Performance and Usability Analysis of Varying Web Service Architectures

Michael Lenner
Department of Computer Science
Columbia University
mml2108@cs.columbia.edu

Henning Schulzrinne
Department of Computer Science
Columbia University
hgs@cs.columbia.edu

Abstract

We tested the performance of four web application architectures, namely CGI, PHP, Java servlets, and Apache Axis SOAP. All four architectures implemented a series of typical web application tasks. Our findings indicated that PHP produced the smallest delay, while the SOAP implementation produces the largest.

1. Introduction

As the World Wide Web grew, it became clear that simply serving static HTML web pages would not be adequate to support the complex needs of the various new and emerging web applications.[3] In the past, HTML web servers followed the basic HTTP protocol [11] to receive a client request, and return a static document with HTML content. In this design, the HTML webpage was a physical file the web server accessed and returned. Because of this, all HTML content had to be created beforehand and then saved where the web server would have access to it. Not only was this impractical for web applications that needed to display frequently changing data or large volumes of data with similar structure, but it also presented storage issues as the amount of static HTML data that was required to be saved increased.

The introduction of server-side processing that produced dynamic HTML content was the solution. One of the first such architectures to become a standard was the Common Gateway Interface (CGI).[15] CGI allowed web developers to author a script that would run automatically on the web server when the proper URI was accessed. This script could read input parameters from the user, access a server side data source, and of course, produce dynamic HTML to be returned to the requesting host.

Following CGI, there were other such server side processing architectures that were developed. PHP [12], which provided all the functionality that CGI did, provided the ability to embed dynamic instructions within a static HTML document, such that the web server would execute a sort of preprocessor before sending a response to the client. While both PHP and CGI provided powerful functionality to a web developer, they lacked the robustness of a more powerful, higher level language.

To allow a web application to harness the power of the more powerful languages, a number of technologies were developed. One of these was Java servlets. Java servlets are applications written in Java that run on a Java application server. An application server is a web server that is also capable of running high level language applications to produce dynamic HTML content for the requesting client. Additionally, the application server provided not only the framework in which to run these servlets, but also a wide array of supporting functionality for customizing the web application behavior. To name only a few, application servers provide the ability to use load balancing functionality, persistent and non-persistent objects, and transactional processing. These advancements gave web developers the power to write web applications that match the functionality of a standalone program.

Finally, one of the more recent advancements in web applications has been the XML Simple Object Access Protocol (SOAP) standard.[13] Distinguishing this technology from the previous three, this advancement was not in the form of a new means for server side processing resulting in dynamic HTML content. What SOAP brought to the table was a new way for a client and host to

communicate complex data. While the previous three technologies gave much more power to server side processing, the output to the client was not standardized beyond the HTTP specification. The SOAP standard created a protocol for the data contained within the HTTP requests and responses that allowed web applications to communicate more complex data than simply HTML tags.[3] Additionally, through the use of the Web Service Description Language (WSDL) [14] standard, the SOAP architecture allowed for a universally accepted format for exposing a web application for automatic access via a SOAP client.

2. Performance Testing Environment

The above mentioned four web service architectures comprised the subset which will be tested. All applications will be tested in their ability to receive an input parameter from the user, process that parameter, and respond to the user with dynamically created content. CGI, PHP, and Java servlet returned HTML, while the SOAP web service returned XML.

The overall metric measured in these time trials was response time. That is, the start time of the access was compared against the end time of the response for the overall response time. No measurements of throughput for the tested architectures was made.

2.1 Testing Setup

In order to create a controlled environment in which to test these architectures, the following setup was used:

1) Web Server: The web server used was the Apache HTTP server, version 2.0.54. By default, the Apache web server is capable of handling CGI scripts.
2) PHP Module: Using the Apache web server's ability to use dynamically loadable shared modules, the PHP shared module was loaded into the Apache web server for PHP support. PHP version 5.0.4 was installed.
3) Application Server: We used the Jakarta Tomcat application server, version 5.5.
4) Apache-Tomact Connector: To keep the testing environment as close to a control as possible, We forced all client requests to go through the Apache web server, including those that are requesting URL's served by the application server. This required the installation of another shared module into the Apache web server to allow for communication via the web server and the Tomcat application server. We installed the mod_jk2 connector. [6]
5) Database Server: Since some of the benchmark functionality testing included database access, we installed the MySQL database server, version 4.1.12. [9]
6) SOAP Server: Installation of a SOAP implementation was required to respond to the SOAP requests. We used the Apache Java Axis SOAP implementation, version 1.2. Axis is actually a Java servlet that runs within the Tomcat application server.

Below is a diagram showing the overall data flow of the testing environment.

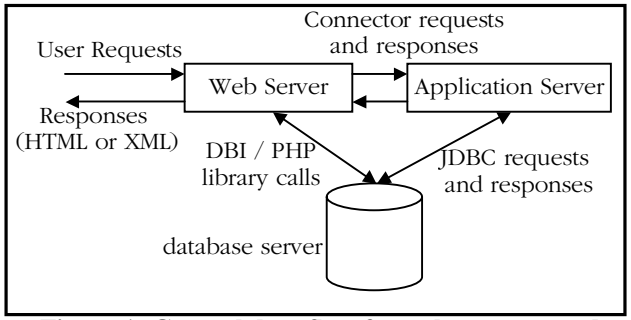

Figure 1: General data flow for web requests and responses

As shown, all incoming requests pass first through the web server. In the case of CGI and PHP, the processing is done within the web server, including access to the database if necessary. Servlets and SOAP applications, requests are passed to the application server, which in turn sends the results back to the web server. All responses are sent back to the requesting host via the web server.

