CHAPTER 6

Performance & Scalability Verification - Web Application

6.1 Introduction

A large number of businesses have critical applications running on technologies that are obsolescent with poor prospects for support, interoperability and programmer availability. The applications eventually require to be reengineered. Reengineering these applications needs crucial decisions from the performance and scalability perspective. Deploying applications that rely on Web servers, Intranets, and client/server technologies is a challenge both in assuring that the functionality will be maintained and in guaranteeing that the functionality will be delivered with an acceptable performance. Performance problems can bring all sorts of undesired consequences, including financial and sales loss, decreased productivity, and a bad reputation for a company. It is important to plan ahead of the system deployment and capacity of networked systems. Capacity planning involves being able to predict when the existing applications will fail to meet the required performance levels. Performance prediction may be accomplished through performance models [39,51]. Common challenges associated with performance analysis of applications built on a new technology are:

- Vendors are not experienced with the use of the technology for applications with complex characteristics (large number of components, users and data). Hence, they are unaware of performance in such situations.
• References are limited because only a small number of businesses have adapted the new platform. This limits peer experience that can be used to analyze performance

• Unknown factors that would impact the results and decisions make correctness of the performance verification critical.

Analyzing and verifying performance and scalability of a new technology has been a challenge for most businesses that adapt it during the initial phase of its evolution. With minimal references from the vendors, the task of performance verification becomes critical. Performance testing is a good approach to verifying the performance of a system in a multi-user scenario.

There are several steps involved in performance testing – setting up the test environment, creation of test scripts using a COTS (Commerically Available Off the-shelf) load testing tool, executing the scripts simulating a multi-user scenario and analyzing the test results. The results generated by the performance testing needs to be authenticated and verified. This chapter proposes and details certain quantitative techniques that can be used in verifying performance and scalability of the web application. The proposed techniques are validated by following them during the performance testing of .NET architecture for a complex application with thousands of users. The chapter also discussed some of the performance optimization techniques. In an attempt to verify the performance of a complex client server inventory application, reengineered using .NET, the chapter shares the issues faced and the approach followed to effectively verify the performance of the application. The verification and analysis techniques discussed in the chapter are applicable for any application built on .NET technology. However, the inventory application uses ASP.NET and SQL Server 2000.
Listed below are the important application characteristics:

- Complex user screens with several tabs, grids containing large data.
- More than six thousand users.
- More than two thousand locations accessing the application.
- A database of approximately 800 GB.

6.2 Performance Verification

Performance testing can be viewed as a "Black Box" testing [125], which focuses on application and system behavior from outside, with no knowledge of the program code that supports the system. The main objectives of the performance testing is:

- Maximum number of concurrent users that can be supported while offering "acceptable performance."
- Maximum number of concurrent users that can be supported prior to causing a system failure.
- Location of bottlenecks within the application architecture.
- Impact of a software or hardware change on the overall performance of the application.
- Scalability issues

The section describes the test environment and details the scenarios that had an impact on the analysis. The performance test set-up of the application is shown in Figure 6.1.
FIGURE 6.1: PERFORMANCE TEST SETUP

The application functionality is implemented using ASP .NET for user interface and SQL Server 2000 as the database. A separate database is used to store the sessions of the users.

6.2.1 Increasing Number Of Simulated Users

Load testing comprises of generating and increasing the number of simulated users to analyze the impact of multiple users on the application performance. Most COTS load testing tools can be configured to increase the number of users during a test run. A load-testing tool creates a specified number of simulated user threads and runs the test script for each user thread several times. .NET server (like other software servers) creates software resources - components, threads and database connections for every new user request and caches these resources. Hence, the initial processing time of the server is considerably higher for the first run of the test script than the subsequent test script runs – referred to as the server reaching a steady state. It is required to increase the number of simulated users only after the server has reached a steady state. A test conducted by increasing the number of users from 10 to 170 users and adding 100 users in a small time interval of five minutes is shown in Figure 6.2.

The response times are inconsistent with high deviations for the same number of simulated users. The results of this test provide little insight on the performance of the system.

67
**FIGURE 6.2: INCREASING SIMULATED USERS RAPIDLY**
The same test run with longer time intervals between increases in simulated users is shown in Figure 6.3. The response times are consistent and report lower values for the same number of users. The results of the tests indicated that a test execution should account for the time taken by the server to stabilize for analyzing performance.

