Comparison of Dynamic Web Content Processing Language Performance Under a LAMP Architecture

Musa Jafar
mjafar@mail.wtamu.edu

Russell Anderson
randerson@mail.wtamu.edu

Amjad Abdullat
aabdullat@mail.wtamu.edu

CIS Department, West Texas A&M University
Canyon, TX 79018

Abstract

LAMP is an open source, web-based application solution stack. It is comprised of (1) an operating system platform running Linux, (2) an Apache web server, (3) a MySQL database management system, and (4) a Dynamic Web Content Processor (tightly coupled with Apache) that is a combination of one or more of Perl, Python and PHP scripting languages paired with their corresponding MySQL Database Interface Module. In this paper, we compare various performance measures of Perl, Python and PHP separately without a database interface and in conjunction with their MySQL Database Interface Modules within a LAMP framework. We performed our tests under two separate Linux, Apache and Dynamic Web Content environments: an SE Linux environment and a Redhat Enterprise Linux environment. A single MySQL database management system that resided on a separate Redhat Linux box served both environments. We used a hardware appliance framework for our test configuration, generation and data gathering. An appliance framework is repeatable and easily configurable. It allows a performance engineer to focus effort on the design, configuration and monitoring of tests, and the analysis of test results. In all cases, whether database connectivity was involved or not, PHP outperformed Perl and Python. We also present the implementation of a mechanism to handle the propagation of database engine status-codes to the web-client, this is important when automatic client-based testing is performed, because the HTTP server is incapable of automatically propagating third-tier applications status-codes to the HTTP client.

Keywords: LAMP solution stack, Web Applications Development, Perl, Python, PHP, MySQL, APACHE, Linux, DBI, Dynamic Web Content Processor, HTTP 1.1 status-code.

1. INTRODUCTION

The term LAMP, originally formalized by Dougherty (2001) of O'Reilly Media, Inc., refers to the non-proprietary, open source, web development, deployment, and production platform that is comprised of individual open source components. LAMP uses Linux

for the operating system, Apache for the web server, MySQL for database management, and a combination of Perl, Python or PHP as language(s) to generate dynamic content on the server. More recently, Ruby was added to the platform. Lately, Some deployments replaced Apache with lighthttpd from Open Source or IIS from Microsoft. Depending on the components replaced, the platform is also known as **W**AMP when Microsoft **W**indows replaces **L**inux or **WI**MP when Microsoft **W**indows replaces **L**inux and **I**IS server replaces **A**pache. Doyle (2008) provided more information on the evolution of the various dynamic web content processors framework and the various technologies for web applications development.

Pedersen (2004), Walberg (2007) and Menascé (2002) indicated that to measure a web application's performance, there are many factors to consider that are not necessarily independent. It requires the fine tuning and optimization of bandwidth, processes, memory management, CPU usage, disk usage, session management, granular configurations across the board, kernel reconfiguration, accelerators, load balancers, proxies, routing, TCP/IP parameter calibrations, etc. Usually, performance tests are conducted in a very controlled environment. The research presented in this paper is no exception. Titchkosky (2003) provided a good survey of network performance studies on web-server performance under different network settings such as wide area networks, parallel wide area networks, ATM networks, and content caching.

Other researchers performed application-solution web server performance test comparisons under a standard World Wide Web environment for a chosen set of dynamic web content processers and web servers. These are closer to the research presented in this paper. Gousios (2002) compared processing performance of servlets, FastCGI, PHP and Mod-Perl under Apache and PostgresSQL. Cecchet (2003) compared the performance of PHP, servlets and EJB under Apache and MySQL by completely implementing a client-browser emulator. The work of Titchkosky (2003) is complementary to that of Cecchet (2003). They used Apache, PHP, Perl, Serve-Side Java (Tomcat, Jetty, Resin) and MySQL. They were very elaborate in their attempt to control the test environment. Ramana (2005) compared the performance of PHP and C under LAMP, WAMP and WIMP architectures – replacing Linux with Windows and Apache with IIS.

Although the LAMP architecture is prevailing as a mainstream web-based architecture, unfortunately, there does not appear to be a complete study that addresses the issues related to a pure LAMP architecture. The closest are the studies by Gousios (2002), Cecchet (2003) and Titchkosky (2003).

In this paper, we perform a complete study comparing the performance of Perl, Python and PHP separately and in conjunction with their database connectors under LAMP architecture using two separate Linux environments (SE Linux and Redhat Enterprise Linux). The two environments were served by a third Linux environment hosting the back-end MySQL database. The infrastructure that we used to emulate a web-client and to configure and generate tests was a hardware appliance-based framework. It is fundamentally different from the infrastructure used by the authors just mentioned. The framework allowed us to focus on the task where a performance engineer designs, configures, runs, monitors, and analyzes tests without having to write code, distribute code across multiple machines, and synchronize running tests. An appliance framework is also replicable. Tests are repeatable and easily reconfigured.

In the following sections, we present our benchmark framework and what makes it different. In section three we present our test results. Section four is an elaboration on our methodology and section five is the summary and conclusions of the paper. All figures are grouped in an appendix at the end of the paper.

2. THE BENCHMARK TESTING FRAMEWORK

The objective of this research was to compare the performance of the three dynamic web content processors: Perl, PHP and Python within a LAMP solution-stack under six test scenarios.

The first three scenarios were concurrent-user scenarios. We measured and compared the average-page-response-time at nine different levels: 1, 5, 10, 25, 50, 75 100, 125 and 150 concurrent users respectively. Scenario one is a pure CGI scenario, no data-

base connectivity is required. Scenario two is a simple database query scenario. Scenario three is a database insert-update-delete transaction scenario. For example, for the 25 concurrent users test of any of the above 3 scenarios, 25 concurrent users were established at the beginning of the test. Whenever a user is terminated a new user was established to maintain the 25 concurrent user level for the duration of the test. Under each test scenario, we performed 54 tests as follows:

For each platform (SE Linux, Redhat)
  For each language (Perl, Python, PHP),
    For each number of concurrent users (1,5,10,25,50,75,100,125,50)

      1- configure a test,

      2- perform and monitor a test

      3- gather test results

    End For

  End For

  4- Tabulate, analyze and plot the average-page-response time for Perl, Python and PHP under a platform

End For

The next three scenarios were transactions per second scenarios. The objective of these tests was to stress the LAMP solution stack to the level where transactions start to fail and the transaction success rate for the duration of a test is below 80%. For example, at the 25 transactions per second level of Perl under SE Linux, we fed 25 new transactions per second regardless of the status of the previously generated transactions. If a failure rate of 20% or more was exhibited, we considered the 25 transactions per second as cutoff point and we did not need to go to a test with higher transactions per second. Failures are attributed to time-outs, database deadlocks, or the lack of resources on the web server, database server or the database management system itself. The three test scenarios were the same pure CGI scenario, simple database query scenario, and an insert-update-delete transaction scenario.