2.2 Web Service Benchmark Functionalities

As stated, we used a suite of conceptual tasks that each of the four tested architectures implemented.[1][2] Each task was meant to exercise that architectures' ability to accomplish a commonly used web application design. What the tasks have in

common is that each requires the client to supply one input parameter, which is read by the web application. Then, the application responds to the client with either an HTML or XML message. We used a total of four tasks in the benchmark suite.

Simple Database Access: Implementations of this task read in one parameter from the user's HTTP request. This parameter was used as part of a single select query accessing one table of the database. The returned result consisted of *one* tuple from that database table, the specific tuple depending on the input parameter from the client. The result was then returned as an HTML web page containing the data within the returned tuple. In the case of the SOAP implementation, the returned data was an XML encoded string array, each string containing one item from the database tuple.

Large Database Access: This task is much the same as the previous, except for the fact that a large data set is returned to the user. Whereas the previous task returns one tuple, this task returns data on the order of 15,000 tuples. Again, the specific data is dependent on a user input parameter contained within the HTTP request. As was the case with the previous task, a simple select query was used on one table in the database. All data returned was via an HTML response, except, of course for the SOAP implementation. In this case, raw XML (within the SOAP envelope) containing the database tuples was returned.

Large File Access: Implementations of this task were required to read in one large text file (approximately 4 MB) and return the text to the user. The selection of the file was determined by a user supplied input parameter contained within the HTTP request. Results were returned via an HTML response except for the SOAP implementation which returned a single XML encoded string containing the text.

Null Operation: In order to aid in properly calibrating the performance results, we implemented a null task using all four architectures. This task requires no input from the requesting host and simply returns a "Hello World!" string back to the requesting client. In calibration, normally a true non-operation would be ideal, however for the web application domain this type of function does not exists. Regardless, the data provided still provides a valid starting point for performance testing and evaluation.

2.3 Gathering Performance Metrics

The issue most often arrived at in attempting to measure the performance of these applications is in trying to maintain a controlled environment. To that end, testing was done at specific points in the data flow of a URL access common to all four architectures. As shown in Figure 1, all accesses arrive via the web server and all responses are sent via the web server. Therefore, we used the web server as the ultimate authority in terms of response times of the varying architectures.

Upon reception of the initial HTTP request, the web server logs the access as well as the time the access was received. This gives us a valid start point at which to begin measuring the web applications response time. Once the web server has served the request back to the requesting host, a log entry, with the current time, is made once again. This is used as the completion of the web applications' response time.

As shown in Figure 1, a testing strategy such as this does penalize the applications that run on the application server as compared to those that run on the web server. However, in testing we feel that this penalty is warranted, as these applications are required to run on such an application server, which is a necessary level of complexity the other architectures do not have.

To accurately record these start and end times for each web application access, we took advantage of the stages contained within the request loop the Apache web server uses to serve requests. [4] Once the web server receives an incoming request, it reads the HTTP headers and verifies it is a valid request. The very next step is what is called the Post Read Request Phase, and shared modules can be written to be activated at this stage. For testing, we wrote a module that is activated during post read request. The module simply reads the URL being accessed, and logs the access to a file with the current time. This method gives a valid start time.

In the request loop, the final stage, which occurs directly after the response is served, is the logging stage. However taking advantage of this stage is not as simple as using the Apache custom log feature to record time stamps. Doing so will produce erroneous results as the *current* time cannot be accessed via the custom log format, only the *access* time, which is recorded at some earlier point

in the request loop cycle. Instead, we forced the custom log to pipe to a small application that recorded the URL as well as the current time, giving an accurate time for the completion of the response phase of the request loop.

2.4 Testing Environment Controls

Further measures were taken to ensure as close to a controlled testing environment as possible. First, all server daemons were run on a single host. This was done as to remove network latency from the performance results. While a case could be made that having a web, application, and database server all running on one host could introduce possible scheduling delays, these factors were deemed far less significant than those that would arise from running on separate hosts.

Additionally, no web browser was used in accessing the web applications. This was done for two reasons. The first being that the addition of the web browser into the data flow could add unnecessary variables as the web server's logging of the completion of the response may depend on the browser's ability to receive the response. Second, the SOAP applications could not be accessed via the web browser (at least not directly). Instead, all applications were accessed via simple client code writing in Java, running on the same host that all the services were deployed on.

3. Performance Testing Results

Using the testing environment described above, a series of tests were executed on the implemented architectures using the benchmark suite of web application tasks. Two separate clients, both written in Java, were used to access the various web applications deployed on the web and application servers. The first client accessed those web applications that were exposed via the HTTP Post method. These included the CGI, PHP, and Java servlet. The second client was used to send SOAP requests and receive SOAP responses. Both clients made one hundred accesses to the various web applications per test execution to provide a valid sample size for performance metrics.

As accesses were made by each client, a log file was built which contained the initial request time as well as the final response time for each URL access. Once complete, a simple Perl script was used to analyze the log and provide performance statistics for each URL.

3.1 SOAP Access Issues

In performing the access time trials, it was discovered that the SOAP implementation was incapable of handling datasets on the order that the Large Database and File Access tasks required. This was not due to any limitations in the Axis SOAP servlet that was processing all SOAP requests but rather to the testing environment itself.

