTP performance evaluation outlining some problems and solutions. Section 3 describes PerfTTCN, our modifications on the theory and the implementation. After, we illustrate the usage of PerfTTCN with an example of HTTP (both 1.0 and 1.1) and present some test cases with explanation. The results of the tests can be found in section 4, where we compare the output of different configurations (HTTP/1.0 serial, HTTP/1.0 parallel, HTTP/1.1 persistent, HTTP/1.1 pipelined). Finally, we draw a conclusion and describe the future directions of our work.

2. RELATED WORK

The main purpose of this paper is to show how our PerfTTCN tool can be used for performance analysis. Our intention is to show consistent measurements taken with our tool rather than arguing on existing research results or introducing some new phenomenon. We chose HTTP as a base for our demonstration because there are a number of well-founded research results available.

In this section we discuss related work in the field of HTTP performance analysis. The first paper on HTTP performance problems [10] was published in 1994. It pinpointed certain design errors in HTTP protocol features to interfere with TCP. Single retrieval per request combined with small document size yield short lifespan connections, which cause performance and server

scalability problems. Diminished protocol performance can be expressed in terms of packet overhead caused by continuous TCP connection establishment and termination that requires 7 packets. Moreover, the short living connections cannot utilize the available network bandwith efficiently because they are still in slow-start phase when finishing transmission. On the server side, this results in numerous connections being in TIME_WAIT state, which makes servers run out of resources soon. Unfortunately, the author did not suggest any workaround to the presented problems.

Another significant paper aiming on decreasing HTTP latency is [9]. Basically it presented the same problems as [10] but also suggests methods for avoiding the problem, such as long-lived connections and the introduction of two new methods. The GETALL method was intended to retrieve a document with all embedded objects using only one request. The second one is the GETLIST method that was the basic idea behind the pipelining feature of HTTP/1.1. The authors also compared their experimental implementation with existing HTTP applications and measured gains both in number of transmitted packets and latency.

[8] describes further investigations on proposed persistent HTTP extensions with wide focusing simulation studies. It also goes into detail in interaction between HTTP and TCP with respect to experimental protocol extensions. Transaction TCP tries to get rid of connection handling overhead and problems related to that too many server connections are in TIME_WAIT state. This paper has been a serious contribution that led to changes in HTTP dynamics included in HTTP/1.1.

Later, [14] showed that persistent HTTP itself does not solve the latency problem. This is especially true on low bandwidth links, where it can hardly accomplish gain even in optimistic case.

[7] pointed out additional performance bottlenecks in TCP-HTTP interactions, which might, in the worst case, lead up to 20 times slower download time using persistent connections. Delayed acknowledgements as instructed by [15] combined with Nagle may cause serious delays that affects persistent HTTP traffic when small amount of data is transferred. The authors raised the problem of restarting idle TCP connections in combination with HTTP.

There have been continous investigations whether HTTP/1.1 can outperform HTTP/1.0 or not. [6] compared parallel 1.0, (persistent) 1.1 and 1.1 pipelined implementations. The authors took measurements on a sample web site using two different httpds. They also observed the impact of changing web content (e.g. using PNG graphics and CSS1 style sheets instead of GIF) as well as considered speed gains using HTTP/1.1's caching mechanism. Content compression techniques and different QoS links were also investigated. Their measurements proved that there are serious gains in overhead in terms of packets transmitted. However, in latency – which is the most interesting factor from

the user's perspective – there are no dramatic improvements even if pipelining is implemented.

Some papers address HTTP-TCP interaction from TCPs point of view. [16] deals with the HTTP latency problem. It proposed a new method for recovering from idle TCP connections. TCP congestion control did not regulate clearly what should happen with cwnd after recovering from a long inactive period. Some implementations use Jacobson's conservative approach of resetting the restart window (RW) to 1 segment size. Others simply use the previous value of cwnd and restart the connection with a burst. The latter technique would be very advantageous for HTTP latency. Nevertheless, this behaviour jeopardizes network stability. Authors of this paper gave a new proposal which is a good compromise between the two extremities mentioned before.

