A Comparison of Portable Dynamic Web Content Technologies for the Apache Server[*][†]

George Gousios
Department of Information and Communication Systems
University of the Aegean
cs98011@icsd.aegean.gr

Diomidis Spinellis
Department Management Science and Technology
Athens University of Economics and Business
dds@aueb.gr

Abstract

Apache is considered to be the most extensible, secure and widely used server on the Internet. On our talk we focus on its first characteristic, extensibility, analyzing many techniques used to provide dynamic content. Available solutions are based either on extensions to the web server itself or on the execution of user-space programs. These solutions include, among others, CGI scripts, PHP, mod_perl, mod_python and Java Servlets. For each technology we present its basic design goals and the development facilities it offers. We compare the efficiency of these technologies by means of custom-made benchmarks we run to measure each solution's throughput. Finally, we present each technique's drawbacks, with references to lessons learned during the complete deploy-and-test process.

1 Introduction

Dynamic web content typically entails user input on a web page processed by the server to affect the generation of a corresponding new page. In most applications the server-side processing is the key to the whole process. The web server is responsible for handling user input, start a program that processes it (or just pass it to an already running program), get the results and send them back to the user. The processing program often communicates with a database to find stored information and keeps session data to remember the user's previous state.

To achieve the above, there are a number of different approaches:

[*] In *SANE 2002: 3rd International System Administration and Networking Conference Proceedings* , pp. 103–119. Best refereed paper award. Maastricht, The Netherlands, May 2002. NLUUG.

[†] This is a machine-readable rendering of a working paper draft that led to a publication. The publication should always be cited in preference to this draft using the reference in the previous footnote. This material is presented to ensure timely dissemination of scholarly and technical work. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder.

- Per-request execution: Every time dynamic content is needed, start a new program to process the request.
- A pool of the requested processes or threads is already running. Each new request is handled by a separate process or thread.
- Templating: We have already designed the page. Its request-specific contents are added and it is sent back to the user.
- Web server extensions: The web server itself is extended through proprietary APIs to handle dynamic content.

The third approach implies that the previous two produce themselves the HTML code for every request. Also, the third technique requires the presence of either the second (usually) or the first (seldom).

The technologies that work for today's web sites, according to the previous classification, are the following:

- Per-request execution: Common Gateway Interface (CGI).
- Fast CGI, Java Servlets.
- Templating: Microsoft Active Server Pages (ASP), Sun Java Server Pages (JSP), Hypertext Pre-Processor (PHP).
- Web server extensions: Server-side Includes, mod_perl (for Apache), NSAPI (Netscape FastTrack), ISAPI(MS IIS).

The goal of our work was to evaluate the relative performance of key open-source dynamic web serving technologies. In the following sections we will present common technologies used for serving dynamic content under the apache web server, outline the methodology we used for comparing their performance, present the results we obtained, and discuss some of the lessons we learned.

2 Available Technologies

2.1 The Common Gateway Interface

The Common Gateway Interface, usually referred to as CGI, is the ancestor of technologies that enable dynamic content at the web. It is a simple interface, supported by almost every web server and thus is the best known protocol among developers.

CGI programs were first written in C and the Bourne shell. The low level of C in combination with the difficulty of handling text data using it, quickly drew developers attention to Perl. Today, Perl is the dominant CGI language, but libraries exist and CGI programs can be written in almost every possible language.

CGI programs communicate with the web server using environment variables. The information that can be accessed includes among other the GET or POST request arguments, and the request's HTTP header. For every request, a new copy of the CGI program is executed. It examines the request's header and arguments and process them accordingly.

The major disadvantages of CGI are somewhat obvious:

- A new request forces the system spawn a new process. Imagine the overhead involved when starting a Perl or even a Java interpreter and then connect to a database. Now multiply it with some tens or hundreds of requests a heavy loaded site accepts a second. CGI sites often suffer from speed problems even if when using a precompiled CGI program written in C.

- The protocol was not designed with session tracking in mind. Session data are lost, since the program dies after its execution. Designers have to write special code to track sessions.
- CGI programs mesh presentation with program logic. The HTML page must be prepared before the programmers start writing CGI code.

These disadvantages initially led developers to use hacks like communicating with other already running processes using the web-server's API or using generic IPC mechanisms. A unified, but totaly different in principle, approach, FastCGI, emerged back in 1995. The important difference between FastCGI and CGI is that FastCGI scripts are compiled and started only once. Globally initialized objects, for example database connections, are thus persistently maintained. The FastCGI scripts are run by a mini application server, a process that is responsible to run the script's main loop and return the results to the web server. FastCGI scripts work outside the web server and the communication is done using Unix domain sockets. All requests are handled by this process in a FIFO order. There is also a possibility to spread the load of running FastCGI scripts into different machines by using TCP/IP connections instead of Unix sockets.