As stated earlier, all requests came in via the web server, and, if necessary, were forwarded to the application server (and returned back) via an Apache to Tomcat connector. It was at this point in the data flow that the SOAP implementation broke down. In attempting to send the very large SOAP envelope (all XML plain text) back to the Apache web server, the connector would crash.

Because of this limitation, the SOAP architecture could not be tested for the Large Database and Large File Access tasks. While the functionality of the application could be tested by accessing the services directly from the application server (and therefore bypassing the connector), doing so would violate the control of the experiment. Additionally, circumventing the web server in the data flow does not allow for the performance metric gathering as described earlier.

Since this limitation is not truly a quality of the SOAP implementation but rather of the connector, it cannot be counted against the SOAP architecture. The SOAP response is returned as expected when the Web Service is accessed from the application server directly.

That being said, it is a very common set up for a system to use a web server as its common gateway and make use of connector technology to access applications that must run under an application server. Because of the fact that the SOAP implementation requires the return of a fully formed SOAP envelope, it would not be uncommon for such a large piece of data to cause a crash. Therefore, in this respect, we think this issue can be seen as a drawback to the SOAP implementation.

3.2 Initialization Issues

In the process of testing, it was determined that for certain web service architectures, there was a considerable increase in delay for the very first access. Once that access was complete, the performance time stayed relatively constant at a value significantly lower than that of the first access.

Below (Figure 2) is a table showing typical first access skews for the various architectures along with the skew factor from the average access time. In all cases there is at least a doubling of the average access time, and in some cases as high as a factor of 39 increase.

Architecture	Task	Average First Access Time (s)	Average Access Time (s)	Skew Factor
CGI	Simple DB	0.304	0.126	***2.413***
	Long DB	18.629	1.610	***11.571***
	File Access	11.282	0.898	***12.563***
PHP	Simple DB	0.112	0.005	***22.400***
	Long DB	17.081	0.551	***31.000***
	File Access	11.588	0.220	***52.673***
Java Servlet	Simple DB	0.530	0.025	***21.200***
	Long DB	19.180	0.485	***39.546***
	File Access	10.780	3.823	***2.820***
SOAP	Simple DB	1.051	0.472	***2.227***
	Long DB			
	File Access			

Figure 2: Initialization latency chart

It is interesting to note that the overall skews seem to be higher for both the PHP and Java servlet architectures than for that of CGI. This can most likely be attributed to the fact that both PHP and Java servlet require the use of an installed shared module in the Apache web server, whereas the CGI implementation does not. It would make sense that the native Apache web server code would run faster than that of installed modules.

Additionally, these results are effected by the order of magnitude of a given application's response time for a given task. In the case of the architecture-task combinations that had a larger average access time, in general, the skew factor was less dramatic. This is to be expected as the above mentioned running of shared module code most likely contributes a constant time latency, which would seem much larger in comparison to an application whose average time is quite small. The one exception to this is the large database access via Java servlet. In the majority of the time trials, this access demonstrated response times with increases by a factor of 40. This is most likely attributed to the JDBC driver used to connect the servlet with the MySQL database.

Finally, database caching behavior can be cited as well as one of the reasons for this initialization skew. While the caching behavior of the MySQL database is out of the scope of this paper, we can assume that the initial execution of the SQL query will likely be the most expensive one.

3.3 Final Response Time Results

After removing initial accesses from the results, the data below was obtained. In Figures 3-6, average access times are shown for each of the four benchmark tasks, including the Null task.

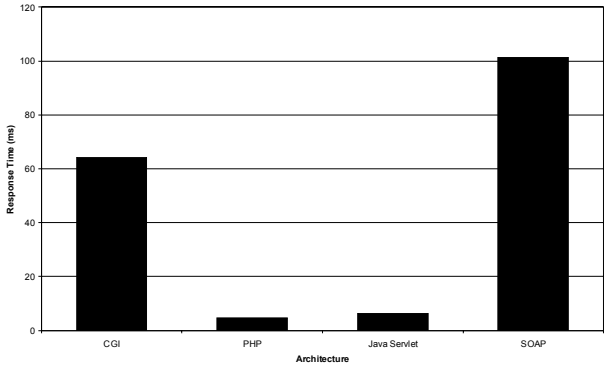

Figure 3: Null (Hello World) task average response times

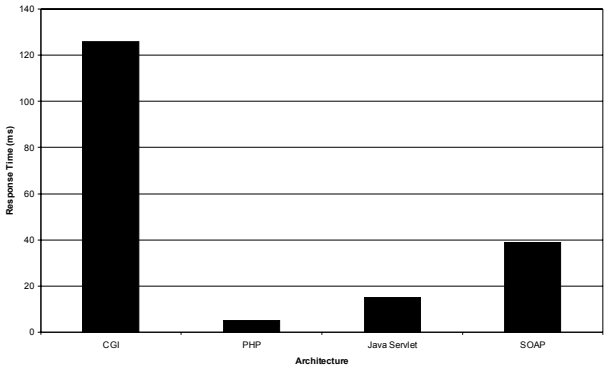

Figure 4: Simple database task average response times

The order of performance, from fastest to slowest, varied across all four tasks. Beginning with the "Hello World" operation, it is quite clear that the SOAP implementation requires the greatest overhead. This was as expected as the generation of

the XML SOAP envelope is an additional step that none of the other architectures require.