The latest version of TCP congestion control [18] gives new guidelines for TCP implementors. Among other serious issues, such as slow start and congestion avoidance now it also deals with restarting idle connections. However, the modifications suggested by the previous paper have not been included. [18] requires to set RW = IW ($\leq 2 * SMSS$ bytes or 2 segments). And that means, that recovering from idle persistent connections will remain a bottleneck in HTTP. This also justifies the findings of [6] that pipelining is needed in HTTP/1.1 implementations in order to outperform HTTP/1.0 with many simultaneous connections.

3. PERFTTCN

As PerfTTCN is only a proposal there will not be any commercial software supporting it in the near future. Therefore, our aim was to design and realize a performance testing environment, which has the similar functionality as the original PerfTTCN. Since there are many commercial implementations, which can execute test suites written in TTCN, it was obvious that we can add these extra functions as extensions to such a software.

The first implementation of PerfTTCN was presented together with the proposal in [1]. The researchers of GMD FOKUS used the GCI compiler of ITEX, which generated an executable C program code from the TTCN test suite. Then the generated code was completed with functions, which realized PerfTTCN's extensions.

We did not want to use the non-standard extension tables of PerfTTCN because it is very difficult – if possible at all – to make commercial tools interpret them. Thus, our performance test suites consist of only the standard TTCN tables though they contain all information about the background traffic and measurements as well.

These test suites have two extra PCO types (for the background traffic generation and measurements) and several ASP definitions. Our developed

modules – which handle these ASPs – can be considered as a part of the test execution environment. An advantage of this architecture is that existing conformance test suites can be easily converted into performance test suites by adding the needed definitions and ASP events.

3.1 Extensions to the Original Proposal

In the implementation of GMD FOKUS the extensions cannot be handled independently from the test suite. Thus the modification of the generated source code is always needed and it cannot be done automatically. One needs to know the structure of ASPs and PDUs in order to perform measurements. Furthermore, the configuration and parameters of the background traffic is stored separately from the test suite. In our solution all parameters are defined in the test suite and the PerfTTCN's extensions can be applied to any protocol without any changes.

Our extension module can evaluate the performed measurements run-time, during the execution of our tests. This means the measured values are accessible from the test suite and the performance test verdicts can be determined immediately at the end of the test case.

In our implementation (Figure 1), the parameters of the background traffic are set in the dynamic part during the execution of the test. Since we do not need to hard-wire these values, adaptive test cases can be developed, which can change the characteristics of the generated load based on the measured values.

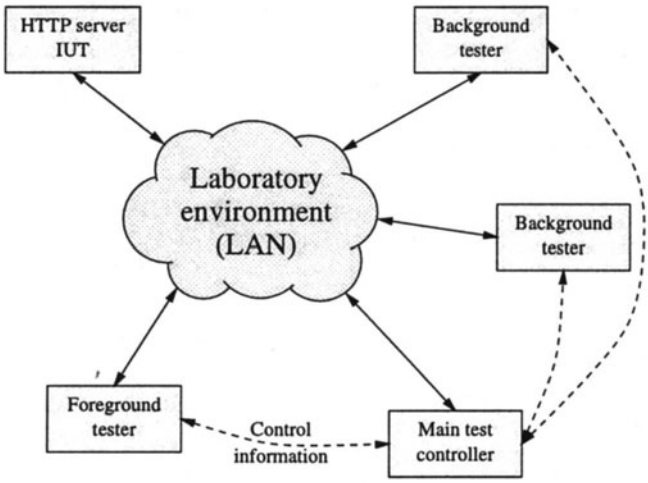


Figure 1. Test configuration of PerfTTCN

The original implementation limited the background traffic models to Markov Modulated Poisson Processes. But there are many protocols – especially in the application layer – whose traffic cannot be modelled with MMPPs. So we used traffic generators which do not have this limitation. With the current version of our extension module we can describe and generate arbitrary IP-based traffic and interfaces to any traffic generator systems can be easily developed.