While FastCGI's process model seems to be much faster than CGI's, the FastCGI protocol is not supported by specialized development tools. FastCGI incorporates a session-tracking like feature in the protocol, that is available when supported by the server implementation It is called session affinity and all it does is to route requests coming from the same client to the same FastCGI application server. Through the use of special libraries, the programmer can maintain session tracking info.

Figure 1 contains an example of how a FastCGI script is written. This script does simple form processing and inserts the values acquired from the user into a PostgreSQL database. It is an analog to the script that was used for the benchmarks mentioned later in this article.

2.2 The Servlet Approach

Servlets were initially proposed by Sun as a CGI replacement; they are now an open standard, in the spirit of Java. Servlets are programs, written in Java, that extend the functionality of the application server in which they are run. The process model used is analogous to the FastCGI. For every request, a new thread is spawned to handle it. Requests for different servlets cause different threads to start without having to wait for their turn. Servlets run in an environment called servlet container, which is in fact a JVM with some classes preloaded to provide extra functionality to the running servlets and to allow them to communicate with the outer world, for example to receive request parameters. The execution thread only executes the function that is appropriate for the HTTP method that called the servlet (usually GET or POST), but has access to objects that are globally initialized. This way we can have objects initialized only once, but used for the whole servlet lifetime: persistent database connections, connection pools and RMI connections to other running processes.

Developing web applications with servlets has many advantages that are hard to come by in any other (open source) system. A servlet can include every Java class, so uniform access to databases via JDBC and several XML parsers and XSLT transformers can be used by the developer. Specialized tools like `ant`, the Java analog to `make`, and `jasper`, the JSP pre-compiler can be used by developers to design, develop and finally install a web site without having their attention drawn to trivial details. Another advantage of servlets is that they provide classes and methods dedicated to both user-space (cookies) and server-space session tracking. Also, the fact that they are written in Java makes servlets fully portable. Finally, the servlet

```
#!/usr/bin/perl
#load the necessary modules
use Pg;
use CGI qw/:standard/;
use CGI::Fast;
#connect to the database
$conn = Pg::connectdb("dbname=comments host=193.250.160.3\
user=george password=george");
die $conn>errorMessage unless PGRES_CONNECTION_OK eq $conn>status;
#Create a new FastCGI object
#This is the main program loop
#Every
while(new CGI::Fast){
  $name=param("name");
  $email=param("email");
  $comments=param("comments");
  #insert the record to the database
  $query="insert into comments values('".$name."','".$email.\
  "','".$comments."')";
  $result=$conn>exec($query);
  die $conn>errorMessage unless PGRES_COMMAND_OK eq $result>resultStatus;
  print "All done OK";
}
#close the connection
$conn->requestCancel
```

Figure 1: Form processing with FastCGI

protocol specifies the existence of the Java Server Pages template system, which is in turn supported by most commercial web development tools.

To be able to execute servlets, a web server must be equipped with a servlet runner. This can be done either with a module or a standalone servlet engine, to which the requests for servlets are redirected. The most well known (open source) servlet engine is Tomcat [5] by the Apache group; an offering that can be easily integrated with the Apache web server. An explicit goal of Tomcat's development is to support the latest servlet specifications as documented by Sun. Servlet containers are also implemented by all commercial J2EE compliant servers like iPlanet or WebSphere.

In figure 2 we illustrate the functionality of the previous FastCGI script, written in Java.

2.3 Templating: The Hypertext Preprocessor (PHP)

Writing a script that simply prints some HTML tags combined with request-specific results of a database query is often a needlessly error-prone development process with an end product that is usually difficult to maintain. This is partially due to the complex entanglement of presentation and processing details in many scripting languages including Perl and Python. The idea of embedding scripting code in the HTML page was the initial motivation that led to the development of the templating technique. Today, templating systems range from simple pages with database access to large XML-based e-commerce applications. Template systems find better acceptance in the grounds of web-publishing and content based sites. Template solutions include ColdFusion, PHP, JSP and ASP, although most of them are based on the process model of an application server. The most commonly used template system is PHP. It runs in all major web server platforms. PHP's design goal was

```
import java.io.*;
import java.lang.*;
import javax.servlet.*;
import javax.servlet.http.*;
import java.sql.*;

public class comments extends HttpServlet {
  Connection con=null;
  //This is executed only once during servlet loading
  public void init(ServletConfig config) throws ServletException{
    super.init(config);
    try {
      Class.forName("org.postgresql.Driver");
      con=DriverManager.getConnection("jdbc:postgresql:comments",\ "george","george");
    }
    catch (ClassNotFoundException e) {
      System.out.println("No such class:"+e.getMessage());
    }
    catch (SQLException s) {
      System.out.println("Connection error"+s.getMessage());
    }
  }

  //The function that handles POST requests
  public void doPost (HttpServletRequest req, HttpServletResponse res)
  throws ServletException,IOException{
    //Get input parameters
    String name=req.getParameter("name");
    String email=req.getParameter("email");
    String comments=req.getParameter("comments");
    PrintWriter out=res.getWriter();
    res.setContentType("text/html");
    //Insert the record into the database
    try {
      Statement stmt=con.createStatement();
      stmt.execute("INSERT INTO comments values('"+name+"','"+email+\"','"+comments+"')");
      out.println("All done ok");
    }
    catch (SQLException s) {
      out.println("Connection error"+s.getMessage());
    }
    finally {
      out.println("</BODY></HTML>");
    }
  }

  //GET and POST requests are handled in the same way
  public void doGet (HttpServletRequest req, HttpServletResponse res)
  throws ServletException,IOException{
    doPost (req,res);
  }

  //Only called when the servlet is unloaded
  public void destroy() {
    try {
      con.close();
    }
    catch (SQLException s) { }
  }
}
```