As far as overall performance, the PHP implementation finished fastest in three out of the four tasks, while finishing in second in the remaining task. The CGI implementation performed the slowest, finishing last in two out of the four tasks.

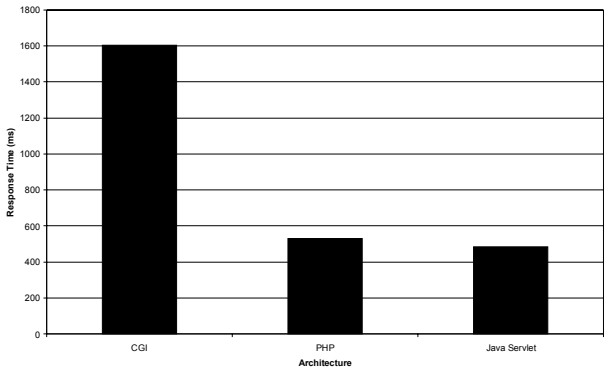

Figure 5: Large database task average response times

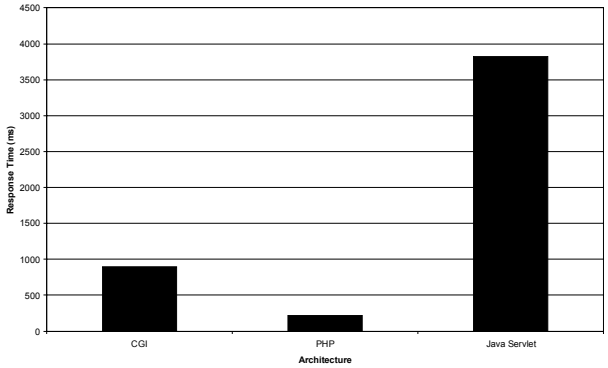

Figure 6: Large file access task average response times

Since the SOAP implementation was not testable in two of the four tasks, it was not considered in the overall performance comparison. However, it can be assumed from the Null operation results as well as the fact that the added SOAP envelope creation is required, that the SOAP implementation would perform the slowest overall if forced to complete the larger tasks. In fact, rough estimates using sent and received times from the SOAP clients show that the large database (11.297 sec. average) and the large file access (9.534 sec. average) response times support that claim.

3.4 Response Time Consistency

Worth mentioning is the fact that for almost all tasks across all architectures, the response times remained relatively close to constant. When comparing standard deviations to the means, the data point spreads were small. For example, the CGI implementation of the large database access task had a mean response time of 1.934 seconds, with a standard deviation of 0.275, a full order of magnitude less (as explained in Section 3.2, these values are calculated excluding initial response times).

For tasks that had much smaller mean run times (on the order of 10 ms and less) the standard deviations appear larger when compared to the mean. For example, the Java servlet implementation of the simple database access task had a mean execution time of 30.4 ms, with a standard deviation of 86 ms. While this value seems high when compared to the mean, one must keep in mind a fact of analyzing response times. That point is that all execution times are of course bounded by zero. This effectively skews the results normally obtained by comparing a standard deviation with a mean, as data points cannot be normally distributed as would be expected. Therefore while under a normal distribution an 86 ms standard deviation would seem high in comparison to the 30.4 ms mean, in this scenario it is actually still quite low.

The one exception to consistent response time results was the Java servlet implementation of the large file access task. While all other implementations of this task remained at a somewhat constant response time, the servlet data yielded a 1.923 second mean, with a standard deviation of 3.396 seconds. Data points ranged from 0.83 seconds to some very lengthy response times over 17 seconds. These high data points were not merely outliers as they appeared consistently in all trials.

We have come to believe the explanation lies in the Java implementation of file I/O. All file input and output is stream based, and in the case of the Java servlet, we implemented a `FileReader` class wrapped in a `BufferedReader` class (J2SE ver. 1.4.2). The workflow through which the file data travels before actually being sent out to the requesting client is significantly complex when compared to the other architectures. By introducing this much additional program logic, the response time would understandably have larger variations. Additionally, the servlet implementation of this task read and sent out file data within the same loop, as opposed to reading in all data into a buffer, then outputting it all at once. Again, this introduces

further complications in terms of possible timing issues between the output stream from the servlet to the web server and the input stream from the servlet to the file.

4. Usability Testing Results

By nature of any usability test, the results can tend to be somewhat subjective. In determining what architecture is the most intuitive, or most straightforward, it is difficult to report impartial data. To produce the most objective results, heuristic user trials and evaluations would be required, however in the case of this paper, only the findings of the authors are taken into account. With that in mind, every attempt was made to make the usability analysis as empirical as possible.

To that end, we have broken down the implementation of the four benchmark tasks into smaller, logical design pieces. Then, for each piece, we discuss the ease of difficulty of implementation. Finally, we end with some architecture specific benefits and limitations.

4.1 Deployment

Examining the Null implementations is a good point at which to evaluate the level of effort in simply deploying a web service under each architecture. Deployment here does not consider test environment setup, as covered in Section 2.1. Rather, this section deals with implementation after the framework is in place.

For a CGI Null operation, little is required. The empty CGI file must be placed in the proper location for the web server to access, and also must be set with the correct permissions. In the case of Perl CGI, no compilation is required (as Perl is interpreted), however an empty CGI executable written in C++ would still need to be compiled prior to deployment.

To deploy a PHP service that, in effect, does nothing, is absolutely trivial. It is no more difficult than writing an HTML static page in plain text that merely has empty <html> and <body> tags.