Concurrent Connections Test Scenarios

scenario One (Pure CGI, No database Connectivity): For each platform, using each of the content generator languages and each of the configured number of concurrent users, the web clients requested a simple CGI-Script execution that dynamically generated a web page, returning "Hello World". No database connectivity was involved. Figure 1 is a sequence diagram representing the skeleton of the scenario. Figures 4 and 5 are the comparison plots under SE Linux and Redhat Linux of the test scenario.

scenario Two (a Simple Database query): For each platform, using each of the content generator languages and each of the configured number of concurrent users, the web clients requested execution of a simple SQL query against the database, formatted the result, and returned the content page to the web client. The SQL query was:"*Select SQL_NO_CACHE count(*) from Person*". Figure 2 is a sequence diagram representing the skeleton of test scenarios two and three. Figures 6 and 7 are the comparison plots under SE Linux and Redhat Linux of the test scenario.

Scenario *Three (a Database Insert-Update-Delete Transaction):* For each platform, using each of the content generator languages and each of the configured number of concurrent users, the web clients requested the execution of a transaction against the database (which included an insert, an update, and a delete), formatted the result and returned the content page to the web client. Figures 8 and 9 are the comparison plots under SE Linux and Redhat Linux of the test scenario.

Transactions per second test: Scenarios Four through Six

These were stress tests. Scenarios four, five, and six were performed using a constant transactions per second setting rather than the previous concurrent user setting. As stated earlier, the appliance will generate the configured transactions every second independent of the status of the previously submitted transactions. These tests were performed progressively, increasing the transaction rate, until the success rate dropped below 80%. Table 2 is a tabulation of the maximum number of tolerated transactions per second until we achieved the threshold degradation rate.

Test Metrics Gathered

To generate realistic and high volume traffic within a replicable environment, a hardware appliance from Spirent Communication (Spirent 2003) was used, the Avalanche 220EE. This device is designed specifically to assess network and server capacity by generating large quantities of realistic and user configurable network traffic. It implements a client-browser emulator with all the fundamental capabilities of HTTP 1.0 and 1.1 protocol (session management, cookies, SSL, certificates, etc.). Through user defined settings, thousands of web clients from multiple sub networks can be simulated to request services from HTTP servers. For testing purposes, two separate front end LAMP (Linux, Apache Perl, Python, PHP, and database connectors) environments were deployed: an SE Linux installation and a Redhat Enterprise server installation. The two hardware platforms were identical; the configurations of Apache, Perl, Python, PHP and their connectors on the two platforms were as identical as possible. The backend MySQL Database server resided on a separate Redhat Enterprise server. Two identical switches and a router were used for the test environment to manage the appliance, to provide connectivity to the HTTP servers and to the database server. Specification of all test configurations and their management was done using the appliance interface. Investigators did not have to write elaborate shell scripts, distribute the test environment over multiple clients, and manually or programmatically gather results. All of this was accomplished by the appliance and its accompanying analyzer. Figure 3 is a screenshot of the user interface to configure a test by the appliance. Table 1 is a sample test output, See (Kenney 2005) for more elaborate description of the appliance capabilities.

Whether it was concurrent users or transactions per second, each test was composed of 4 phases: (1) a warm-up phase, (2) a ramp-up phase, (3) a 4 minute steady phase and (4) a cool-down phase. The total duration of each test was 6 minutes. For each test performed under the six scenarios, the following data was collected: a pcap log file of all network traffic of the test (a pcap file is a network traffic dump of packets in and out of a network interface of a machine); desired and current load; cumulative attempted, successful and unsuccessful transactions; incoming and outgoing traffic in KBPS; min, max and current time to TCP SYNC/ACK in milliseconds; min, max and current round trip time in milliseconds; min, max and current time to TCP first byte in milliseconds; TCP hand-shake parameters; min, max and current response time per URL milliseconds; and all HTTP status-codes. The appliance gathered these metrics and provided us with the summaries listed in 4-second intervals. In other words, each data point gathered was a summary of a 4-second interval of traffic activities. For example the appliance provided us with summaries of the minimum page response time, average-page-response time, and maximum page response time for each of the 4-second intervals for all users that were served within the interval. Accordingly, for a 6 minutes duration test, we gathered cumulative and real time summary statistics for 90 intervals. This paper presents and compares the average-page-response time of each test cumulatively. Other network traffic statistics (TCP statistics, connection management statistics, etc.) are outside the scope of this paper.

A comparison of our framework with previous studies exemplifies the benefits of conducting network tests using an appliance framework. For example, Gousios (2002) performed four benchmark tests comparing FASTCGI, mod-Perl, PHP and servlets. They used a combination of Apache JMeter ("a desktop application designed to load test functional behavior and measure performance") to load Perl scripts to fork processes and generate load. All the server-side components were run on the same machine, using PostgresSQL instead of MySQL. They did not use a distributed environment. Gousios did not benchmark a complete LAMP framework either. Comparing that framework and the effort required to generate benchmark tests, then gather and analyze the results, lends strong credence to the inclusion of special purpose network appliances to conduct performance analyses. Another reason to use dedicated appliances is that Gousios was "not able to perform the tests for more than 97 to 98 clients because the benchmark program exhausted the physical memory of the client machine" (Gousios, 2002). Gousios used shell-based scripting to gather data and perform calculations and did not use the pcap log contents from a network analyzer to gather data,

which would have made their results more reliable. Cecchet (2003) performed elaborate "performance comparison of middleware architectures for generating dynamic web content" implementing the TPC-W transactional web e-commerce benchmark specification from tpc.org. However, Cecchet's platform was Apache, Tomcat, PHP, EJB and servlets, which is not a complete LAMP architecture. Cecchet implemented an elaborate HTTP client emulator that required laborious scripting. In our case, the appliance provided this functionality. Titchkosky (2003) extended the work of Cecchet (2003), relying heavily on shell scripting for distribution of tests. Also, monitoring and analysis relied heavily on raw network commands (netstat, httpref, sar, etc.) and web server log analyses. Titchkosky's approach was labor intensive, hard to code, hard to reconfigure, and had to be programmed to get the full spectrum of network traffic. In all of these cases, it was unclear how database server timeouts, deadlocks, connection errors, etc. were accounted for, managed and propagated to the HTTP client. In a multi-tier web environment, the reason for failure is not necessarily an HTTP error.