Figure 2: Form processing with servlets

```
<?php
$db = pg_pconnect ("host=localhost dbname=comments user=george password=george");
$update="insert into comments(name,email,comment) values ('$name','$email','$comments')";
echo $name,$email,$comments;
pg_exec($db,$update);
?>
```

Figure 3: Form processing with PHP

to provide a means of fast web site development. PHP uses a Perl-like syntax with C-style loops and conditionals. It features easy to use commands for nearly every possible operation in a web environment. It can handle file uploads, DBMS-specific persistent database connections, sessions and everything else up to on the fly PDF creation. The ease of use and the wide set of supported modules drove PHP in wide acceptance.

PHP can work either as an Apache extension or as a CGI script. Of course the former is the more efficient approach. Working as an Apache extension means that its parser is loaded during server boot and every new Apache process forked will contain it. The PHP scripts are interpreted every time a request is done but some objects that are declared persistent such as database connections can stay open. The difference from servlets and FastCGI is that there are no shared objects. Database connections for example belong to each Apache process that runs the script. This approach can load an external resource with many created objects, but, in combination with large amounts of memory, can increase performance. The PHP's core, the Zend scripting engine, is responsible to locate and interpret the script, load the PHP's modules that are necessary and instantiate communications with external resources.

The following example is the analog to the previous servlet and FastCGI scripts, which demonstrates the simplicity of the code needed for simple form processing.

2.4 Extension API's

All major web servers provide an API to allow the development of extension modules. These modules usually extend the functionality of a web server or provide means for executing programs in languages not directly supported by the web server. The most used Apache scripting extension module (see [6]), apart from PHP, is mod_perl. The mod_perl module brings the full power of the Perl programming language to the Apache HTTP server. This is achieved by linking the Perl runtime library into the server and providing an object-oriented Perl interface to the server's C language API. These pieces are seamlessly glued together by the mod_perl server plug in, making it is possible to write Apache modules entirely in Perl.

The module brings a persistent Perl interpreter into Apache. "Persistent" in this context means that the scripts to be executed are parsed only once and the overhead of repeatedly starting an external interpreter and parsing the Perl script are avoided. Notice however that the scope of persistence covers only an Apache instance (child) and not the whole process group. So, persistent database connections will die and parsed scripts will be lost, if the parent decides that the child must die (for example, when the MaxSpareServers are above the number specified in the server's configuration file). The distribution pack of mod_perl also includes some Perl libraries to enable simple CGI scripts to run as native mod_perl applications. These libraries are intended to make the transition from CGI to mod_perl effortless. A widely used library (actually an Apache extension developed using the mod_perl facilities) is the Apache::Registry module.

```
#!/usr/bin/perl w
use strict;
use Pg;
use CGI;
$conn = Pg::connectdb("dbname=comments host=localhost user=george password=george");
die $conn>errorMessage unless !$conn>status;
my $name=CGI::param("name");
my $email=CGI::param("email");
my $comments=CGI::param("comments");
my $query="insert into comments values('".$name."','".$email."','".$comments."')";
my $result=Pg::con->exec($query);
```

Figure 4: Form processing with mod_perl

The example in figure 4 shows a script that can be used as a mod_perl extension as well as a CGI without changes.

3 Performance Comparison

Servlets in theory seem to be the best solution for web application development, but in practice it is common secret that PHP allows very fast web development and fast execution. On the other hand mod_perl can use the full set of the extensive module collection that Apache features, while being very fast and relatively easy to implement. Are all these capable of beating the simplicity and efficiency of the FastCGI protocol? Being in such a dilemma, we decided to measure each protocol's throughput by means of custom made benchmarks as well as using the `Jmeter` ([5]) load generator.