In the case of the Java servlet, a java class must be created that inherits from the `HttpServlet` class. In the case of the Axis SOAP service, a deployment descriptor XML document must be included along with a Java class that executes the actual service functionality. In both these cases, the level of complexity is far greater than that of CGI or PHP.

The Axis SOAP implementation does however provide an alternate deployment strategy that rivals CGI and PHP for simplicity. By writing a Java class that implements the Web Service functionality (in the Null case, a class with a method that returns a "Hello World!" string), and simply giving it a ".jws" file extension, the Axis SOAP implementation will automatically create the service without requiring any XML descriptors.

4.2 Database Access

For all architectures, additional modifications were required to access the MySQL database. In the case of PHP, this modification was the most trivial, as a compiler option was provided when building the PHP shared module to include MySQL support. In the case of the other architectures, a separate database package was required. This package had to be downloaded, installed and configured.

There are three different API's used to access the MySQL database used in the four architectures. Those include Perl DBI, the PHP API, and the Java JDBC API. The PHP API requires four function calls to achieve the large database access implementation, the Perl DBI uses six, and the JDBC requires seven.

4.3 HTML Output

HTML output of course applies to all architectures except the SOAP implementation, as that does not return HTML. Using Perl CGI, all HTML output is sent to the client using `print` statements. Within the print statements are the actual HTML tags, with dynamic data included as well. This is a similar method used in the Java servlet architecture, using `println` statements from a `Writer` object from the `HttpServletRequest` object. PHP stands alone in that an actual HTML document is created with the dynamic content being provided by embedded PHP statements. In terms of simplicity, the PHP implementation again seems superior.

4.4 SOAP Client Access

A distinguishing factor of a SOAP implementation is the XML SOAP Envelope that requests and responses are sent within. This makes accessing a SOAP web service (and receiving as well) more complex. In the case of the remaining three architectures, they are simply serving HTML over HTTP. As this practice is so common, the ability to receive such responses is embedded into any web browser (via the <form> element) as well as many programming languages (`URLConnection` class in Java).

Because of this, an additional level of complexity is required to access SOAP web services as compared to the others. That being said, SOAP web services were design to do much more than send simply HTML, and from that standpoint the comparison is somewhat unfair. However, from a pure usability standpoint in terms of deploying general web applications, the requirement of a special client to send and receive messages within a SOAP envelope makes the SOAP architecture more complex to use and implement.

5. Conclusion and Future Work

The intent of this paper was not to discover the "best" architecture to use for a web service. If anything, the results have shown that each have pros and cons, as well as their specific domain in which they excel.

In terms of overall quantitative response times, the results are self-explanatory. As shown from Figures 3-6, overall, the PHP architecture implementations performed the best across the widest range of tasks. In terms of usability, we would argue again that PHP came out ahead of the others. Taking into account the trivial deployment strategy, an easily configurable and usable database API, and an embedded HTML design, it is not surprising at all that PHP has become one of, if not the most, popular method for implementing web applications on the Internet today.

For future work, we would like to of course be able to fully test the SOAP implementation as we were unable to do. Using possible modifications to the connector application, or perhaps a separately developed connector, future tests could be closer to complete. Additionally, we would like to expand the benchmark task suite to include tasks more suited for the SOAP protocol. These tasks would include passing complex data back and forth between clients, to be used by applications in complex ways. It would be interesting to see how the "lower level" architectures could handle passing that type of data, and with what performance results.

6. References

[1] Lance Titchkosky, Martin Arlitt, Carey Williamson. A Performance Comparison of Dynamic Web Technologies, *Proceedings of IEEE MASCOTS 2003*, October 2003

[2] Giulano Casale. Combining Queueing Networks and Web Usage Mining Techniques for Web Performance Analysis, *2005 ACM Symposium on Applied Computing*, 2005

[3] Madhusudhan Govindaraju, Aleksander Slominski, Kenneth Chiu, Pu Liu, Robert van Engelen, Michael J. Lewis. Toward Characterizing the Performance of SOAP Toolkits, *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, Novermber 2004

[4] Lincold Stein, Doug MacEachern, *Writing Apache Modules with Perl and C*, O'Reilly & Associates, 1999

[5] Apache Software Foundation, "Axis User's Guide," http://ws.apache.org

[6] Apache Software Foundation, "The Apache Jakarta Tomcat Connector Documentation Index," http://jakarta.apache.org

[7] Eric Armstrong, Jennifer Ball, Stephanie Bodoff, Debbie Bode Carson, Ian Evans, Dale Green, Kim Haase, Eric Jendrock, "The J2EE 1.4 Tutorial," http://java.sun.com/j2ee

[8] Lei Xu, Baowen Xu, Jixiang Jiang. Testing Web Applications Focusing on Their Specialties, *ACM SIGSOFT Software Engineering Notes*, January 2005

[9] MySQL AB "MySQL Reference Manual," http://www.mysql.com

[10] P. Barford and M. Crovella. Measuring Web Performance in the Wide Area, *ACM Performance Evaluation Review*, September 1999

[11] W3C, HTTP – Hypertext Transfer Protocol, http://www.w3.org/Protocols/

[12] Mehdi Achour, Friedhelm Betz, Antony Dovgal, Nuno Lopes, Philip Olson, Georg Richter, Damien Seguy, Jakub Vrana, PHP Manual, http://www.php.net/manual/en/
[13] W3C, SOAP Version 1.2, http://www.w3.org/TR/soap/
[14] W3C, Web Services Description Language Version 1.1, http://www.w3.org/TR/soap/
[15] W3C, Common Gateway Interface, http://www.w3c.org/CGI

A. Appendix

Below are listings of the web service implementations using the four architectures tested.