3. TEST RESULTS

Scenario One (Pure CGI, No database Connectivity)

As discussed earlier for each of the concurrent connections benchmark, a total of 54 tests were performed. These are 27 tests for each of the two Linux platforms (9 tests for each of the three languages within a platform). The following is the Perl script code for the scenario. The Python and PHP scripts are similar.

```
#!/usr/bin/perl
print "Content-type: text/html\n\n";
print "<html><head><title>";a
print "Perl</title></head>";
print "<body><H1>";
print " Hello from Perl";
print "</H1></body></html>\n";
```

Figures 4 shows the average page response time for a web-page request in milliseconds averaged for the duration of the six minutes test under SE Linux. The horizontal axis records the "number of concurrent users" of the test; the vertical axis measures the "average-page-response time" for that test. Figure 5 shows the same results under Redhat Linux. Under both environments, PHP outperformed Perl and Python. The Linux flavor was not a factor in these tests. In looking at the graphs, one can see that the test results for both SE and Redhat Linux were almost identical.

Scenario Two (Simple database query)

As previously described, test scenario two adds a simple database query to scenario one. The following is the Perl script code for the scenario. The Python and PHP scripts are similar. Uniquely identifying names have been replaced by xxx. Line breaks were added to beautify the two columns format.

```
#!/usr/bin/perl
use strict; use DBI(); use warnings;
my $sqlStatement =
  "Select SQL_NO_CACHE count(*)
              from perlperson";
my $dsn =
  "DBI:mysql:database=xxx;host='xxx';port
    =xxx";
my $dbh = DBI->connect($dsn,
  "xxx","xxx",{PrintError=>0});
if(my $x = DBI::err) {
    responseBack(501,
              "Unknown $x DBI Error");
}
my $sth=$dbh-> prepare($sqlStatement);
if(my $x = DBI::err) {
    responseBack(501,
              "Unknown $x DBI Error");
}
my $mysh = $sth->execute();
if(my $x = DBI::err) {
    responseBack(501,
              "Unknown $x DBI Error");
}
my $result;
my $resultSet =
 "<h1>Welcome to Perl-MySQL<h1><br>";
while($result =
        $sth->fetchrow_hashref()){
        $resultSet =
        "$resultSet + $result+ <br>";
  if(my $x = DBI::err) {
      responseBack(501,
              "Unknown $x DBI Error");
  }
  }
  $sth->finish();
  if(my $x = DBI::err) {
    responseBack(501,
```

```
                "Unknown $x DBI Error");
  }
  $dbh->disconnect();
  responseBack(200, "OK", $resultSet);
sub responseBack {
    my $err = 501;
    my $message =
      "Connection to Database Failed";
    my $results = "$err  $message";
    if(@_){
      ($err, $message, $results) = @_;
    }
    print("status: $err $message\n");
    print("content-type: text/html\n\n");
    print("<html>\n");
    print("<head>\n");
    print(
      "<title> $err $message </title>\n");
    print("</head>\n");
    print("<body>\n");
    print("<h1> $results </h1>\n");
    print("</body>\n");
    print("</html>\n");
    exit;
  }
```

Figures 6 and 7 show the results of the tests under SE Linux and Redhat Linux. Under both environments, PHP coupled with its DBI significantly outperformed Perl and Python. For the SE Linux environment, however, both Perl and Python exhibited no difference in performance. Python DBI performed a lot better under the Redhat environment than the SE Linux environment.

Scenario Three (Concurrent Insert-Update-Delete)

Scenario three adds the transaction overhead of updates and delete to the database. The following is the PHP script code for the scenario. The Python and PHP scripts are similar.

```
<?php
  $link = mysql_connect('xxx', 'xxx', 'xxx');
  $errNo = -1;
  $err = "Connection Failed";
  if(!$link){
    if(mysql_errno($link)){
      $errNo = mysql_errno($link);
    }
    if(mysql_error($link)){
      $err = mysql_error($link);
    }
    header("HTTP/1.1 501 $errNo $err");
    header("Content-Length: 0");
    print("HTTP/1.1 501 $errNo $err\n");
    exit;
  }
  if(!mysql_select_db("xxx",$link)){
    $errNo = mysql_errno($link);
    $err = mysql_error($link);
    header("HTTP/1.1 501 $errNo $err");
    header("Content-Length: 0");
    print("HTTP/1.1 501 $errNo $err\n");
    exit;
  }
  function microtime_float() {
    list($usec, $sec) =
        explode(" ", microtime());
    return ((float)$usec + (float)$sec);
  }
  function sendError($errNo,$err){
    header("HTTP/1.1 501 $errNo $err");
    header("Content-Length: 0");
    print("HTTP/1.1 501 $errNo $err\n");
    exit;
  }
  $currTime = microtime_float();
  $sql ="insert into PHPTestInsert(ts)
                        values($currTime)";
  $result = mysql_query($sql, $link);
  if(!$result) sendEr-
ror(mysql_errno($link),mysql_error($link));
  $sql = "select 1 from PHPTestInsert
        where ts = $currTime for UPDATE";
  $result = mysql_query($sql, $link);
  if(!$result)
    sendError(mysql_errno($link),
                mysql_error($link));
  $sql = "update PHPTestInsert set volume
      = volume + 1 where ts = $currTime";
  $result = mysql_query($sql, $link);
  if(!$result)
sendError(mysql_errno($link),
                mysql_error($link));
  $sql = "delete from PHPTestInsert
                where ts = $currTime";
  $result = mysql_query($sql, $link);
  if(!$result) sendError(mysql_errno($link),
                mysql_error($link));
  print("<HTML> <HEAD><TITLE>”);
  print(“Hello from PHP”);
  print(“</TITLE> </HEAD>”);
  print(“<BODY><H1>”);
  print(“Hello PHP”);
  print(“</H1> <br>");
  print("</BODY> </HTML>");
  mysql_free_result($result);
  mysql_close($link);
?>
```

Figures 8 and 9 show the results of the tests. They are strikingly similar to those of scenario two.

