Servlet Benchmark

by

Youqing Zheng

ID: 113-88-5356

City University of CUNY

2001

Course: I9800, Spring 2000

Abstract

The Servlet APIs wins widespread industry adoption and is being accepted as industry standards to build server-side programming interface. The benchmark suites is important to push forward this tendency by giving the various vendors a tools to measure as well as differentiate their products. This paper aims to provide a proprietary approach to facilitate software or measurement companies produce their suites which are then adopted as de facto industry standards.

There are six sections. After first section's introduction. In the second section, I discuss the performance measures and then give benchmark methodology used in the following. In the third section, a PDL(program design language) high level construction is developed. I am using this PDL as basis of my Java code implementation. The next section deals with the results analysis. We introduce a counterpart, a CGI with same function, to compare the results. In the last section, I conclude my benchmark routine and result.

TABLE OF CONTENTS

GLOSSARY

Servlet. A Servlet is a web component, managed by a container, that generates dynamic content. Servlets are small, platform independent Java classes compiled to an architecture neutral bytecode that can be loaded dynamically into and run by a web server. Servlets interact with web clients via a request response paradigm implemented by the Servlet container. This request-response model is based on the behavior of the Hypertext Transfer Protocol (HTTP). (Sun, 1999)

PDL. PDL is designed for the production of structured designs in a top-down manner. It is a *pidgin* language in that it uses the vocabulary of one language (i.e. English) and the overall syntax of another (i.e., a structured programming language). In a sense, it can be thought of as *structured English*. The current version of PDL is PDL81.

CGI. Abbreviation for common gateway interface. A means for allowing programs or scripts (usually written in C++ or Perl) to add functionality to the World Wide Web.

I. Purpose and Backgroud

Traditionally, Common Gateway Interface is a widely used interface to handle server side functionality. The performance of CGI is stable and it has been a reasonable choice to develop web projects. The main drawback of the CGI technique is overall inefficiency in handling concurrent client requests (Williamson, 1999). This downside of CGI limits its usage in many web applications. Today, with the surge of e-commerce, a number of alternative solutions are available. Java Servlets, Perl and PHP are considered to be the most popular alternative solutions to CGI. Among all these solutions, the Java Servlet approach has become the most popular one for e-commerce applications. Servlets have rich features that provide strong influence on tuning the performance of web applications. Context sharing and multithreading are the important features of the Java Servlet solution.

Third-party servlet containers are available for Apache Web Server, iPlanet Web Server, Microsoft IIS, and others. Servlet containers can also be integrated with web-enabled application servers, such as BEA WebLogic Application Server, IBM WebSphere, iPlanet Application Server, and others.

Whenever a protocol or API wins widespread adoption, efforts to produce benchmarking or performance suites is not far behind. This serves as a tool for the various vendors to measure and differentiate their products. This project would develop a vendor-free[1] approach to benchmark Servlet.

II. Basics of Performance and Benchmark

2.1 Performance

2.1.1 Performance Definition

To analyze performance, we need to define its meaning in the client/server context. Generally speaking, there are two main category of performance approaches, one common to many other execution paradigms, and another more especially defined for client/server models.

- ***Absolute performance***

  Absolute performance measures the times taken by an application to execute. This definition is well suited for programs with no asynchronous delays. And it's used as an

[1] Vendor-free refers to Servlet container free. The approach here is indeed requests a common web server vendor in order to provide a benchmark base for the Servlet container. We use Apache as web server, because most Servlet container vendor provide plugin for it, such Tomcat, Rsin, Jserv and WebLogic.

index of speed for such contexts. For the server side benchmark, we are dealing with network congestion and machine load etc., all the asynchronous stuff. So this idea of performance may not fit into our benchmark.

- ***Perceived performance***

Perceived performance measures the times taken by a server to execute and terminate the client requested action. This definition is well suited for client/server paradigm where the serve performance perceived from the client matches this measure. From now on, the performance in this paper will all refer to this concept.

2.1.2 Performance Measures

- *Latency*

Latency refers to the amount of time the web server takes to process a request. This is an important measure, because we don't want user to wait too long for a request to be processed. It is important to keep the average latencies as low as possible. Latency typically measured in milliseconds.

- *Throughput*

The quantity of data the web server running a particular application is able to push through per unit of time. This is important, especially for dynamic content applications that are generating rich content (including multimedia content). Throughput typically measured in kilobytes per second.