A.1 CGI – Simple Database Access (Simple_db.cgi)

```
#!/usr/bin/perl -wT

use DBI;
use CGI;

$cgi = new CGI;
$id = $cgi->param("player_id");

print $cgi->header;

# get data
$db_handle = "database=ws;mysql_socket=/tmp/mysql.sock";
$db = DBI->connect("DBI:mysql:$db_handle", 'mike') or die "Can't connect: " . DBI->errstr;
$st = $db->prepare("select * from Roster where player_id = $id") or die "Can't prepare: " . $db->errstr;
$st->execute() or die "Can't execute: " . st->errstr;

print "<html>\n";
print "<head></head>\n";
print "<body>\n";

# output data
@data = $st->fetchrow_array();
print <<"EOF";
<table cellspacing="4">
<tr>
<th>First Name</th><th>Last Name</th><th>Pos</th><th>Order</th><th>OPS</th>
</tr>
<tr>
<td>$data[2]</td><td>$data[1]</td><td>$data[3]</td><td>$data[4]</td><td>$data[5]</td>
</tr>
</table>
</body></html>
EOF

# disconnect
$db->disconnect;
```

A.2 PHP – Simple Database Access (Simple_db.php)

```
<html>
<head></head>
<body>

<?php
// connecting
$link = mysql_connect('localhost', 'mike') or die('Could not connect: ' . mysql_error());

// selecting db
mysql_select_db('ws') or die('Could not select database');

// retrieve parameter from html form
$id = $_POST['player_id'];

// Performing SQL query
$query = "select * from Roster where player_id = $id";
$data = mysql_query($query) or die('Query failed: ' . mysql_error());

// process results
$data = mysql_fetch_array($data, MYSQL_NUM);
?>

<table cellspacing="4">
<tr>
<th>First Name</th><th>Last Name</th><th>Pos</th><th>Order</th><th>OPS</th>
</tr>
<tr>
<td><?php echo $data[2]?></td>
<td><?php echo $data[1]?></td>
<td><?php echo $data[3]?></td>
<td><?php echo $data[4]?></td>
<td><?php echo $data[5]?></td>
</tr>
</table>

<?php
// cleanup
mysql_close($link);
?>

</body>
</html>
```

A.3 Java servlet – Simple Database Access (Simple_db.java)

```
import javax.servlet.http.*
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;

public class Simple_db extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse resp)
        throws java.io.IOException {

        try {
            // grab data from http request
            String id = req.getParameter("player_id");

            // connect to db
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/ws?user=mike ");

            // set up statement
            Statement stmnt = null;
            ResultSet rs = null;

            // execute
            stmnt = conn.createStatement();
            rs = stmnt.executeQuery("select * from Roster where player_id = " + id);

            // set headers for output
            resp.setContentType("text/html");
            resp.setBufferSize(8192);
            java.io.PrintWriter out = resp.getWriter();

            // output
            out.println("<html>");
            out.println("<head></head>");
            out.println("<body>");

            out.println("<table cellspacing=\"4\">");
            out.println("<tr>");
            out.println("<th>First Name</th><th>Last Name</th><th>Pos</th><th>Order</th><th>OPS</th>");
            out.println("</tr>");
            out.println("<tr>");

            // show results
            rs.next(); // get first row
            out.println("<td>" + rs.getString(3) + "</td>");
            out.println("<td>" + rs.getString(2) + "</td>");
            out.println("<td>" + rs.getString(4) + "</td>");
            out.println("<td>" + rs.getString(5) + "</td>");
            out.println("<td>" + rs.getString(6) + "</td>");

            out.println("</tr>");
            out.println("</table>");
            out.println("</body>");
            out.println("</html>");

            // cleanup
            out.close();
            stmnt.close();

        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

A.4 Java Axis SOAP – Simple Database Access (Simple_db.jws)

```
import java.sql.Connection;
import java.sql.DriverManager;
import java.sql.SQLException;
import java.sql.ResultSet;
import java.sql.Statement;

public class Simple_db {


    public String[] queryLineup(String id) {

        try {

            // connect to db
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/ws?user=mike");

            // set up statement
            Statement stmnt = null;
            ResultSet rs = null;

            // execute
            stmnt = conn.createStatement();
            rs = stmnt.executeQuery("select * from Roster where player_id = " + id);

            // return results
            rs.next(); // get first row
            String[] result = {rs.getString(3),rs.getString(2),rs.getString(4),rs.getString(5),
                               rs.getString(6)};
            // cleanup
            stmnt.close();

            return result;

        } catch (Exception e) {
            System.err.println(e);
            return null;
        }
    }
}
```

A.5 CGI – Large Database Access (long_db.cgi)

```
#!/usr/bin/perl -wT

use DBI;
use CGI;

$cgi = new CGI;
$letter = $cgi->param("letter");

print $cgi->header;

# get data
$db_handle = "database=ws;mysql_socket=/tmp/mysql.sock";
$db = DBI->connect("DBI:mysql:$db_handle", 'mike') or die "Can't connect: " . DBI->errstr;
$st = $db->prepare("select nameFirst, nameLast, birthDay, birthYear, birthMonth, weight, height, bats,
    throws from Master where nameLast like '$letter%' order by nameLast")
    or die "Can't prepare: " . $db->errstr;
$st->execute() or die "Can't execute: " . st->errstr;

print "<html>\n";
print "<head></head>\n";
print "<body>\n";