Scenarios Four through Six (Transactions per Second Stress Tests)

As explained earlier, the same test scenarios as those for one, two and three were repeated with the difference being that a constant level of concurrent transactions was replaced with a fixed rate of transaction submission. After each test run, the rate was incremented until the LAMP environment under test exhibited a transaction failure rate over 20%. Table 2 shows the results for these three test scenarios. Both results of SE Linux and Redhat Linux are similar for each language. The tolerable TPS rate is the highest rate achieved before exceeding the 20% transactions failure rate. Again in all cases, PHP outperformed Perl and Python.

4. ADDITIONAL OBSERVATIONS

Multi-tiered, HTTP based applications suffer from the inability of the HTTP protocol to propagate meaningful error and status information from a back-end tier, such as the data tier, to the presentation tier – the web client. Web clients communicate with a web server only. Standards for communicating server status information to the client were defined as static status-codes ranging from 1XX to 5XX by W3C protocol including the classic "404 Not Found" status-code (RFC-2616, 1999, HTTP 1.1). These values are numeric and hard coded. They are not extensible to allow for the encoding of status information from a back-end tier such as a database server. For example, if the MySQL database engine could not process a DBI request and returned a database failure XXXX error code to the DBI, it is the responsibility of the DBI processor (Perl, Python, PHP, etc.) to trap the failure code, interpret, handle it and propagate something to the client. For automatic testing this is a problem. Yet, with the growth in cloud computing and their associated behind-the-scenes HTTP request technologies such as AJAX, recognition and handling of non-HTTP status information needs to be more extensively defined. In this paper we suggest a framework for propagating remote tier errors to the client, reported as a special category of HTTP error code that is not within the range of our experiment codes. To trap database failures we forced an out of range HTTP error in scripting code executing on the server. We did this by selecting the "501 Not Implemented" status-code as a catch all, then we used this HTTP 501 error in our analysis to indicate a failure of the DBI. We could have added more resolution by mapping different database errors to different 5XX HTTP status codes. In our case, it was not necessary. All that we needed was to know was when a database failure status had occurred. Without this additional status information, database failures would have been passed and viewed as normal transactions by the HTTP client. The following is a snippet of Perl code to accomplish the task. PHP and Python codes would be similar.

```
#!/usr/bin/perl
use strict;
use DBI();
……
if(my $x = DBI::err) {
  my $y = DBI::errstr;
  responseBack(501, "$x $y DBI Error");
}
sub responseBack {
  my $err = 501;
  my $message =
   "Connection to Database Failed";
   my $results = "$err  $message";

   if(@_){
    ($err, $message, $results) = @_;
   }
  print("status: $err $message\n");
  print("content-type: text/html\n\n");
  print("<html>\n");
  print("<head>\n");
  print("<title> $err $message</title>\n");
  print("</head>\n");
  print("<body>\n");
  print("</body>\n");
  print("</html>\n");

  exit;

}
```

By pushing the error into the header section, the web client of the appliance was able to count it as failure and incorporate it into the statistics and summaries. We think that HTTP return codes need to be dynamic and extensible to allow for the propagation of

multi-tier and vendor specific error and status code information to the HTTP client for handling. This can be accomplished through a well defined XML schema for HTTP that the server can propagate and the client is capable of interpreting.

5. SUMMARY AND CONCLUSIONS

In this paper, performance measures of Perl, Python and PHP under two LAMP architectures were presented. Our tests clearly indicate that PHP outperformed Perl and Python. When no database connectivity was involved (Scenario 1) both Linux platforms exhibited comparable results for the same language. When database connectivity is involved (Scenarios 2 and 3), PHP remained stable, Python exhibited high performance degradation with comparable results for both Linux platforms. Perl also exhibited high performance degradation, with better performance under Redhat than SE Linux. Analyzing the results (Scenarios 2 and 3), most of the performance degradation was due to the establishment of the connection with the database server. With respect to transactions per second (Scenarios 4, 5, and 6) PHP and Perl had comparable results when no database connection was involved. PHP outperformed Perl and Python otherwise.

An appliance framework allows performance engineers to focus on the task at hand. Tests can be replicated, reconfigured and reproduced in a matter of minutes through the graphical user interface of the appliance. The framework transforms the job of the performance engineer from a programmer to that of a test designer and analyzer.

Automatic testing exposed the limitations of the HTTP protocol in propagating third-tier status codes such as database connection timeouts, deadlocks and violations as HTTP header information to the web-client for interpretation.

Finally, designing an enterprise solution based on the performance of a language alone is an oversimplification. Solution architectures are complicated. Backend processing, consolidation of data across multiple domains, robustness, security, scalability and services processing are at the core of enterprise solution architectures. Although the performance of a language is important, other factors need to be considered when selecting languages for an enterprise application.

6. REFERENCES

Brebner, P. Cecchet, E. et. al. (2005) Middleware benchmarking approaches, results experiences. Concurrency and Computation: Practice and Experience, Wiley InterScience 17:1799-1805

Cecchet, E. Chanda, A. et. al. (2003) Performance Comparison of Middleware Architectures for Generating Dynamic Web Content. 4th ACM/IFIP/USENIX International Middleware Conference, June 2003, 16-20.

Dougherty, D. (2001) LAMP: The Open Source Web Platform. www.onlamp.com/pub/a/onlamp/2001/01/25/lamp.html

Doyle, B. Videira Lopes, C. (2008) Survey of Technologies for Web Application Development. eprint arVix:0801.2618v1

Gousios, G. (2002) A Comparison of Portable Dynamic Web Content Technologies for the Apache Server.SANE 2002: 3rd International System Administration and Networking Conference Proceedings, May 2002 103–119 (Best Paper Award)

Jansons, S. and Cook G. J. (2002) Web-Enabled Database Connectivity: A Comparison of Programming, Scripting, and Application-Based Access. Information Systems Management 19:1, 14-22.

Kenney John. Avalanche Load Generation: How to improve your Rate-based Tests. www.spirent.com/documents/3426.pdf

Menascé, D. (2002) Load Testing of Web Sites. IEEE Internet Computing vol. 6, no. 4.