- *Connection Rate*

The number of new connections the web server is capable of accepting per unit of time. This measure gives an idea about how well the web site will scale to large numbers of client accesses. The maximum feasible connection rate is usually measured by increasing the load until average latencies rise considerable and/or CPU utilization reaches 100%. Typically measured in connections per second.

- *Request Rate*

  Number of HTTP requests the web server is capable of processing per unit of time, given any reasonable mix of client behavior (HTTP1.0, HTTP1.1, pipelined requests, keep-alive requests). If each HTTP request is served on separate connection. This measurement is same as connection rate.

- *Stability*

  The variance of latency the web server takes to process a request. This measure gives the quality of web service during a period of time. The bigger the variance the lower quality the web service, even with low latency. Typically measured in milliseconds.

2.2 Methodology

In order to make our test having the maximum comparability, we made the test consistent with WebBench^{TM} standard test.

2.2.2 Test Scenarios

- *100% Static Test*

  This test is aim to get the upper bond of web server performance. All requests get back a static html pages, which contains 6250 bytes.

- *100% CGI Test*

  We create this test to measure the performance of CGI programs written in C. All the requests are made to the same C CGI program, which returns 6250 bytes of HTML.

- *100% Servlet Test*

  We create this test to measure the performance of Java Servlet. All of the requests are made to the same Java Servlet, which simply returns 6250 bytes of HTML.

2.2.3 SAP test

The package I developed tests the Server-side Application Performance, or simply SAP. I introduce the SAP tests:

- To provide an application –independent way to compare the dynamic performance of web service using different SAs (Server-side Application) and

- To determine the overhead associated with using a particular SA. SAP test s use applications that do the minimum processing needed to return the same average number of bytes as a static test will return. This means that SAP tests can be used to make fair comparison between static-only and dynamic test scenarios because the response rate for SAP programs are not artificially inflated because they return less data than static requests.

The SAP test essentially measures the minimum overhead for using a particular SA. A common alternative to SAP is to simulate a real world application. One obvious problem with this alternative is that the simulated application will almost certainly behave differently than the one you want to deploy. While that is also true with SAP programs, they have an advantage over simulated applications because they are simpler to implement, smaller and use fewer resources. This simplicity and size advantage lets you use SAP programs to evaluate the true overhead of using a SA as well as making it much easier to develop and run a test.

2.2.4 SAP Metrics

- *SA Request rate*

  The SA request rate is the number of requests/second the web server delivers. It is useful for comparing SA performance under load. It incorporates the performance of the server hardware, the server operating system, the web server software, and the SA. It is based on aggregating the performance of all the requests from all of the client systems used in a test. Comparing SAP based on SA request rate results in a server-centric comparison.

Benchmark SAP solely based on the request rate could be misleading. The SA request rate will always be the upper bound of the application performance. In order to improve the validity of evaluation, we could introduce the SA latency.

- *SA Latency*

SA latency helps the evaluation of SA performance form a user's perspective. It demonstrate the user waiting time with a server with current load.

By looking at the SA latency on each client system, you can see if a web server is handling client requests unequally and if the clients will experience an unacceptable long wait for a response. If either of these undesirable conditions is true, the useful peak performance of the web service is lower than the peak performance you are measuring. We can determine the useful peak performance by finding the maximum performance point at with the web server treats client requests equally and at which it has an acceptable SAP latency.

- *SA stability*

As I mentioned in SA latency, the stability of a SA is also a very important measure to SAP. It evaluate the quality of web server under certain latency. It is critical for a web service in that the unstable SA would serve the client unequally and force more "refresh" button pushing. We measure SA stability using standard deviation of latency.

- *SA Efficiency*

We define SA efficiency as the ratio of a SA metric to the same static test metric. That is:

SA request efficiency = SA request rate / static request rate

SA latency efficiency = SA latency / static latency

SA stability efficiency = SA stability / static stability

SA efficiency is a measure of how much dynamic request performance using a particular SA will degrade from that of static-only requests.

III. PDL Design

PDL (program design language) was originally developed by the company Caine, Farber & Gordon and has been modified substantially since they published their initial paper on it in 1975. Because PDL resembles English, it's natural to assume that any English-like description that collects your thoughts will have roughly the same effect as any other.

I will use PDL to design the test package for server and client side. Some of the main advantage to do this is:

- Language independent. In order to benchmark Servlet, we have to set up the counterparts. They might be written in C, C++ , Perl and PHP etc. But no matter which language you use, the SA must following the same workflow to them comparable. PDL is one of the best solution to fulfill this task. With PDL, we can make sure all the specific language implementation would be under one schema.