3.1 The Server

Having the intention to measure each protocol's throughput and not the database's performance, we created a very simple database schema. The application to be served consisted of a single table. A technology specific script/program was responsible to execute a simple *Select ** query and print the results formatted in an HTML table. We tried to write the scripts in a way that made them as fast to execute as possible, thus omitting condition or value integrity checks and providing using database connections were this was possible.

Our server hardware consisted of a PIII 733 Mhz machine, using 384 MB of memory. The server's operating system was Suse Linux 7.3 with an updated kernel (2.4.17) and an ext3 (ext2 with journaling extensions) filesystem. We also installed Apache 1.3.20, Tomcat 3.3, PostgreSQL 7.1.3 as database, Sun JDK 1.4.0 and Perl 5.6.1.

3.2 Server Configuration

An important, and often bitterly debated aspect of every benchmark is the server's configuration and tuning. The following paragraphs outline how was the server's operating environment configured. The Linux kernel provides the `/proc/sys/` interface to its internal parameters that can be tuned. Using the `powertweak` tool, we tuned some parameters, most notably those affecting the TIME_WAIT sockets (reducing their time of life) and the hard disk subsystem (enabling DMA).

The Apache web server was setup according to the `highperformance.conf-dist` and the `perf-tuning.html` files that accompany Apache's source distribution. Knowing that our server was an ordinary PC and not a dedicated web server with fast SCSI drives and advanced I/O, we decided to have 30 children starting, minimum 10 children running and a soft 50 children maximum, which could be overlooked to satisfy traffic spikes. Each child could serve at least 1000 requests before passing away. The keep-alive timeout was set relatively low (5 seconds) because there was no need to satisfy keep-alive requests from the client. We also disabled both symbolic link checking and logging. During benchmark runs, Apache was configured using the excellent tool that is present in Suse Linux 7.3 distribution, `yast`. This tool is very efficient on configuring which modules should be loaded. When a benchmark for a certain technology was to be run, every not-related module was unloaded. The only modules present while a benchmark was run were the ones related to the technology benchmarked and some core modules like mod_alias and mod_mime.

In the case of servlet testing, we also had to configure Tomcat. In the official Tomcat distribution there is no document related to tuning, but hints do exist in parts of its documentation. After careful studying of the documentation, we set up the communication between Tomcat and Apache with the (recommended) apj13 protocol and disabled logging, deleted every web application not needed and disabled the jni worker.

Moving on, the mod_perl documentation includes a very good tuning HOWTO, which we followed to the extend possible. We wrote an initialization script that preloads all the needed libraries, and used the *strict* Perl library to ensure the code quality, as proposed by the tuning HOWTO.

Finally, in the case of FastCGI, we preloaded the necessary scripts by declaring them as application servers in the `httpd.conf` file.

3.3 The Client

The client hardware was a powerful Presario laptop featuring a Duron running at 950 Mhz with 256MB RAM. The two machines communicated via 100baseTX network, using a direct reversed-cable dedicated connection. The client also run Linux (2.4.18), Perl 5.6.1 and Java 1.4.

3.4 The Benchmarks

According to [9], the main objectives of performance testing are to measure the maximum number of concurrent clients that can get "acceptable" performance and also the maximum number of clients prior to system failure. The same source also suggests two variations of performance testing that we generally appreciated and tried to embody in our experiments: Load testing, which models intense network traffic but also simulates the user's "think time" and stress tests which are the same with load tests but without the delays caused by the user's think time. We tried to measure the following quantities for the technologies illustrated previously.

Throughput (load test) The maximum number of clients that can be served, while performance is acceptable. Also, the traffic that is generated by responses.

Latency (load test) The time a request takes to be served while load is rising.

Robustness (stress test) The number of clients that can be served before we encounter errors.

Resilience (load test) The average time a request takes to be served and its standard deviation.

For each of the aforementioned quantities we produced a test case using either the Jmeter load generator or custom benchmarks written by us in Perl. What follows is a small description of each benchmark, mainly focusing on the test case we created rather than technical details. Each benchmark corresponds to the above classification of the tested values.

BENCH1 starts a loop that forks children in a rate of a child every 2 seconds. Each child performs requests and then sleeps for amounts of time ranging from 1 to 8 seconds, to simulate the user's "think time". The response size to each request was 73,5 kb, a typical size for modern web sites. If the response does not arrive in a period of 3 seconds, the whole process group is killed. The measured value is the number of clients that were served in conjunctio with the time the benchmark failed, and the average traffic each protocol produced.

BENCH2 starts by forking a client that does HTTP pinging, thus requesting a resource, in particular a script/program which performs a `Select *` query, get the results and count the time taken by the server to respond. The requests are done sequentially, a request must be finished before another starts. The parent program forks a child that performs requests every 10 seconds, until a total number of 20 children are forked. The children print to `stdout` the time taken by each request, allowing us to calculate the average time for a specific number of concurrent clients. The response size was 195kb per request.