3.2 The Implementation

For the execution of our tests we use Ericsson's System Certification System – SCS. It has an interpreter-based TTCN test executor, which provides a generic framework for testing various protocols. SCS has to be adapted to the tested protocol using a so called test port. A test port is a library used by the test executor which maps the TTCN ASPs onto SPs of the underlying service provider in order to exchange data with IUT.

Each test port consists of a C++ class, which implements the ASPs of a given PCO type. The ASPs to be sent or received are mapped to functions, which must be written by the user. The ASN.1 and TTCN data types are converted into C structures.

Since we did not want to change the format of the TTCN test suite by adding new tables, we have defined a new interface for PerfTTCN through ASPs. We have developed two test ports that implement the extensions of PerfTTCN instead of communicating with IUT.

The main idea of this conception is that these test ports provide a service for the test executor which can be accessed through ASPs. In order to use their services first we have to define some objects (like measurements, traffic models, etc.) inside the test port. The definitions are done by sending special declaration ASPs usually in the preamble of the test case. Each object must have a unique name that identifies it in further operations.

3.3 Measurements

One of the developed test ports deals with measurements and analysis (Figure 2(a)). These functions are used by FTCs, which communicate directly with IUT according to the TTCN dynamic behaviour specification. This test port can carry out performance measurements during our test (e.g. the elapsed time between the sending of a request and the IUT's response can be measured) and it can perform statistical calculations automatically. We need these functions to accomplish performance tests because the base TTCN language lacks in such measurements. There are timers and timeout events, which can only detect error situations. The derived parameters, such as throughput rates, can be simply calculated from the measured times.

This test port supports three types of objects: measurements, characteristics and performance constraints.

A measurement object means a simple measurement of elapsed time between two test events based on the system time of the test executor equipment. Such a measurement behaves like a stop-watch, it is to be started and stopped by sending special ASPs to the test port at the corresponding test events. A measurement has a unit – like TTCN timers – which should be set in the declaration. The test port supports parallel measurements as well, so the different objects are treated independently. The last value of a measurement can be queried at any time.

The characteristic stands for a statistical property – like mean or peak value – of one given measurement. The declaration of a characteristic should consist of the name of the corresponding measurement and the type of the property. We can set a minimal sample size or a minimal sampling time interval as well. The actual value of a characteristic is automatically calculated by the test port according to the repeated measurements. This value – providing that the characeristic has a valid one – can be accessed at any time.

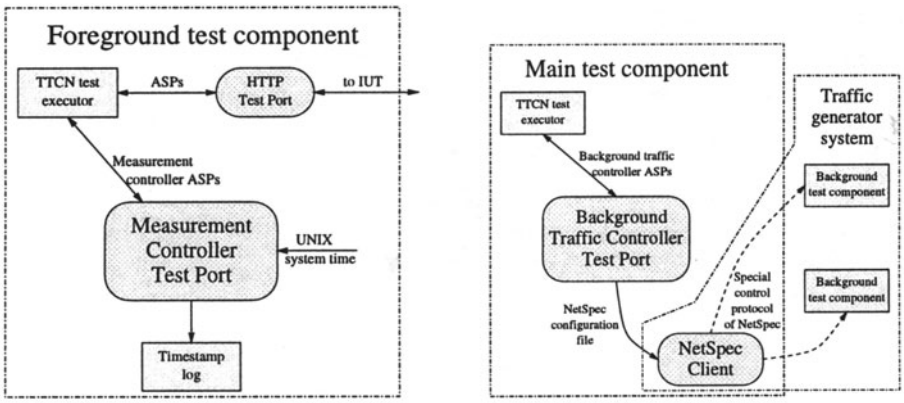
A performance constraint is a logical expression which contains the condition that the IUT has to fulfil to pass the performance test case. The expression should be given in a GeneralString argument of the ASP. It may refer to characteristics or other performance constraints and may use the usual arithmetical, logical and comparison operators. A performance constraint – like a verdict in TTCN – can have one of the three possible values: pass, fail or inconclusive. The inconclusive value means that one or more of the referenced characteristics still do not have a value because of the insufficient number of samples. The value can be queried at any time, but it is useful at the end of the test case's postamble to set the final verdict.