- PDL makes reviews easier. For a benchmark package intends to be used by third party, it is important to make low-level design reviews easier and reduce the need to review the code itself. With PDL, we can do review detailed designs without examining source code.

- PDL-to-Code approach supports the idea of iterative refinement. We can start with a high-level architecture, refine the architecture to PDL, and then refine the PDL to source code. This successive refinement in small steps allows you to check you design as you drive it to lower levels of detail. The result is that you can catch high-level errors at the highest level, mid-level errors at the middle level, and low-level errors at the lowest level -- before any of them becomes a problem or contaminates work at more detailed level.

3.1 Server Side Interface Design

The server side design for our “Hello World” like SA is quiet simple. Her e is the PDL design:

```
This is SA routine outputs a static HTML page to client.

Get the request from client

Set the response header to “text/html”

Send the static html to client
```

3.2 Test Routine Design

The client side design is something complicated than server side, because we want to build a client simulate the web browser and meanwhile gather all data we need later to analysis the performance. More important this client should be multi-threaded.

This is SAP test client make the simultaneous requests to SA and output SAP metrics.

```
Get the configuration (max clients, test duration and SA url)
Do for each number (x) less than max clients (for example 1, 2,….maxClients)
    Initial the statistic variable
    Build x connections to SA
    If any connection failed
        record the fails
        close this client
    else
        do for the test duration
            make request to SA and record as well accumulate single request throughput, latency and request rate
        enddo
    endif
    Calculate throughput, latency, request rate and latency standard deviation for this group.
EndDo
```

IV. Implementation

The implementation of SA is straight forward, please look at List 1. For test client, I'm using Java to implement it, please reference to List 2.

4.1 Threads Scheduling

I implement each client as a thread and each group as a thread group. For example, if we want to build 10 clients to call the SA simultaneously, we would have one thread group call "10" and 10 threads each named as "10-i" (i=1…10).

By this approach, we can let the main thread just monitors the ThreadGroup to determine the status of each client , contrasting to loop over each thread. The code piece for main thread is :

```
ThreadGroup group = new ThreadGroup(""+client); // clients =1,2,3….maxclient
Thread[] threads = new Thread[client];
for(int i=0;i<client;++i){
   threads[i] = new BenchThread(group, "client_"+client+"_"+i, url_req); //sponsor i clients
}
…
…
for(int i=0;i<client;++i){
   threads[i].start();//run clients simultaneous
}
Object obj = new Object();
synchronized(obj){
   log("waiting for "+(1000*benchTime)+" millis");
```

```
    obj.wait(1000*benchTime); //wait for clients finished
}
while(true){ //check active threads in the ThreadGroup, is no, finish this test run.
    if(group.activeCount()==0) break;
    synchronized(obj){
    log("waiting for another 1000 millis");
    obj.wait(1000); //wait a little bit
}
```

The BenchThread is the implementation of test client.

4.2 BenchThread Implementation

The main function of BenchThread (inner class of the benchmark client) is to simulate HTTP browser to get request back and record the SAP data to later use. In this benchmark, I only use the “GET” method in HTTP request. If using the URLConnection to simulate the client, there would be lots of overhead due to OO features of Java. In order to make this client as thin as possible, I choice to use plain socket make the HTTP request. There are two classes to encapsulate the HTTP GET call. One is benchmark.util.GetData (List 3), which defines the header the HTTP call. Another one is benchmark..util.HttpGet (List 4), which opens a socket, sends out a request using an instance of GetData and then get the response back. I choice to let it return an java.io.InputStream to the caller to give the caller more control on the response. The relevant code is following:

```
public InputStream getToHost(GetData data) throws Exception{
    Socket socket=null;
    OutputStream outStream=null;
    InputStream inStream=null;
    StringBuffer result=new StringBuffer();
    socket= new Socket(data.host, Integer.parseInt(data.port));
    outStream = new BufferedOutputStream(socket.getOutputStream(), bufferLength);
    outStream.write((data.method+" "+data.path+" "+data.protocol+"\n").getBytes());
    if(data.cookie!=null) outStream.write(("Cookie: "+data.cookie+"\n").getBytes());
    if(data.referer !=null) outStream.write(("Referer: "+URLEncoder.encode(data.referer)+"\n").getBytes());
    if(data.userAgent !=null) outStream.write(("User-Agent: "+data.userAgent+"\n").getBytes());
    outStream.write(("Connection: close\n\n").getBytes());
    outStream.flush();
    return socket.getInputStream();
}
```

4.3 Statistic logic Implementation

I use incremental algorithm to gather the statistic data. For each client (each thread), it is plain and easy to implement. But for the client group (thread group), we have to synchronize the group scope data with the client group data. Let's first look at the client scope.