BENCH3 was responsible to start threads that perform a single request and then die. As we wanted to run a stress test, we did not specify any delays between requests and allowed them to execute in parallel. We tried testing with 14, 16, 17, 20, 22 and 30 threads per second. We collected the errors either coming from the server or through the socket for each test case. The response size was 50kb in size.

BENCH4 also used the Jmeter load generator. We configured it to start 100 threads in a period of 10 seconds that perform HTTP pinging to the server. Between two requests there is a small 200ms delay. After doing the request, the thread exits. The response time for each request was logged in a file. Using these results we calculated the average response time and the standard deviation for every new response. The sample response that was produced by the server was 50kb in size.

4 Benchmark Results

First of all, we have to draw the reader's attention to the lack of CGI results. Although it was our intention to test and present the results of the CGI protocol, the fact that the results were orders of magnitude different from other technologies (differences of about 1000%), drove us to the decision of not including these results.

Apart from that, we have to notice that all other technologies produced very encouraging results for the server hardware on which the tests were run.

4.1 BENCH1

The BENCH1 benchmark was run for 3 different values of the "think time" property (2, 5 and 8 seconds). The seed for the random number generator was the same in all test cases, to ensure that the children slept for equal amounts of time among tests and thus the requests were exactly the same. We collected the number of clients that were run when the benchmark was killed and presented them in figure 5. Also, the average traffic generated by each protocol is illustrated in table 1.