The test port makes a log file which contains the most important events including the measured values to enable further data processing and evaluation.

3.4 Background Traffic Generation

The TTCN-based implementation of BTCs is quite difficult if possible at all. Our approach was to build ready-made traffic generators into the TTCN test executor system. Since the protocol layer used by the traffic generators should match the tested layer in most cases, we need to use different traffic generators for testing different layers. Thus, our purpose was to develop a uniform control interface to different traffic generators. There are traffic generators available, which can be controlled remotely from a central point, therefore we have integrated such a system first.

The traffic generator that we built into our PerfTTCN environment is called NetSpec [13]. It was developed at the University of Kansas and its source code



(a) (b)

Figure 2. Measurement controller test port (a), background traffic controller test port (b)

is freely available. It is portable to several UNIX platforms (Solaris, FreeBSD, Linux, etc.) and its components that are running on different platforms can also co-operate. The base system can generate traffic flows over TCP or UDP and there are extension modules for ATM, CORBA, etc. NetSpec has a distributed client-server architecture, which uses a special – TCP-based – control protocol. The only thing that is needed for traffic generation is to run a main server daemon on each computer that we want to participate in our configuration. The traffic models and configuration is described in a text file using a simple block-oriented language. The downloading of traffic parameters and the synchronization are done by the server daemons automatically. NetSpec has a limitation that it can generate only one-way data flows, so symmetric TCP flows cannot be modelled with it.

Our second test port is responsible for the control of background traffic (Figure 2(b)). The main purpose of this module is to make the process of performance testing fully automated. Since the traffic flows are described and controlled from a TTCN test suite, there is no need for user interaction during the test.

The parameters of background traffic can be defined in the test port with three types of objects: traffic models, components and loads. These objects are independent of the used traffic generator. The NetSpec-specific parameters are described in ASN.1 CHOICE constructs, which can be extended to support the features of other load generators.

In our approach a traffic model covers all stochastic behaviours of a single one-way flow. The model describes only the data source and is independent of the type of the underlying service. The destination node is assumed to be an ideal sink. With this abstraction we can generate both connectionless datagram (UDP) and connection-oriented stream (TCP) traffic based on the same model.

NetSpec supports user defined and application-layer traffic models. The application-layer models emulate the generated traffic of the most commonly used Internet applications like WWW, FTP or telnet. These models have few parameters (e.g. average request interarrivals or document sizes) because they assume the distribution type of the traffic parameters fixed. A user defined traffic model enables to specify the base stochastic characteristics of the traffic. The request interarrival time, the number of blocks per request and the block size can be defined using mathematical or empiric distributions.

NetSpec has another type of model called full. This is the simplest one of all. In this case the source node tries to send as much data as possible to utilize all the available bandwidth. This is useful to cause artifical overload situations on a part of the network.

In the test port first we have to define our models. The parameters of a model are described in nested ASN.1 SEQUENCE and CHOICE constructs. For example, several types of distributions with their parameters are represented as the alternatives of a CHOICE construct.

A traffic component is a one-way flow based on a given model with definite source and destination nodes and transport protocol. Thus, the parameters of the component-definition ASP contain the used protocol (TCP or UDP), the two endpoints and port numbers of the corresponding flow. NetSpec enables to set some protocol specific features, like the window size of TCP. These can be specified in the TTCN test suite as well.

During a performance test the background load is generated by numerous flows. These loads have to be defined in the test port, too. Such a load consists of a set of the components. It is possible that a load includes several instances of the same component. In this case the test port can assign unique port numbers to the multiple instances of the same component.

After the definitions but still in the preamble, the TTCN main test component should start the generated load by sending an ASP. At this point our test port generates a NetSpec configuration file according to the model and component definitions and passes it to the NetSpec client program.