4.3.1 Client data

For each client, I gather 6 category data, ie. Latency, standard deviation of latency, speed, throughput, succeed request, failed request. For each category, please refer to section 1.1.2, if have any question about the definition.

All category data is easy to be accumulated from each request except for the standard deviation of latency. The formula for it is :

$\sigma = (\sum(X_i\text{-mean}(X))^2/n)^{1/2-}$

σ: standard deviation

X_i: latency of each request

Which means we have to get mean of X first in order to get σ and also means that we have to each request info till the client finishes the request and then calculate the σ.

Thanks for the laziness of human, we have another equivalent formula to derive σ from different stand of point.

$\sigma = \sum X_i^2 - \text{mean}(X)^2/n$

So, with this formula, we can first cumulate X^2 then calculate σ at the last step of the routine. Please refer to List 2 for the implementation.

4.3.2 Group data

Using thread will leads to synchronization work in most circumstance. The same situation is in my package. In the group scope, the client might update the latency or throughput or another data simultaneously. In order to solve this problem, I encapsulate each update in a corresponding function, for example, updateLatency for latency, and make each function as synchronized. Together with the threads scheduling, this package could perfectly work as we design.

4.4 Configuration

The benchmark client is configured by XML. A sample configuration likes the following:

```
<?xml version='1.0' ?>
<benchMark>
<path>/servlet/SimplestServlet</path>
<host>192.168.18.75</host>
<port>80</port>
<timeout>60</timeout>
<client>50</client>
<initClient>1</initClient>
<benchTime>1000</benchTime>
<debug>false</debug>
</benchMark>
```

I write a utility class to parse this XML by SAX. Please look at SAXHelper.java (benchmark.util). The relating code in WebBench class is:

```
  private Map parseXML(String configXML) throws Exception{
    SAXHelper helper = new SAXHelper();
    BufferedReader xmlReader = new BufferedReader(new FileReader(configXML));
    StringBuffer buffer =new StringBuffer();
    String tmp =null;
    while((tmp=xmlReader.readLine())!=null){
      buffer.append(tmp);
    }
    return (Map)helper.getTagContent("*", buffer.toString());
  }
```

4.5 Usage

The usage of this package is simple. First you should deploy the SA into whatever application server or Servlet engine you use, then modify the configuration xml. The last step is start the test client by:

```
Java –classpath <lib path>/xmlparserv2.jar benchmark.WebBench <configuration file>
```

This client could run in JDK 1.2 and above.

V. Result Analysis

In this benchmark, I use Windows 2000 professional as Server OS with PIII 666 MHZ processor and 260 MB memory. The web server is Apache 1.3.17 and Servlet container is Tomcat used as plugin to Apache. The startup size of the memory allocation pool (the garbage collected heap) is set to 16MB and the maximum size of the memory allocation pool is set to 120 MB. The test client runs on NT 4.0 with PIII 666 MHZ processor and 260 MB memeory. The network is 100mbps Ethernet.