# output data
print <<"STOP";
<table cellspacing="4">
<tr>
<th align="left">First Name</th><th align="left">Last Name</th><thalign="left">D.O.B.</th>
<th>Weight</th><th>Height</th><th>Bats</th><th>Throws</th>
</tr>
STOP

while (@data = $st->fetchrow_array()) {
    print "<tr>";
    print "<td align=\"left\">$data[0]</td><td align=\"left\">$data[1]</td>";
    print "<td>$data[4]/$data[2]/$data[3]</td>";
    print "<td>$data[5]</td><td>$data[6]</td>";
    print "<td>$data[7]</td><td>$data[8]</td>";
    print "</tr>";
}

print "</table>";
print "</body></html>";

# disconnect
$db->disconnect;
```

A.6 PHP – Large Database Access (long_db.php)

```
<html>
<head></head>
<body>

<?php
// connecting
$link = mysql_connect('localhost', 'mike') or die('Could not connect: ' . mysql_error());

// selecting db
mysql_select_db('ws') or die('Could not select database');

// retrieve parameter from html form
$letter = $_POST['letter'];

// Performing SQL query
$query = "select nameFirst, nameLast, birthDay, birthYear, birthMonth, weight, height, bats, throws from
Master where nameLast like '$letter%' order by nameLast";
$data = mysql_query($query) or die('Query failed: ' . mysql_error());
?>

<table cellspacing="4">
<tr>
<th align="left">First Name</th>
<th align="left">Last Name</th>
<th align="left">D.O.B.</th>
<th>Weight</th>
<th>Height</th>
<th>Bats</th>
<th>Throws</th>
</tr>
<?php while ($result = mysql_fetch_array($data, MYSQL_NUM)) : ?>
  <tr>
  <td><?php echo $result[0];?></td>
  <td><?php echo $result[1];?></td>
  <td><?php echo "$result[4]\\$result[2]\\$result[3]";?></td>
  <td><?php echo $result[5];?></td>
  <td><?php echo $result[6];?></td>
  <td><?php echo $result[7];?></td>
  <td><?php echo $result[8];?></td>
  </tr>
<?php endwhile; ?>
</table>

<?php
// cleanup
mysql_close($link);
?>

</body>
</html>
```

A.7 Java servlet – Long Database Access (Long_db.java)

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

public class Long_db extends HttpServlet {

    public void doPost(HttpServletRequest req, HttpServletResponse resp) throws java.io.IOException {

        try {

            // grab data from http request
            String letter = req.getParameter("letter");

            // connect to db
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/ws?user=mike");

            // set up statement
            Statement stmnt = null;
            ResultSet rs = null;

            // execute
            stmnt = conn.createStatement();
            rs = stmnt.executeQuery("select nameFirst, nameLast, birthDay, birthYear, birthMonth, weight,
height, bats, throws from Master where nameLast like '" + letter + "%' order by nameLast");

            // set headers for output
            resp.setContentType("text/html");
            resp.setBufferSize(8192);
            java.io.PrintWriter out = resp.getWriter();

            // output
            out.println("<html>");out.println("<head></head>");out.println("<body>");

            out.println("<table cellspacing=\"4\">");
            out.println("<tr>");
            out.println("<th align=\"left\">First Name</th>");
            out.println("<th align=\"left\">Last Name</th>");
            out.println("<th align=\"left\">D.O.B.</th>");
            out.println("<th>Weight</th>");out.println("<th>Height</th>");
            out.println("<th>Bats</th>");out.println("<th>Throws</th>");
            out.println("</tr>");

            while (rs.next()) {
                out.println("<tr>");
                out.println("<td>" + rs.getString(1) + "</td>");
                out.println("<td>" + rs.getString(2) + "</td>");
                out.println("<td>" + rs.getString(5) + "/" +
                            rs.getString(3) + "/" +
                            rs.getString(4) + "</td>");
                out.println("<td>" + rs.getString(6) + "</td>");
                out.println("<td>" + rs.getString(7) + "</td>");
                out.println("<td>" + rs.getString(8) + "</td>");
                out.println("<td>" + rs.getString(9) + "</td>");
                out.println("</tr>");
            }
            out.println("</table>");
            out.println("</body>");
            out.println("</html>");

            // cleanup
            out.close();stmnt.close();

        } catch (Exception e) {System.err.println(e);}
    }
}
```