Pedersen, A. (2004) Introducing LAMP Tuning Techniques. www.onlamp.com/pub/a/onlamp/2004/02/05/lamp_tuning.html

Ramana, U. V. Prabhakar, T. V. (2005) "Some Experiments with the Performance of LAMP Architecture" Computer and Information Technology, 2005. CIT 2005. The Fifth International Conference on 21:23, 916-920

RFC-2616 (1991) Hypertext Transfer Protocol – HTTP/1.1. http://www.ietf.org/

Spirent Communication (2003) Avalanche Analyzer User Guide. www.spirent.com/

Spirent Communication (2003) Avalanche-220EE User and Administrator guides. www.spirent.com

Titchkosky, L. Arlitt, M. Williamson C. (2003) A Performance Comparison of Dynamic Web Technologies. ACM SIGMETRICS Performance Evaluation Review 31:3, 2-11

Walberg S. A. (2007) Tuning LAMP Systems, Part1: Understanding the LAMP. www.ibm.com/developerworks/library/l-tune-lamp-1/index.html#resources

Walberg S. A. (2007) Tuning LAMP Systems, Part2: Optimizing Apache and PHP. www.ibm.com/developerworks/web/library/l-tune-lamp-2.html

Walberg S. A. (2007) Tuning LAMP Systems, Part1: Tuning Your MySQL Serve. www.ibm.com/developerworks/linux/library/l-tune-lamp-3.html