Since NetSpec cannot stop its traffic generators at any time, we have to specify a duration for each load. Of course, this duration must be longer than the estimated execution time of the test case.

4. TEST RESULTS

We performed our initial measurements in laboratory conditions on a separate 10 Mbps Ethernet segment with four machines (a Sun Sparc and three PCs with Linux) as shown in Figure 3(a). All stations were connected to a hub, which did not have any external connection. Heintel (Sun workstation) played the roles of FTCs and the main test controller.

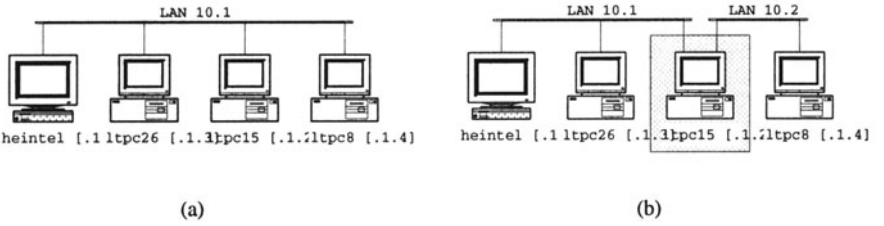


Figure 3. Configuration of the test network with one segment (a) and with two segments (b)

Traffic was generated only between ltpc15 and ltpc26. These two PCs were the BTCs. The Apache Web Server (version 1.3.4) running on ltpc8 acted as IUT.

The background load was generated using NetSpec's "full" traffic model with a blocksize of 8192 bytes (this is the default value in NetSpec). We always generated the same number of TCP flows in both directions. We ensured that all measurements have been executed after the generated background load became stable. Thus the given number of TCP connections occupied fairly all available Ethernet bandwidth.

We increased the number of simultaneous TCP connections up to 100. Additional increase would not be worth since in our 10 Mbps Ethernet the effective bandwidth is 5 kbyte/s for each TCP connection.

$$\frac{10Mbps * 0.8(eff\_rate)}{2 * 100} = 40kbit/s = 5kbyte/s$$

It means that a TCP flow can send only 3 packets in one second because the MTU is 1536. This amount is too small so the congestion window could not grow above 9. Situations like this does not allow a TCP to perform well. Our test network collapsed when we tried to increase the number of TCP connections to 200.

We set up a test web "site" consisting of 13 files: an HTML page and a dozen embedded GIF images. The main HTML page has the size of 29 kbytes, about the half of the images are shorter than 3 kbytes, while the others are between 3

and 20 kbytes. The total size of about 80 kbytes can be considered as a typical web document.

We developed test cases (TCs) for downloading these files from IUT using different features of the two versions of HTTP. The only FTC of the TC named "HTTP/1.0 serial" downloads the documents consecutively using the version 1.0. It opens different TCP connections for each page. This method is the worst of all and is used for reference. The TC "HTTP/1.0 parallel" also uses version 1.0 but it uses parallel TCP connections to decrease download time. The main difference between these two TCs is that the latter uses four FTCs and four parallel TCP connections while the former is satisfied with only one of each. Furthermore, in the parallel TC the main test component assigns the documents to the FTCs dynamically in order to achieve the best latency among stochastic conditions.

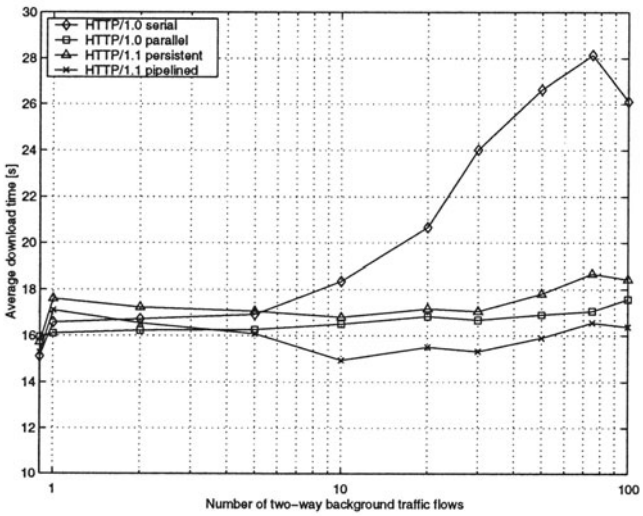


Figure 4. Measurement of HTTP download times against network load using the conformance test port of HTTP (on the single segment configuration)

The TCs "HTTP/1.1 persistent" and "HTTP/1.1 pipelined" use the last version of HTTP. Both of them use persistent connections, thus they download all the documents using the same TCP connection. The latter TC uses the pipelining feature to further enhance download speed.