5.1 Static Tests

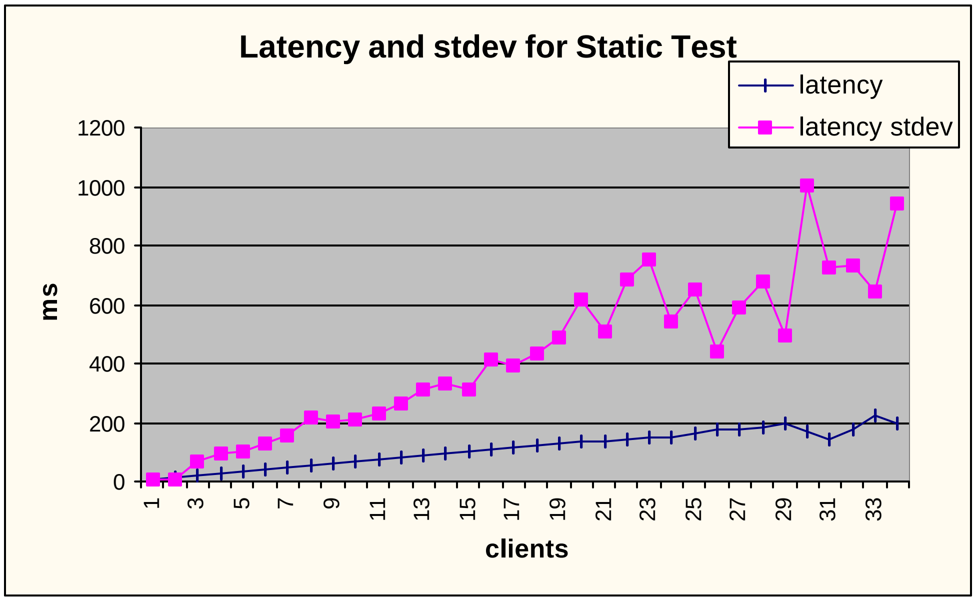

figure 3.1.1

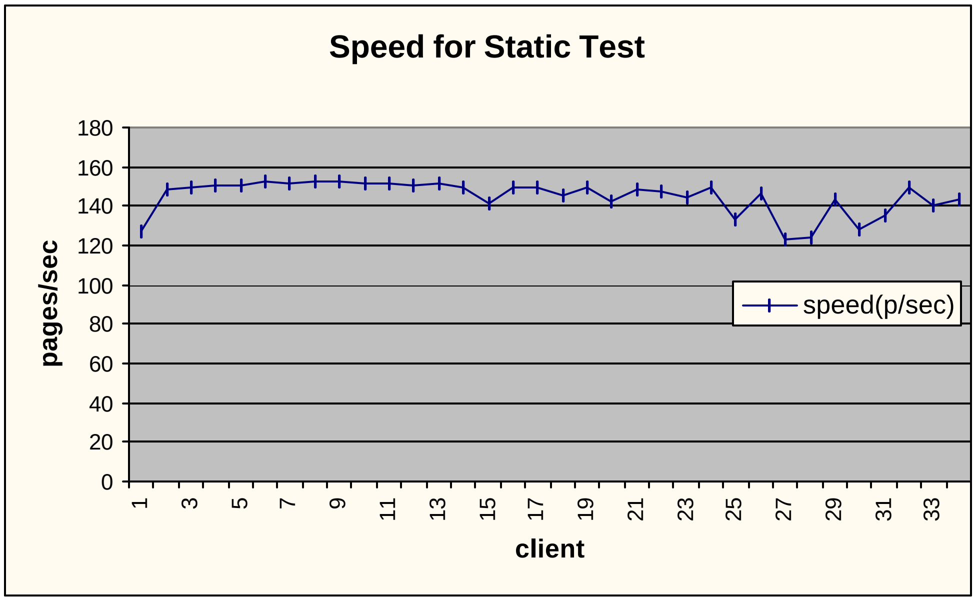

figure 3.1.2

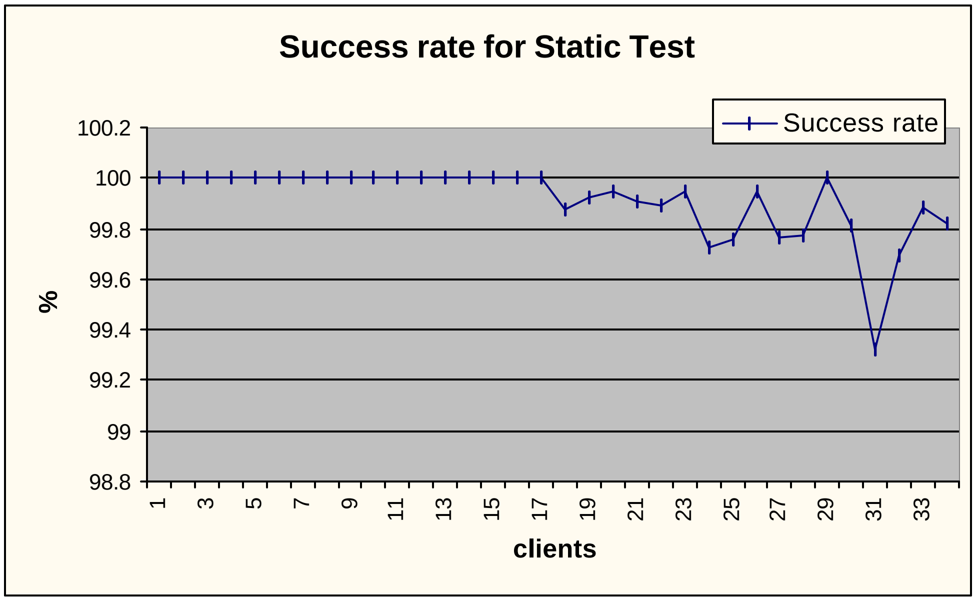

figure 3.1.3

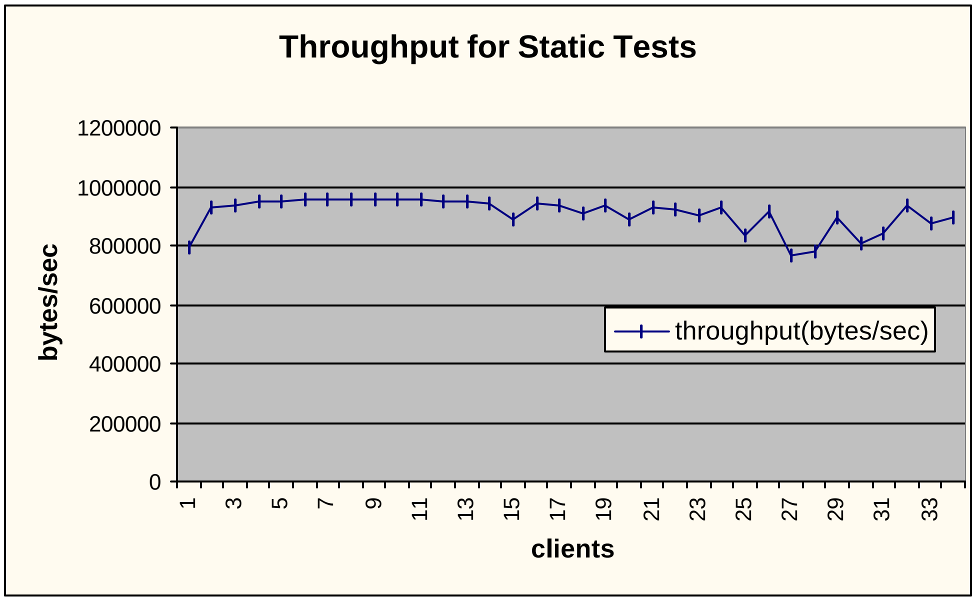

figure 3.1.4

Static test is a test using only 100% pure static html pages. This purpose is to set the web server performance factor in the SAP tests. As in the figure 3.1.1, with the clients increasing, the latency and stdev has the upside tendency. This tendency comes from several source. It might be hardware bottleneck, ie. Memory, io etc. It also might be the web server built-in, for example, process and thread management etc. The same reason could be applied to figure 3.1.2 and figure 3.1.3. No matter which reason, we should deduct this factor from our SAP analysis. I think this is the only way to build the most server –hardware irrelevant benchmark.