7. APPENDIX

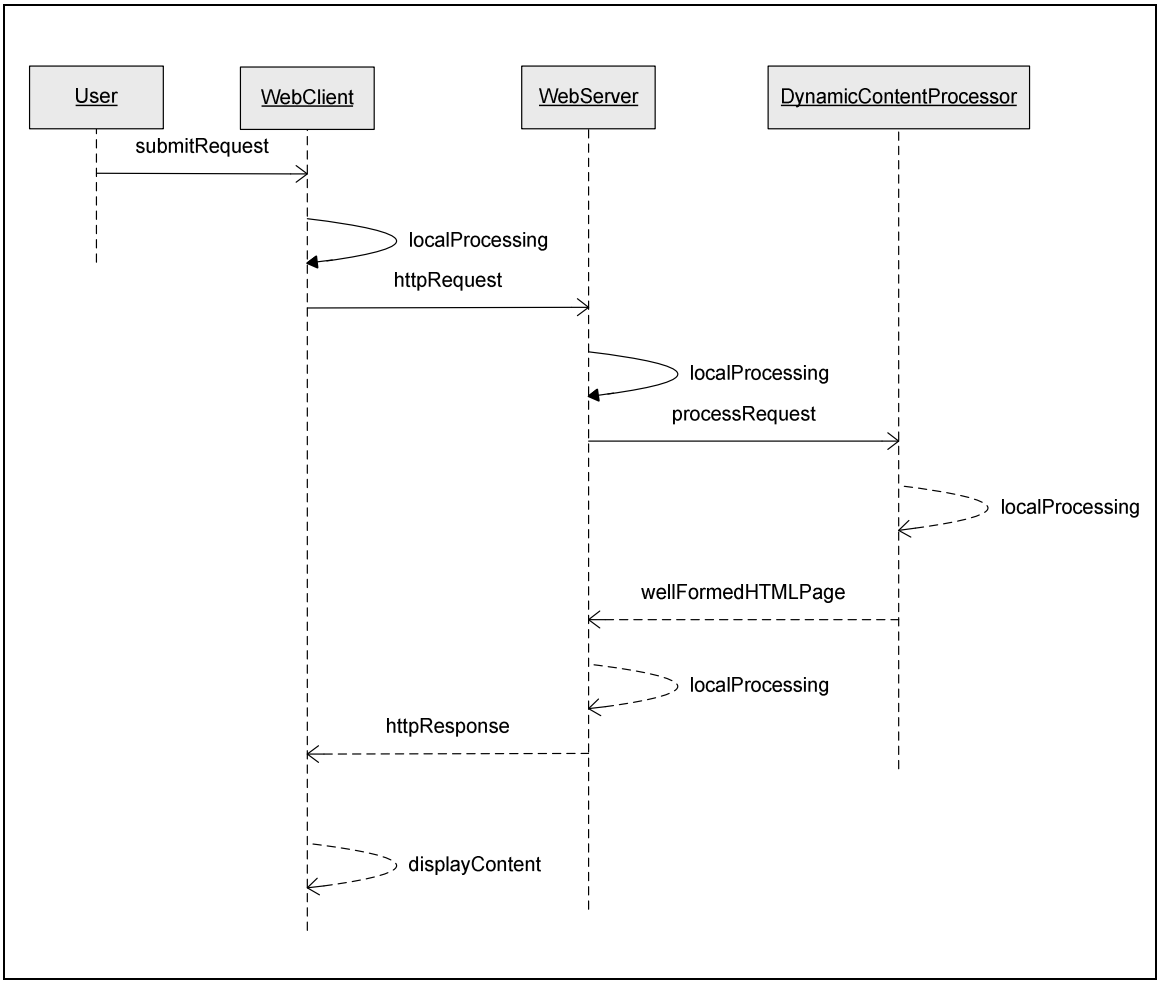

Figure 1: Sequence Diagram for Test Scenario One

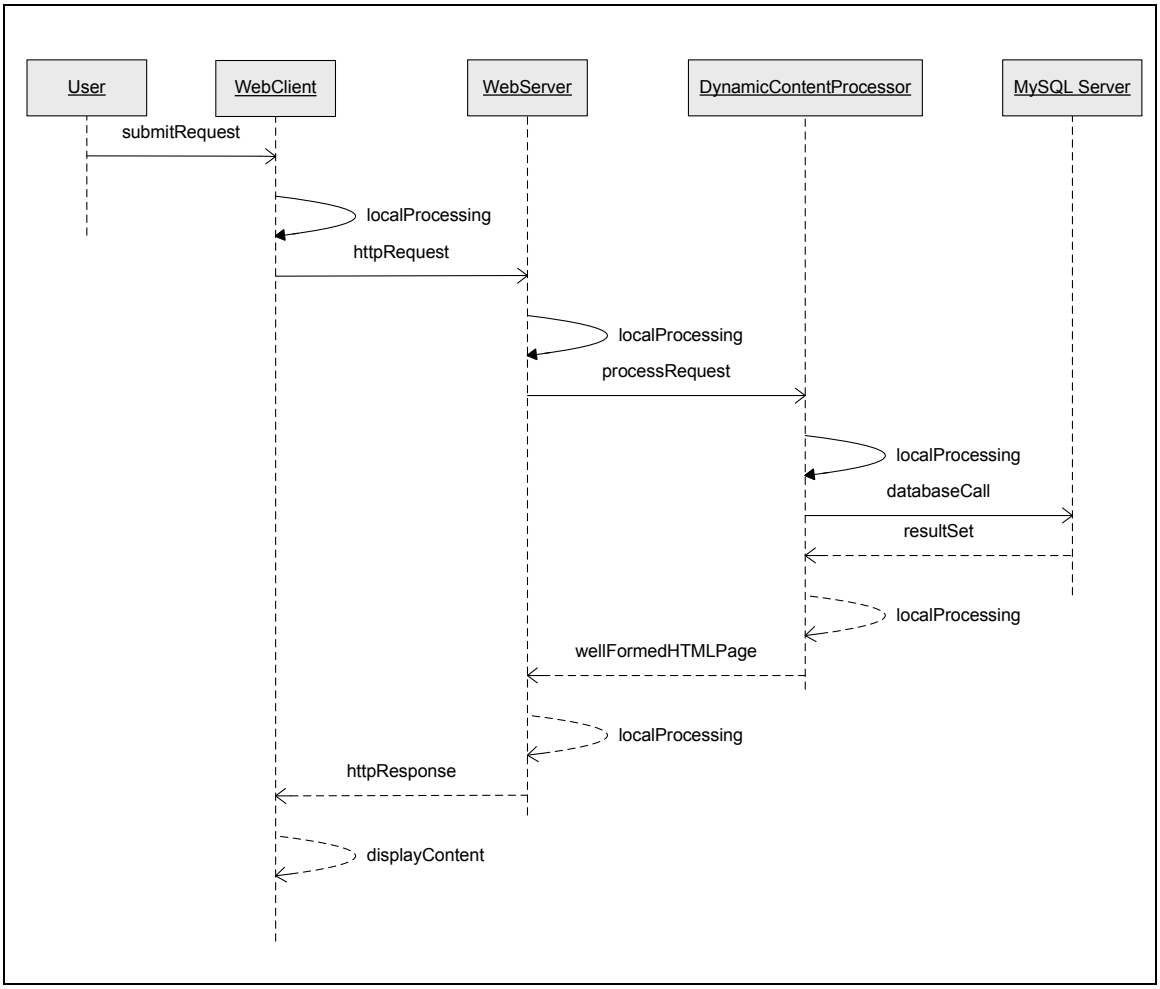

Figure 2: Sequence diagram for test scenarios two and three

Table 1: Avalanche Analyzer Report Table

	Transactions				Time (ms)				TCP Connections	
Test Results Summary		Total	Rate Per Second		Page Response	To TCP SYN/ACK	To First Data Byte	Est. Server Response		Total
	Attempted	17782	80	Minimum	4.0	0.117	4.974	0.0	Attempted	17782
	Successful	17782	80	Maximum	144.0	108.854	144.767	144.175	Established	17782
	Unsuccessful	0	0	Average	8.0	0.184	8.424	8.072		
	Aborted	0	0							