**FIGURE 6.3: INCREASING SIMULATED USERS GRADUALLY**

While testing the performance using a load-testing tool, the time interval between increases in users should ensure that the test script has run multiple times for each simulated user.
6.2.2 Verifying Performance Test

The primary reason to conduct a performance test is to identify the bottlenecks in the system. If the performance test is incorrectly executed, it would lead to incorrect analysis and the cause of bottlenecks may never be identified.

The significance of performance test verification is illustrated with the experience testing the inventory application. The inventory application provides a "search for inventories" operation. The ASP.NET page for this operation consists of several user interface components designed to ensure that the users of the existing client screens have minimal usability issues when they move to the new application. The search operation was performance tested with the number of simulated user increased from 10 to 60.

<table>
<thead>
<tr>
<th>Number Of Users (N)</th>
<th>Throughput Operations/Sec (X)</th>
<th>Request Time = (Response Time + Think Time) Seconds (R + Z)</th>
<th>Little's Law Validation (No Of Users)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>0.46</td>
<td>21.07</td>
<td>9.69</td>
</tr>
<tr>
<td>20</td>
<td>0.5</td>
<td>38.19</td>
<td>10.09</td>
</tr>
<tr>
<td>30</td>
<td>0.41</td>
<td>53.7</td>
<td>22.01</td>
</tr>
<tr>
<td>40</td>
<td>0.35</td>
<td>65.96</td>
<td>23.08</td>
</tr>
<tr>
<td>50</td>
<td>0.305</td>
<td>78.82</td>
<td>24.04</td>
</tr>
<tr>
<td>60</td>
<td>0.31</td>
<td>97.35</td>
<td>30.17</td>
</tr>
</tbody>
</table>

Table 6.1: Search Inventory Results - Little's law Validation

The response times increased and the throughput of the search operation reduced rapidly. The resource utilization (CPU, Disk and Memory, Network) of the servers on which the application was deployed was less than 30%. Hence, the server resources were not the cause for the performance bottleneck. The application logs indicated the total processing time at the .NET server was 10 seconds while the response time reported by the load-testing tool was 97 seconds. After monitoring the application and the deployment configuration, the resource
utilization of the machines running the load-testing tool was monitored. The CPU Utilization of the machines reached 80% - hence the load-testing tool was the bottleneck.

For other applications tested using the same configuration of machines, the load-testing tool was capable of simulating 100 users. However, the tool (using DOM – Document Object Model parsing mechanism for running the test script) was unable to parse the complex ASP simulating only 20 users. The bottleneck was easily identified using Little’s law [40,49] for verification of performance test. Little’s law states that the number of users in the system is the product of throughput of the operation (operations/sec) and time taken for servicing the operation referred to as Request time – (Response time + think time). The validation of Little’s law for the results indicate that the product of throughput and request time fail to match the number of users beyond 20 users. Hence, validating the performance test using Little’s law ensures that the required numbers of users are simulated in the test environment.

Situations where verifying performance test helped identify genuine problems are:

- The load generator was a bottleneck unable of simulating the required number of users due to the complexity of application being tested and the limitation of the hardware configuration on which the load generator is running.
- Excessive failures of user requests owing to page session time outs. The session time out did not get reflected in the load generator report. However, the results failed to comply with Little’s law.

Load-testing tools capabilities can be verified using operational laws. Failing to do so would provide incorrect conclusions on the applications capacity.
6.2.3 Verifying Bandwidth Utilization

At times the system is not performing as expected to perform and we need to ensure that the issue is not owing to network bottleneck. The network bottleneck can be identified by calculating the response time theoretically for different pages based on the page size and comparing the same with the actual response time. If the time spent on the network for a page approximately equals the theoretical calculation, it is safe to assume that there is no network issue.