A.8 Java Axis SOAP – Long Database Access (Long_db.java)

```
import java.sql.*
import org.w3c.dom.*;
import javax.xml.parsers.DocumentBuilder;
import org.apache.axis.utils.XMLUtils;

public class Long_db {

    public Element[] queryLineup(Element[] input) {

        try {

            String letter;

            // for returning record ids
            int count=0;

            // connect to db
            Class.forName("com.mysql.jdbc.Driver").newInstance();
            Connection conn = DriverManager.getConnection("jdbc:mysql://localhost/ws?user=mike");

            // set up statement
            Statement stmnt = null;
            ResultSet rs = null;

            // extract letter from input xml element
            Node temp = input[0].getFirstChild();
            if (temp.getNodeType() == Node.TEXT_NODE)
                letter = temp.getNodeValue();
            else
                letter = "_";

            // execute
            stmnt = conn.createStatement();
            rs = stmnt.executeQuery("select nameFirst, nameLast, birthDay, birthYear, birthMonth, weight,
height, bats, throws from Master where nameLast like '" + letter + "%' order by nameLast");

            // prepare xml document
            DocumentBuilder docBld = XMLUtils.getDocumentBuilder();
            Document doc = docBld.newDocument();

            // create root node
            Element results = doc.createElement("results");

            // return results
            while (rs.next()) {

                // create record node
                Element rec = doc.createElement("record");
                rec.setAttribute("id",String.valueOf(count++));

                // create data nodes - first name
                Element fname = doc.createElement("first_name");
                fname.appendChild(doc.createTextNode(rs.getString(1)));

                // last name
                Element lname = doc.createElement("last_name");
                lname.appendChild(doc.createTextNode(rs.getString(2)));

                // birth day
                Element bday = doc.createElement("birth_day");
                bday.appendChild(doc.createTextNode(rs.getString(3) + "/" +
                                                    rs.getString(5) + "/" +
                                                    rs.getString(4)));

                // weight
                Element weight = doc.createElement("weight");
                weight.appendChild(doc.createTextNode(rs.getString(6)));
```

A.8 [cont.] Java Axis SOAP – Long Database Access (Long_db.java)

```
                // height
                Element height = doc.createElement("height");
                height.appendChild(doc.createTextNode(rs.getString(7)));

                // bats
                Element bats = doc.createElement("bats");
                bats.appendChild(doc.createTextNode(rs.getString(8)));

                Element thrws = doc.createElement("throws");
                thrws.appendChild(doc.createTextNode(rs.getString(9)));

                // add data to record node
                rec.appendChild(fname);
                rec.appendChild(lname);
                rec.appendChild(bday);
                rec.appendChild(weight);
                rec.appendChild(height);
                rec.appendChild(bats);
                rec.appendChild(thrws);

                // add to root node
                results.appendChild(rec);

            }

            // cleanup
            stmnt.close();

            Element[] ret = { results };
            return ret;

        } catch (Exception e) {
            System.err.println(e);
            return null;
        }
    }
}
```

A.9 CGI – File Data (file_data.cgi)

```
#!/usr/bin/perl -wT

use CGI;

$cgi = new CGI;
$id = $cgi->param("file_id");

print $cgi->header;

# choose file
if ($id == 1) {
    $file = "text1";
} elsif ($id == 2) {
    $file = "text2";
} else {
    $file = "text3";
}

# open file
open(INPUT, "/home/mike/ws/file_data/data/" . $file)
    or die "Can't open file: " . $file;

# read data
@data = <INPUT>;

# close file
close(INPUT);

print "<html>\n";
print "<head></head>\n";
print "<body>\n";

# output data
print @data;

print "</body></html>";
```

A.10 PHP – File Data (file_data.php)

```
<html>
<head></head>
<body>

<?php // start php

// retrieve parameter from html form
$id = $_POST['file_id'];

// select file name
switch ($id) {
 case 1:
   $file = "/home/mike/ws/file_data/data/text1";
   break;
 case 2:
   $file = "/home/mike/ws/file_data/data/text2";
   break;
 case 3:
   $file = "/home/mike/ws/file_data/data/text3";
   break;
}

// open file and read
$fp = fopen($file,'r');
$data = fread($fp, filesize($file));

// output data
echo $data;

// end php
?>

</body>
</html>
```

A.11 Java servlet – File Data (file_data.java)

```
import javax.servlet.http.*;
import java.io.BufferedReader;
import java.io.FileReader;

public class File_data extends HttpServlet {


    public void doPost(HttpServletRequest req, HttpServletResponse resp) throws java.io.IOException {

        try {

            // grab data from http request
            String id = req.getParameter("file_id");
            String file;

            switch (Integer.parseInt(id)) {
            case 1:
                file = "/home/mike/ws/file_data/data/text1";
                break;
            case 2:
                file = "/home/mike/ws/file_data/data/text2";
                break;
            case 3:
                file = "/home/mike/ws/file_data/data/text3";
                break;
            default:
                file = "";
            }

            // open file reader
            FileReader filereader = new FileReader(file);
            BufferedReader reader = new BufferedReader(filereader);

            // set headers for output
            resp.setContentType("text/html");
            resp.setBufferSize(8192);
            java.io.PrintWriter out = resp.getWriter();

            // output
            out.println("<html>");
            out.println("<head></head>");
            out.println("<body>");

            // send file data
            while (reader.ready())
                out.println(reader.readLine());

            out.println("</body>");
            out.println("</html>");

            // cleanup
            out.close();

        } catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

A.12 Java Axis SOAP – File Data (file_data.jws)

```
import java.io.BufferedReader;
import java.io.FileReader;

public class File_data {

    public String getFileData(int id) throws java.io.IOException {

        String ret = "";
        String file;

        switch(id) {
        case 1:
            file = "/home/mike/ws/file_data/data/text1";
            break;
        case 2:
            file = "/home/mike/ws/file_data/data/text2";
            break;
        case 3:
            file = "/home/mike/ws/file_data/data/text3";
            break;
        default:
            file = "";
        }

        // open file reader
        FileReader filereader = new FileReader(file);
        BufferedReader reader = new BufferedReader(filereader);

        // send file data
        while (reader.ready())
            ret += reader.readLine();

        return ret;

    }
}
```