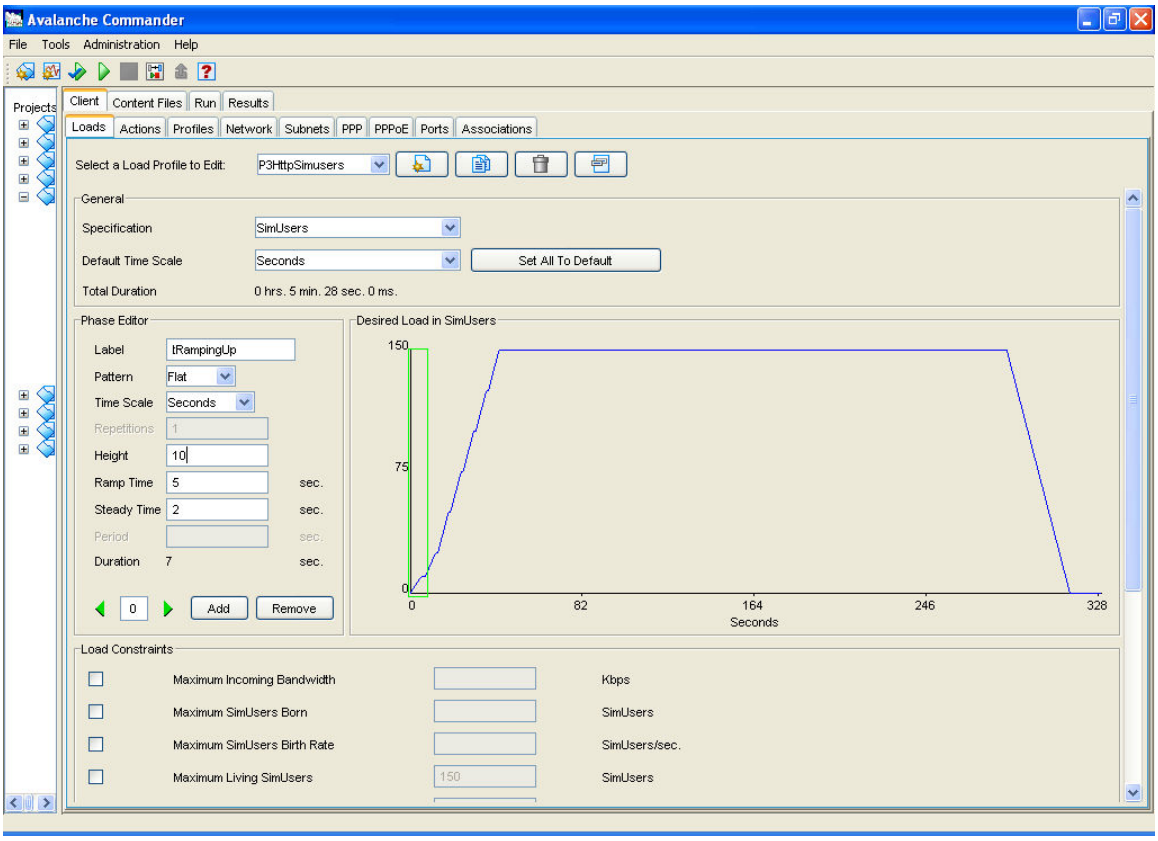

Figure 3: Avalanche Commander Interface

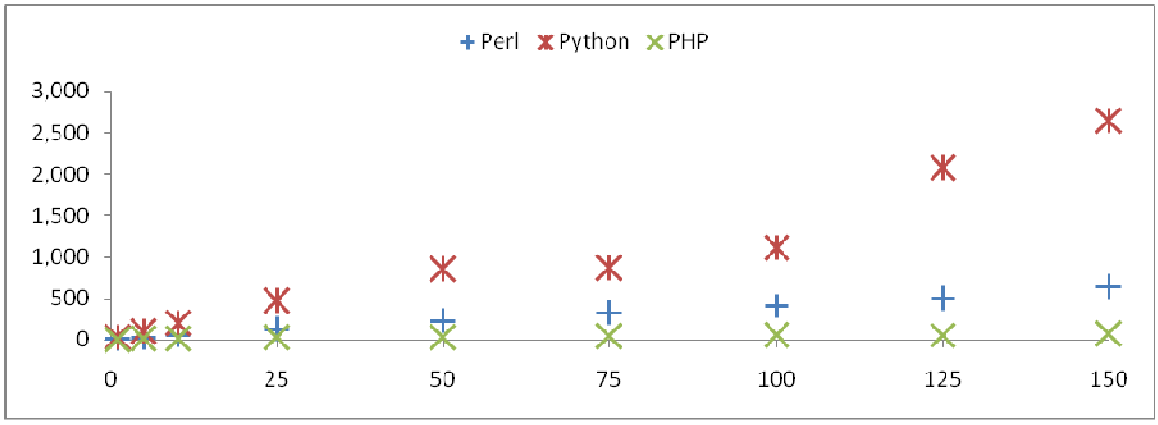

Figure 4: SE Linux, Dynamic Web Page Request

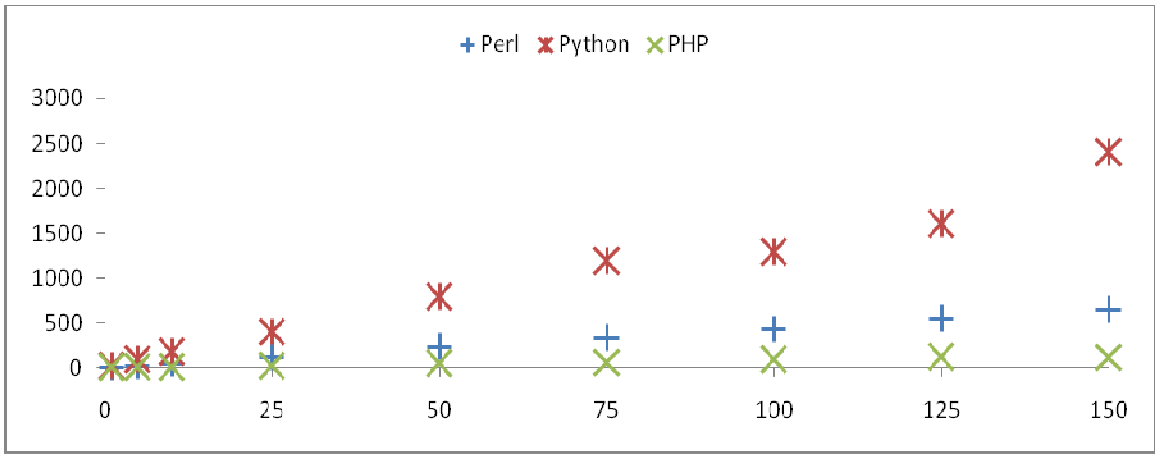

Figure 5: Redhat Dynamic Web Page Request

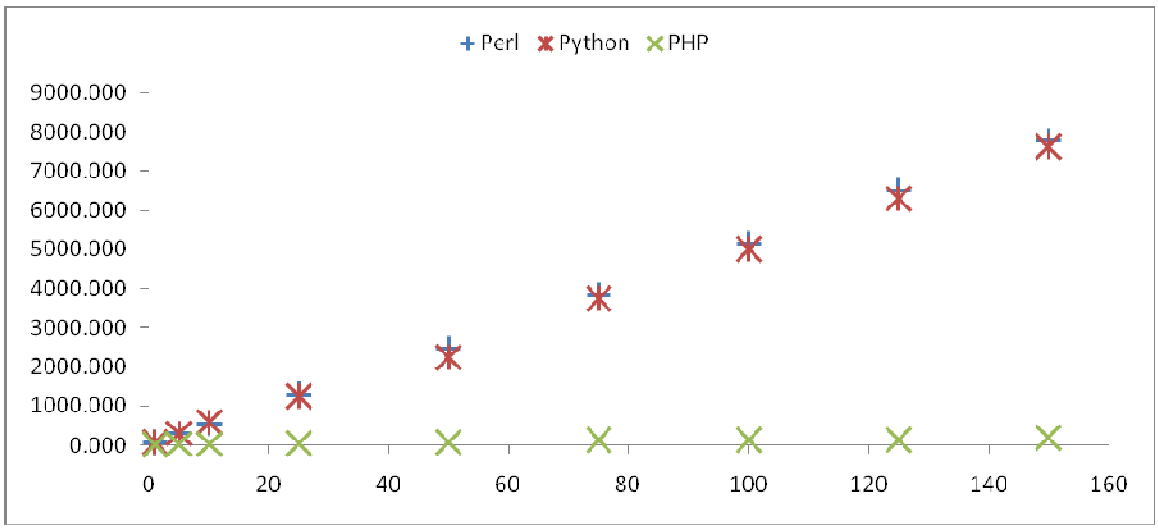

Figure 6: SE Linux, Simple Query Request

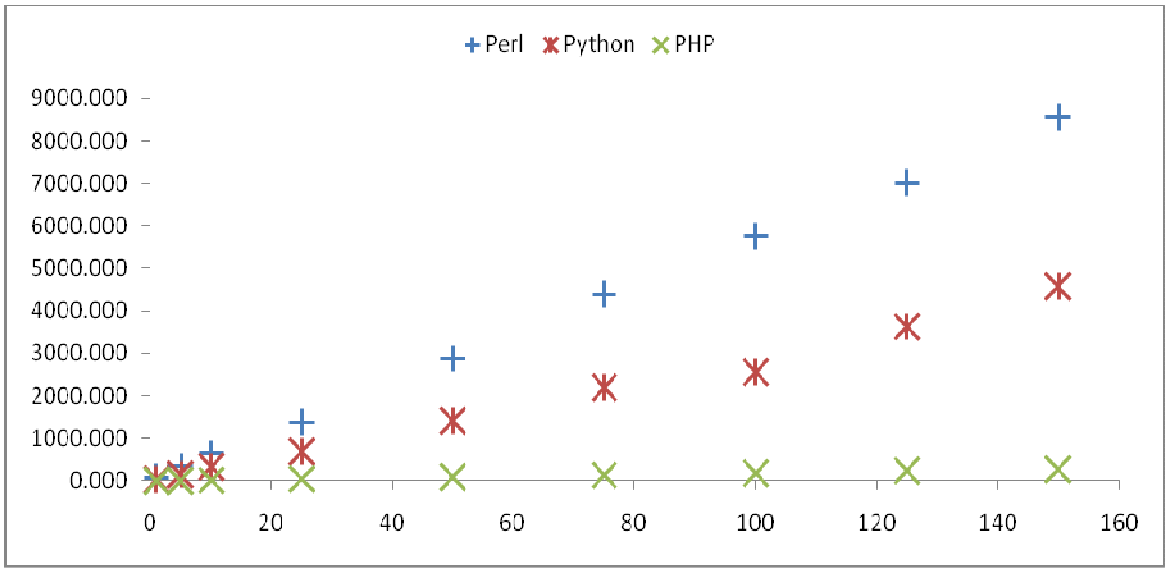

Figure 7: Redhat Simple Query Request