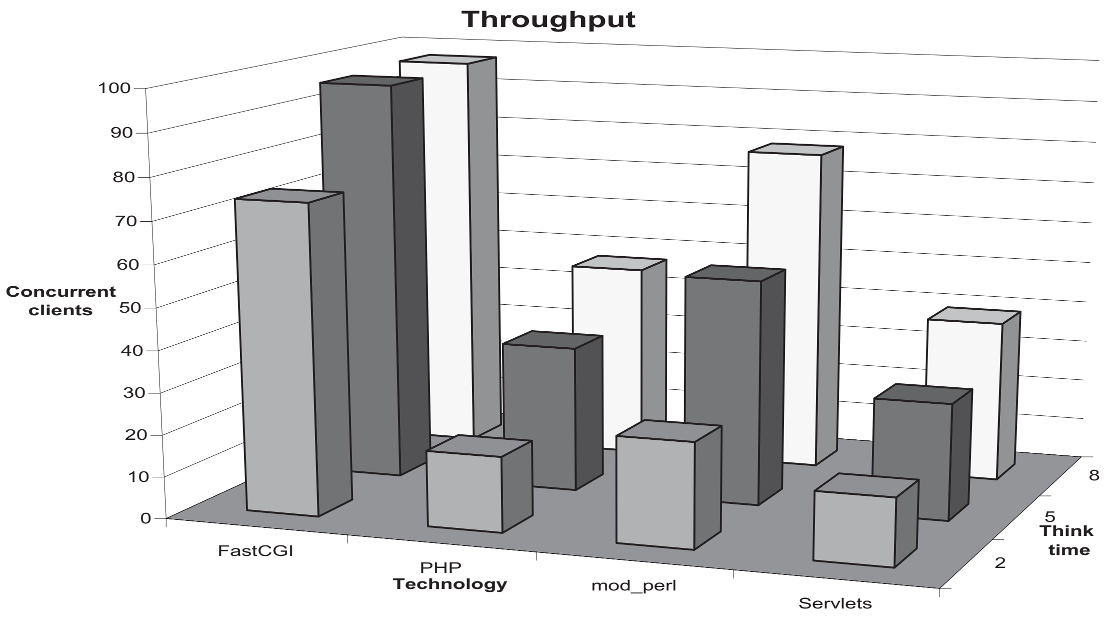

Figure 5: Number of concurrent clients vs think time

Protocol	Traffic
FastCGI	2250
mod_perl	1631
PHP	1248
Servlets	801

Table 1: Average generated traffic in kb/s

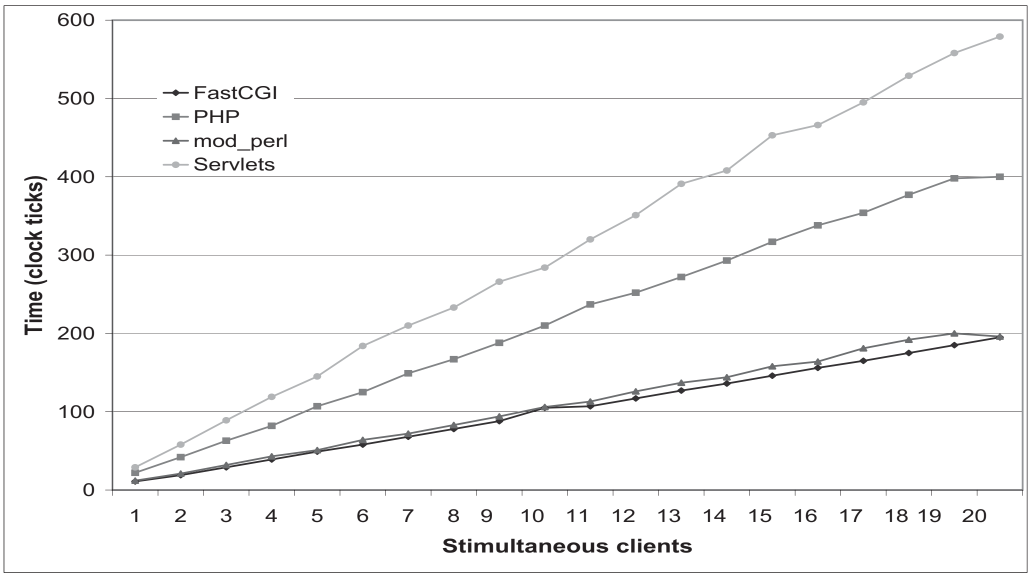

Figure 6: Response latency vs number of concurrent clients

We have to notice here that we were not able to perform the test for more than 97-98 clients because the benchmark program exhausted the physical memory on the client machine. So results for FastCGI are inaccurate in the case of the highest think time. Apart from that, it is obvious that FastCGI is the fastest protocol we tested. It managed to handle more concurrent clients than every other protocol, even when the think time was minimum. It also managed to produce more traffic than every other protocol. Mod_perl also provided satisfactory performance and in fact managed to scale very well when the think time raised, doubling the served clients for a 3 second increase. PHP's performance was characterized by non-linear increase of the served clients as think time increased, but the results are satisfactory if we consider the wealth of features it incorporates. Servlets on the other hand performed rather poorly, managing to serve 75% less clients that FastCGI did.

4.2 BENCH2

The BENCH2 benchmark was run for 4 times (3 warm-up runs and the final). For each number of concurrent clients, we calculated the average time by adding the per request time that was returned by the program and dividing by the total number of requests. We created a graphical representation of the averages we calculated, shown in figure 6.

The figure shows results similar to those of BENCH1. The FastCGI is clearly first, but the gap with mod_perl is narrow. Both protocols provide a linear increase of the response time, having a predictable behavior even with a large number of concurrent clients. On the other hand, PHP and servlets, do not manage to respond fast enough. Servlets were four time slower than FastCGI and PHP was two times slower. The only reason for which PHP behaves better than servlets seems to be the number of concurrently open database connections, which was double than the concurrent clients.

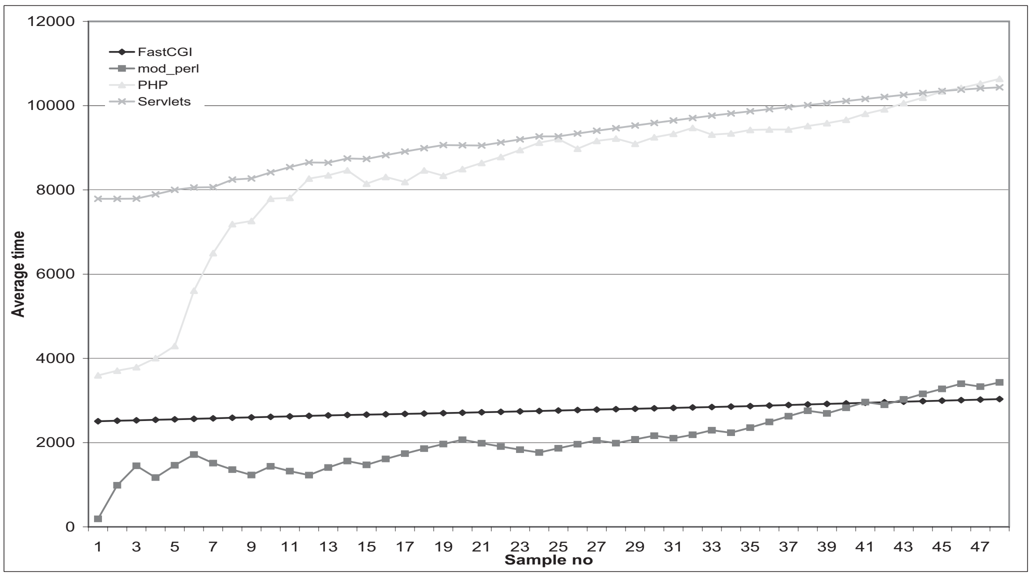

Figure 7: Average time vs response count

4.3 BENCH3

BENCH3's objective was to bring the server down to its knees and make it generate errors. As errors we regarded both the HTTP protocol's 5xx family of headers and the socket errors generated by the server, which were reported as a Java exception by the Jmeter program. We took the risk of including to the total number of errors the socket errors that might have been produced by the client, because it is obvious that if a machine was to produce socket errors this would the server. In the beginning, we started testing with a small number of clients which we soon increased because all protocols behaved very well.