The initial results are shown in Figure 4. The presented results show the mean values of 10 successive measurements as calculated by our measurement controller test port.

At first sight we found that our results satisfied our expectations. We achieved similar results as the majority of cited papers. The best performing "HTTP/1.1

pipeline" always outperformed HTTP/1.0 using four parallel connections. The latency of "HTTP/1.0 serial" grew with the increasing number of background traffic components.

It is interesting, that this is not the case for the rest of our measurements. Another, more significant problem is the download time. It takes more than 15 seconds to download the 80 kB test documents, which is pretty slow on a dedicated Ethernet line. The bottleneck in our configuration turned out to be the HTTP test port in our TTCN executor. This test port was originally developed for conformance test for [11]. It uses very detailed ASN.1 descriptions of HTTP data structures including grammar extensions for building all kinds of invalid PDUs. Additional slowing factor is that we also validate the received data to check if there is mismatch between expected/received data (HTML and GIF content). Unfortunately, the execution platform was not fast enough to cope with these problems.

4.1 Performance Test Port

Finally, we decided to replace our HTTP test port with another one, specially designed for performance measurement called performance test port. We disabled the payload examination and replaced the detailed ASN.1 PDU description part with a simple string based structure. All other parameters, the executed TCs and the configuration were the same. The results, however, showed significant improvement as it can be seen in Figure 5. The download time on the unloaded LAN decreased under 700 ms for all cases. HTTP/1.1 with pipelining was the winner this time, too. Then HTTP/1.0 using multiple connections follows before the persistent HTTP/1.1. Again the original HTTP/1.0 performed worst. It is interesting to note that after a short ramp up phase all curves become flat. That is, over a certain number of competing parallel connections the download times do not change significantly.

A serious drawback of the previous configurations is that both of them measure CSMA/CD performance. Since the traffic generators run on other machines than IUT and FTCs the generated load wipes out the chance of the access to the shared medium only. But our initial goal was to point out differences between implementation details in realistic situations. Thus, we set up a second configuration having two subnets, such as Figure 3(b), where we also introduced an intermediate router powered by Linux 2.2.12.

In addition to performing regression tests using our new configuration, this time we also performed the measurements using real WWW clients in order to compare our results. Netscape Navigator 4.5 runs 4 parallel "Keep-Alive" HTTP/1.0 connections (which is similar to a "HTTP/1.1 persistent" one) while wget uses plain HTTP/1.0. We found our measurement results (Figure 6) using the new test port comparable with those of the mentioned applications. On the

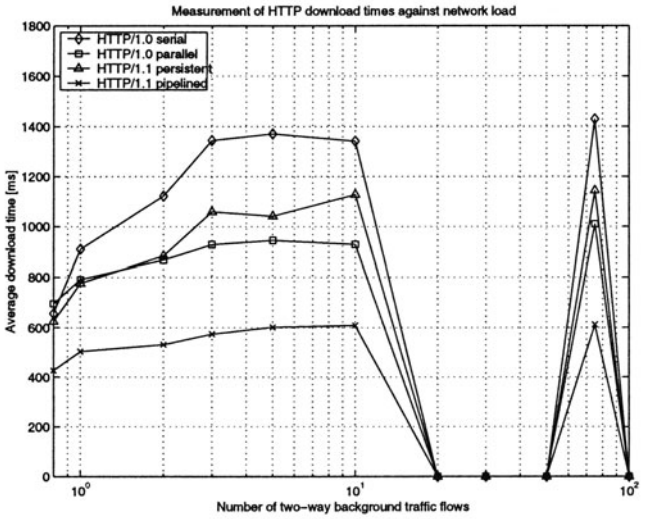


Figure 5. Measurement results using the performance test port of HTTP

one hand this proved that our test port and test cases are correct. On the other hand this gave the evidence that the PerfTTCN concept does really work and it is applicable for performance testing. Further investigation was only possible after these criteria has been fulfilled.