5.2 Servlet and C CGI Tests

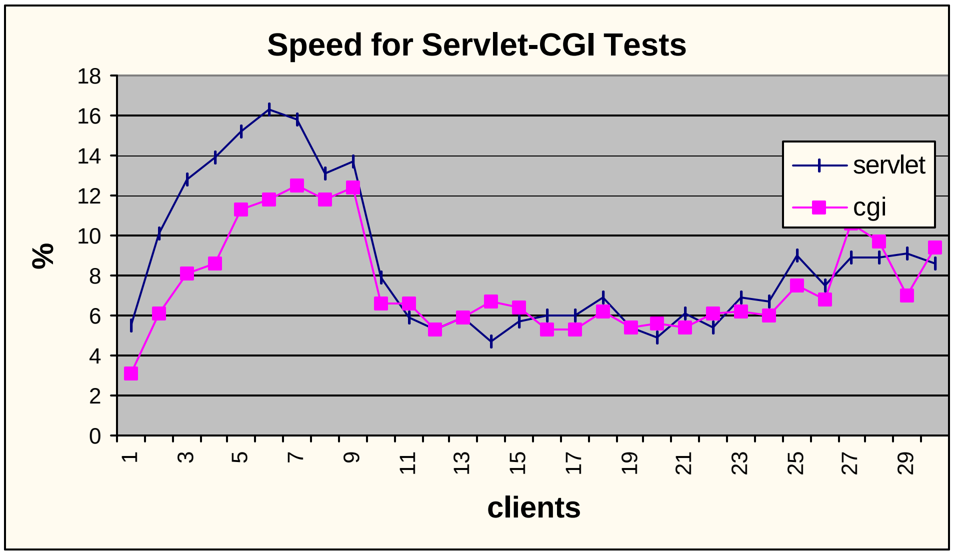

figure 3.2.1

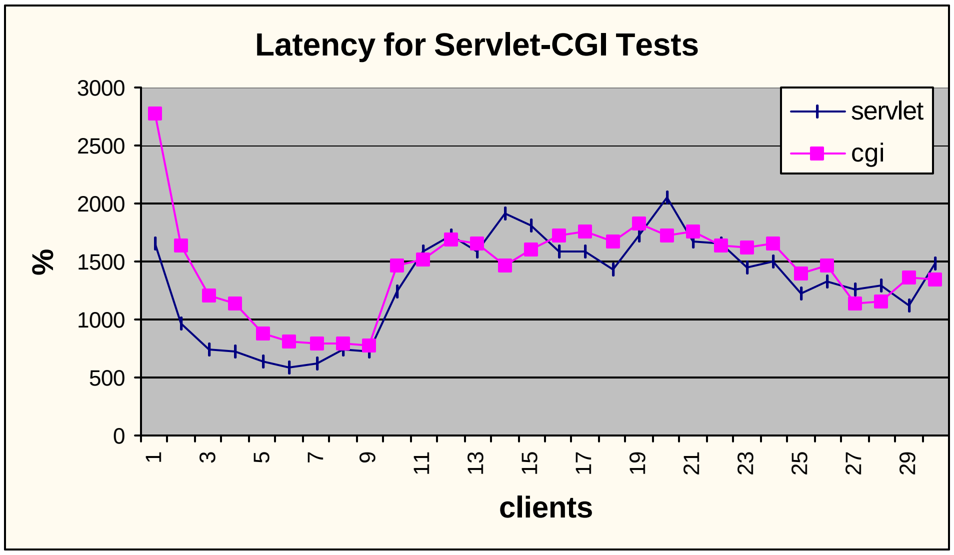

figure 2.2.1

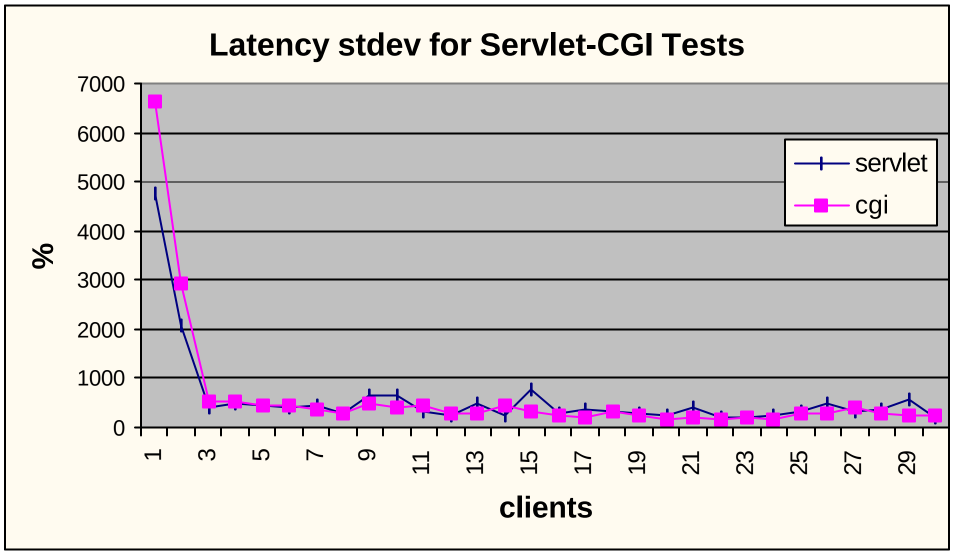

figure 3.2.2

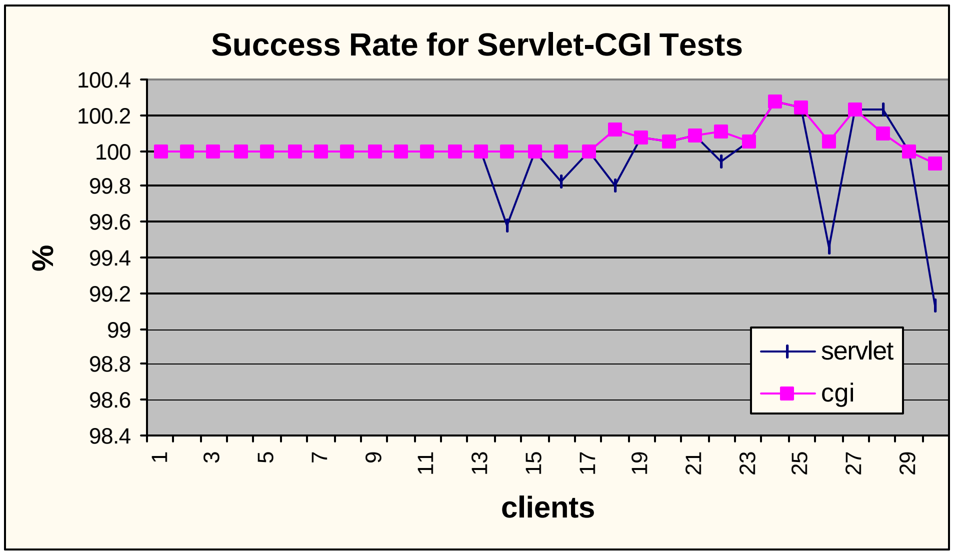

figure 3.2.4

observation and conclusion

1. From latency and speed, both CGI and Servlet SA get better performance when handling small number of concurrent users(less than 10) . We can divide the SA into two parts. One part is in light-loaded environment and another in heavy-loaded one. The life cycle of CGI and Servlet in these two environment keep unchanged.

   For CGI, the life cycle is an obvious cause of its performance problem. A CGI program needs to create a separate process for each user request, and this process will be terminated as soon as the data transfer is completed. We can call CGI as stateless SA. Spawning a separate program instance for each client request takes extra time. The OS has to load the program, allocate memory for the program, and then de-allocate and unload the program from memory (Williamson, 1999). While the OS is performing housekeeping, nothing else can run. This heavy weight context switch prevents CGI programs from improving their efficiency on handling concurrent client requests, thus they are not suitable for high traffic applications that need to process a large number of client requests.

For Servlet, when it is called the very first time, it is loaded into the memory. After the request is processed, the servlet remains in memory and will not be unloaded from memory until the web server is shut down (Williamson, 1999). Except for the SingleTon Servlet, if there are concurrent requests coming, the Servlet engine will sponsor more Servlet thread if no more free. Thread itself is lightweight process comparing with process. At this point, Servlet could get the performance improvement to CGI. But this is not mean Servlet could always performance better than CGI. In the **heavy-loaded** environment, Java VM would be busy on GC and at the extreme point crash the servlet engine.