The outcome of this test was that no protocol produced any error in all test cases. Every protocol showed a very stable behavior and we only managed to increase the test time when we raised the number of concurrent threads. We did not manage to produce a single error, although the test was more a like a DOS attack when the number of clients was high, and the server was not responding. This test results proved the well-known stability of both GNU/Linux and Apache, and turned out to support the Apache's development team commitment on stability rather than absolute performance.

4.4 BENCH4

The BENCH4 benchmark's purpose was to test each protocol's ability of having a stable and predictable behavior. So, we exposed each protocol to a medium (according to our experience from BENCH3) traffic of 10 clients per second and measured the response times. For each response, we calculated the average response time by adding the time taken for each response to the sum of the previous response times and dividing with the current response count. We also calculated each response's deviation from the current average. We are only presenting the last 50 results (from a total of 100), because we noticed that all protocols in the beginning have an unstable behavior as a result from Apache trying to fork the necessary children. The results are illustrated in figures 7 and 8.

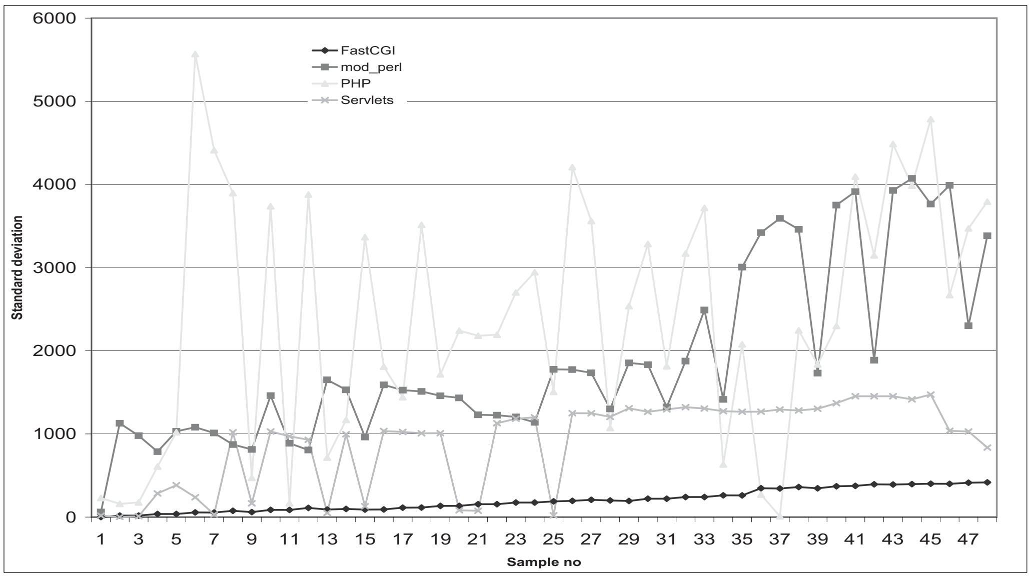

Figure 8: Deviation from the average vs response count

The figures clearly show that once again FastCGI is the fastest and the more predictable protocol we tested. Both the average time for each request and the standard deviation presented an almost linear increase, showing that FastCGI can be used to provide quality of service guarantees. On the other hand, mod_perl and (especially) PHP seemed to suffer from a "lucky-unlucky" syndrome; some clients were "lucky" and were served really fast while others were "unlucky" and waited 3-5 times more time to be served. This behavior is very well imprinted on the deviation diagram, where we can see very acute changes, in comparison to FastCGI's behavior. Also, one of PHP rumored weaknesses, scalability, turned out to be true as shown in the average diagram. The performance of the servlet technology placed it in the middle of our results league. Resilience does not seem to be one of the servlet applications strong points, although, in contrast to PHP, the servlets application exhibited a linear performance degradation.

5 Related Work

Developing, evaluating, comparing, and tuning web servers and content delivery technologies is an active field for both researchers and practitioners. You will find an overview of tools and approaches for developing data-intensive web applications in [2]; according to the presented classification the technologies we examined fall in the web-database programming language and application server category. In [4], the authors provide a walk through technologies available for building enterprise-wide sites, and present their experience on trying to build the 1998 Winter Olympics web site using FastCGI. Also, an insight on accelerating web-services is presented in [7]. In [9], the authors present some general guidelines for performing benchmarks on web applications. If you intend to use CGI or you want to speed up your existing applications, [10] presents an available working solution. [3] is very interesting as a reading, even if you are not going to use servlets.

6 Lessons Learned

Testing the performance of dynamic web-serving technologies of the Apache web server proved to be a worthwhile exercise. The results we obtained were interesting; some even countered commonly held Internet folklore wisdom. Apart from the results we discussed in the previous section, we also gained new insights concerning the performance of web servers based on Apache and GNU/linux. We conclude this paper by outlining the most interesting aspects of what we learned.