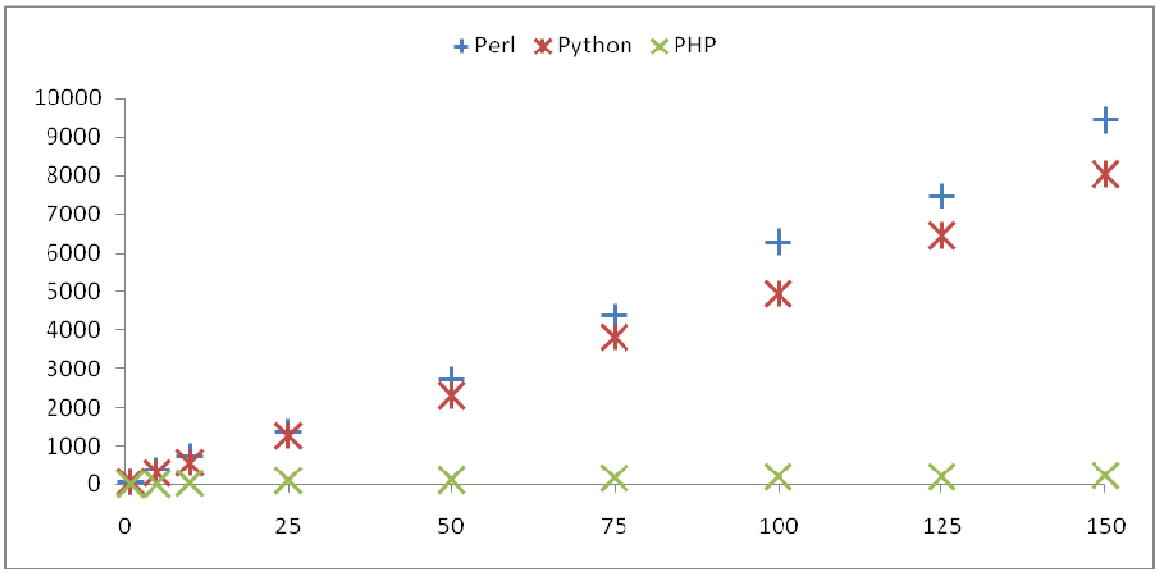

Figure 8: SE Linux, Insert, Update, Delete Transaction

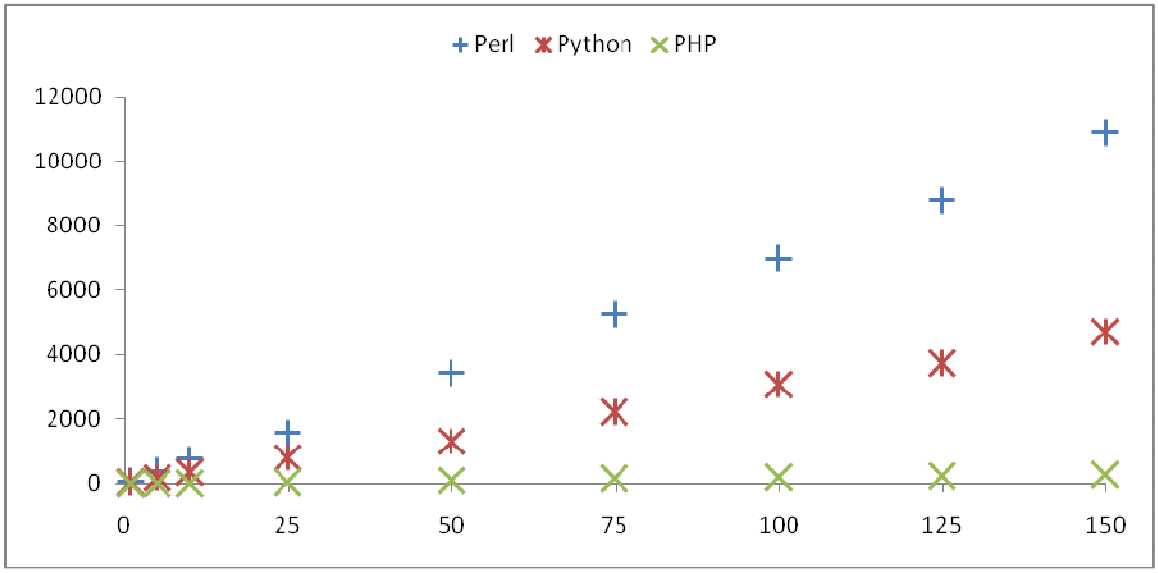

Figure 9: Redhat Insert, Update, Delete Transaction

Table 2. Max Tolerated TPS

	HTTP Simple Request – Dynamically Generated "Hello World" only	Simple database query	Concurrent Database Insert, Update and Delete
Perl	100 TPS	15 TPS	10 TPS
Python	25 TPS	15 TPS	15 TPS
PHP	100 TPS	50 TPS	50 TPS