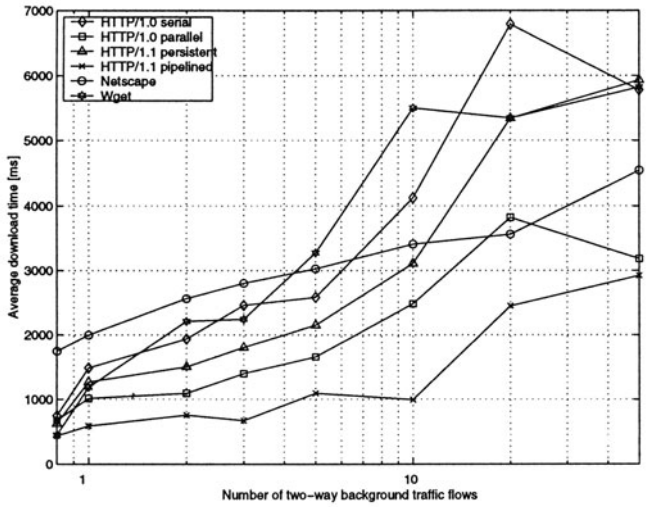


Figure 6. Measurement results on the second configuration (two segment)

Our last mentioned test case – simulating an HTTP/1.1 pipelining client – performed the best again. It is no wonder that it outperformed Netscape Navigator as we do not process the received data just discard it. However, wget was not agressive at all when it comes to download speed. Certainly, this is due to politeness criteria that has to be fulfilled by each robot application. The curves themselves look now as we expected. The mean download time is an exponential function of the number of parallel flows, which is proportional to the network utilization.

Our final measurement in Figure 7 shows what happened when we increased the amount of downloaded data. We designed a new test document consisting of a 2 kbyte HTML, a small 12 kbyte background and 4 pictures of length of about 100 kbytes each. The total size was 545 kbytes. This time we saw that HTTP/1.0 using multiple connections always outperformed the HTTP/1.1 pipelining implementation. We consider this as a consequence of the relatively small number of downloaded objects and their longer size. In the case of our previous test document, in which we had more objects of smaller size, we believe that the extreme background traffic filled up the router queues and pipelining could not work efficiently. This also justified our thought that "HTTP/1.0 parallel" can outperform "HTTP/1.1 pipeline" under certain circumstances. This is done, however, on the expense of more signalling overhead.

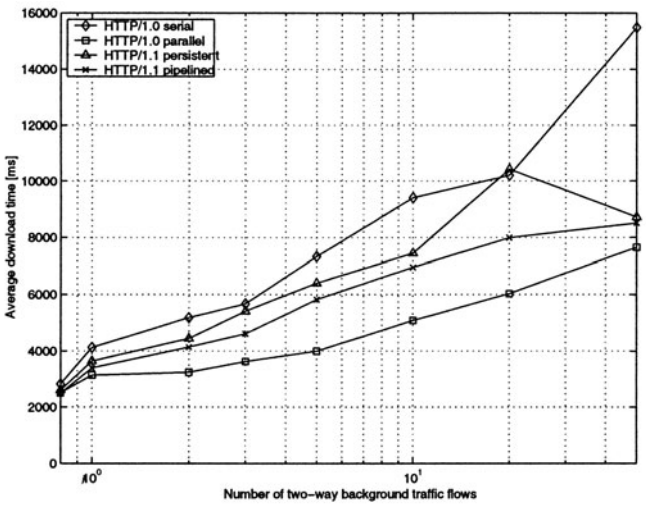


Figure 7. Measurement results downloading the bigger test document (using the second configuration)