- Serving dynamic applications is a memory-intensive task. The fastest memory a server has the better. All web server applications use caching and shared memory to increase performance, so with today's ultra-fast processors the role of memory and the data bus speed in general is becoming more and more important. In our opinion it should be one of the top priorities when deciding for the web server platform.
- The results of PHP were not what we expected. Being exposed to the hype that rules on the Internet about PHP, we expected it to be at least at the second place. It did not scale well (see BENCH4) and exhausted system processing power when it run, leaving it unusable. We must admit that PHP is tightly linked to MySQL, which was not how we used it, but it is our belief that a fast system can be fast irrelevant its environment.
- Servlets on the other hand, behaved exactly as we expected. They run smoothly, Tomcat performed fast enough for a two year old project in the face of a complete redesign, and their behavior was predictable to some extend, and thus easily profilable. The servlet specification, being part of J2EE suite of specifications, would be more appropriate as a framework for large e-commerce sites, where it can be supported by enterprise-wide application servers on systems with lots of processors and memory. In our humble opinion, servlets and Java are not very well suited to quick site development, a field in which PHP excels. Also, we must make clear that for the results presented here we only used Tomcat as a benchmarking platform; other application servers perform much faster (eg see a comparison between Orion and Tomcat at [8]). The servlets platform is very promising and the results presented here may be unfair.
- The application tested was the simplest possible for two reasons: First, in order to be able to transfer it among technologies easily and second because we wanted to measure each protocol's throughput. A real application will have significantly more load on the database, because almost always a dynamic page is generated by several database queries. A single database connection will then be a limiting factor for performance. Even in our case, servlet performance was strangled. A connection pool is almost obligatory in large applications. Connection pooling can be used in other connections too, for example RMI or connections to an email server. To choose the optimal size for a connection pool, a programmer must do a lot of profiling work with stressing benchmarks using the whole range of the available database queries in his site.
- The combination of Apache and GNU/Linux seems invincible in the web serving arena. A relatively low end machine running Apache and FastCGI was able of filling the equivalent of 35 ISDN BRI lines. Should we use a faster hard drive subsystem or a RAID array, the performance could have been significantly higher. During tests, we were interested in seing the effect of the new Linux VM that was embodied in kernels 2.4.10 and after. So, we run the BENCH3 benchmark (it was the heaviest) with mod_perl (the most resource intensive of the four) on the same system with a 2.4.5 kernel several times. We

noticed a difference, which was not above 3-4%, but the system was slightly more responsive to simple console commands like `free` or `ls`.

- The PostgreSQL database did not once betray us. It was robust, scaled very well when the number of connections started to rise, and recovered gracefully after a power failure. It can guarantee data integrity and security. The problem is that it is a heavy application (especially when considering our experience with MySQL) and can consume a lot of hard disk bandwidth. On the other hand it has a lot of important features for enterprise use such as support for very complex queries (useful for producing complex reports), data replication and very good support for custom data types and scripting languages while being almost compliant with the SQL2 standard.

References

[1] The Perl-Apache intergration project. Online http://perl.apache.org.

[2] Piero Fraternali. Tools and approaches for developing data-intensive web applications: A survey. *ACM Computing Surveys*, 31(9):227–263, September 1999.

[3] Jason Hunter and William Crawford. *Java Servlet Programming*. O'Reilly and Associates, Sebastopol, CA, USA, 2001.

[4] Arun Ivengar, Jim Challerger, and Paul Dantzing. High-performance web design techniques. *IEEE Internet Computer*, March-April 2000.

[5] The Jakarta project. Online http://jakarta.apache.org.

[6] Apache module usage. Online http://www.securityspace.com/s_survey/data/-man.200202/apachemods.html.

[7] Sucheta Nandipalli and Shikharesh Majumdar. Techniques for achieving high performance web servers. In *International Conference for Parrarel Processing*. IEEE, 2000.

[8] The orion server benchmarks. Online http://www.orionserver.com/-benchmarks/benchmark.html.

[9] B. M. Subraya and S. B. Subrhamanya. Object driven performance testing of web applications. In *First Asia-Pacific conference on Quality Software*. IEEE, 2000.

[10] Ganesh Venkitachalam and Tzi cker Chiueh. High performance common gateway interface invocation. In *1999 IEEE Workshop on Internet Applications*, San Jose, California, July 1999.

[11] Ganesh Venkitachalam and Tzi cker Chiueh. High performance common gateway interface invocation. Technical Report 60, Computer Science Department, State University of New York at Stony Brook, Stony Brook, NY 11794-4400, USA, 1999. Online http://www.ecsl.cs.sunysb.edu/tr/TR60.ps.Z.