So, the conclusion based on this observation is: **Servlet performance better than CGI only in the light-loaded environment. In the heavy-loaded scenario, the Java virtual machine spends more resource in the GC which pull down the Servlet SAP to the same level of CGI's.**

2. Servlet is more unstable than CGI in the heavy loaded environment. In most case, The Servlet standard deviation of latency is higher than CGI's. This is self-explain by JVM. For CGI, just because it is stateless, given the system load, CGI processes are running in a mostly identical environment. But it is not same case for Servlet. The GC is decided by JVM. Servlet can not control the timing. When JVM is doing the GC, it will make significant use of CPU. The requests at this period would take longer time to fulfill. At the extreme case, some requests might get failure for timeout. From figure 3.2.4, Servlet success rate (relative to static rate) is lower than CGI in most case. So, with the observation 1, we get the second conclusion: **the**

Servlet SAP is mainly system resource constrained. The higher performance costs more system investment.

3. Comparing with static test, both Servlet and CGI efficiency is fairly low. The maximum speed efficiency for Servlet is 16%, for CGI is 12%. The latency degrades 5.8 times for Servlet at least and 7.7 times for CGI at least. The conclusion is: **SA is not suitable for static HTML service.**

VI. Conclusion

In this paper, I developed a test suite to benchmark the Servlet with CGI as counterpart. The result is interesting. Which suggests that Servlet could be a better solution only in the condition that system is light-loaded. Of course, the load level of system depends on many factors, given the OS, hardware and web server attributes, JVM is the bottleneck to the Servlet SAP. With the JVM technology improvement, the Servlet would gain better performance. Also the Servlet should be strictly used in dynamic HTML as possible. For static HTML, web server has the specialty.

VII. Reference

Caine, S. H. and E. K. Gordon. *PDL—a tool for software design*. AFIPS Press, 271-76, 1975

McConnell, Steve. *Code Complete: a Practical handbook of software construction*. Microsoft Press, 1993.

Sun MicroSystem. Java™ Servlet Specification, v2.2 . 1999.

Williamson, A.R. *Java Servlet by Example*, 1999