5. CONCLUSION

We have performed HTTP performance tests in a new way, we have run our comparison tests on top of loaded network. We have measured latency by the number of simultaneous TCP connections. We have managed to get similar results than many researchers before us, with only two remarks. First, we have run our tests on congested network, which has not been done before in any of the cited papers. Second, mainly our focus was on demonstrating that our environment based on PerfTTCN can cope with any other tool in real life measurements. Unlike other traffic engineers, as a result of our investigation we have found that there are situations when HTTP/1.0 using simultaneous connections achieves better performance than HTTP/1.1.

We have run our test cases on two different network configurations downloading two web pages having different characteristics using one type of three available traffic models. Future work can be, for instance, to repeat the measurements with other traffic models. The tests could also be repeated simultaneously, i.e. running all four types of data transfer in parallel and investigate their competition. Besides, the real-time evaluation of distributed measurements (when the start and end events are on different test components running on different machines) is a quite interesting (and difficult) problem, too.

References

[1] I. Schieferdecker, B. Stepien, A. Rennoch: PerfTTCN, a TTCN language extension for performance testing, Testing of Communicating Systems, Cheju Island, Korea, September 1997.

[2] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, T. Berners-Lee: Hypertext Transfer Protocol – HTTP/1.1, RFC 2616, IETF, Network Working Group, June 1999.

[3] T. Berners-Lee, R. Fielding, H. Frystyk: Hypertext Transfer Protocol – HTTP/1.0, RFC 1945, IETF Network Working Group, May 1996.

[4] OSI - Open System Interconnection, Conformance testing methodology and framework, ISO/IEC 9646, 1997.

[5] B. Baumgarten, A. Giessler: OSI conformance testing methodology and TTCN, North Holland, 1994.

[6] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, C. Lilley: Network performance effects of HTTP/1.1, CSS1 and PNG, http://www.w3.org/Protocols/HTTP/Performance/Pipeline.html, June 1997.

[7] J. Heidemann: Performance interactions between P-HTTP and TCP implementations, ACM Computer Communications Review, pp. 65-73, April 1997.

[8] J. C. Mogul: The case for persistent-connection HTTP, Western Research Laboratory Research Report 95/4, http://www.research.digital.com/wrl/publications/abstracts/95.4.html, May 1995.

[9] V. N. Padmanabhan, J. C. Mogul: Improving HTTP latency, Computer Networks and ISDN Systems, Vol. 28., pp. 25-35, December 1995.

[10] S. E. Spero: Analysis of HTTP performance problems, http://www.w3.org/Protocols/HTTP/1.0/HTTPPerformance.html, July 1994.

[11] R. Gecse: Conformance testing methodology of Internet protocols, Testing of Communicating Systems, Tomsk, Russia, September 1998.

[12] R. Gecse, P. Krémer: Automated test of TCP congestion control algorithms, Testing of Communicating Systems, Budapest, Hungary, September 1999.

[13] R. Jonkman: NetSpec User Manual, http://www.ittc.ukans.edu/netspec, April 1999.

[14] J. Touch, J. Heidemann, K. Obraczka: Analysis of HTTPPerformance, USC/Information Sciences Institute, http://www.isi.edu/lsam/publications/http-perf/, Aug. 16, 1996.

[15] R. Braden: Requirements for Internet Hosts – Communication Layers, STD 3, RFC 1122, October 1989.

[16] V. Visweswaraiah, J. Heidemann: Improving Restart of Idle TCP Connections, USC TR 97-661, http://www.isi.edu/ johnh/PAPERS/Visweswaraiah97b.html, November 10, 1997.

[17] T. Ogishi, A. Idoue, T. Kato and K. Suzuki: Intelligent protocol analyzer for WWW server accesses with exception handling function, Testing of Communicating Systems, Tomsk, Russia, September 1998.

[18] M. Allman, V. Paxson, W. Stevens: TCP Congestion Control, RFC 2581, April 1999.