Network Time Calculation on 128kbps for a 60 KB page size

- Assuming 90% utilization of bandwidth
- Assuming 0 processing time

\[
\text{Network Time} = \frac{60 \times 8}{(90\% \times 128)} = 4 \text{ Seconds} \tag{6.1}
\]

Going by the calculation above a 60 KB page size will take 4 seconds on the network. Table 6.2 below shows that during the first round of performance testing the average % variation between theoretically calculated network time and actual network time was more than 40%, which was a good indicator to believe that there was some issue related to network. After doing an investigation we figured out that as the performance testing environment was sharing the corporate LAN with other users the same was constrained. An isolated network segment for performance testing was configured for round 2 and the average % variation between theoretically calculated network time and actual network time was less than 10%, which was a good indicator to believe that there was no issue related to network.
### Table 6.2: Network Response Time

<table>
<thead>
<tr>
<th>Page</th>
<th>Page Size without JS (KB)</th>
<th>Theoretical Calculation of Network Time</th>
<th>Actual Network Time (Round 1)</th>
<th>% Variation</th>
<th>Actual Network Time (Round 2)</th>
<th>% Variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Page1</td>
<td>50.8</td>
<td>3.5</td>
<td>6</td>
<td>41.67</td>
<td>4</td>
<td>12.50</td>
</tr>
<tr>
<td>Page2</td>
<td>29</td>
<td>2</td>
<td>3.5</td>
<td>42.86</td>
<td>2</td>
<td>0.00</td>
</tr>
<tr>
<td>Page3</td>
<td>25.4</td>
<td>1.8</td>
<td>3</td>
<td>40.00</td>
<td>2</td>
<td>10.00</td>
</tr>
<tr>
<td>Page4</td>
<td>34.5</td>
<td>2.4</td>
<td>3.5</td>
<td>31.43</td>
<td>2.5</td>
<td>4.00</td>
</tr>
<tr>
<td>Page5</td>
<td>31.8</td>
<td>2.2</td>
<td>3.8</td>
<td>42.11</td>
<td>2.5</td>
<td>12.00</td>
</tr>
<tr>
<td>Page6</td>
<td>48.9</td>
<td>3.4</td>
<td>5</td>
<td>32.00</td>
<td>3.6</td>
<td>5.56</td>
</tr>
</tbody>
</table>

**Average % Variation**: 38.34

**7.34**

### 6.3 Scalability Analysis of .NET

It is important to identify the options available in upgrading the hardware configuration, to handle increased user base while maintaining the same performance. Analyzing scalability requires three metrics [50,54].

**Speedup S**: increase in the rate of doing work with increase in the number of processors – ratio of the completion times, which can be extended to the ratio of throughputs

**Efficiency E**: measures the work rate per processor and is ratio of Speedup S and number of processors

Scalability Ψ (k1, k2): from k1 number of processors to k2 number of processors is the ratio of Efficiencies It has an ideal value of unity.

\[
Ψ(k_1, k_2) = \frac{E(k_2)}{E(k_1)} \cdot \frac{S(k_2)/k_2}{S(k_1)/k_1} \cdot \frac{X(k_2)/k_2}{X(k_1)/k_1} = \frac{X(k_2)}{X(k_1)}
\]  

(6.2)

Where, \(X(k)\) = Throughput of the application with \(k\) processors.
Using equation (1), the scalability of ASP.NET with SQL server 2000 was computed referring to Nile Benchmarks [53,55]. The computed results are shown in Table 6.3.

<table>
<thead>
<tr>
<th>(\psi(K1\ CPU,\ K2\ CPU))</th>
<th>Scalability</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\psi(2,4))</td>
<td>0.978</td>
</tr>
<tr>
<td>(\psi(2,8))</td>
<td>0.721</td>
</tr>
</tbody>
</table>

Table 6.3: Scalability across processors

Table 6.3 indicates that the architecture does not scale for 8 Processors. The scalability obtained by scaling up the number of processors from 2 to 8 is very low compared to scalability obtained by scaling-up the processors from 2 to 4. Hence, the scale-out approach of 4 processor servers (addition of 4 processor servers) should be adopted to handle increase in workload.

6.4 Performance Optimization

Tuning and optimization are often specific to the applications. Some generic steps taken to reduce the size of the pages and data transferred between the browser and the server for optimal use of the network bandwidth are:

- For low bandwidth access of a web app, most of the response time is spent on the network. For example, for a 128 kbps access, about 5 - 10 seconds could be on the network and 1 second on the server. We have to reduce the page size by introducing options like having Previous / Next for grids to reduce the number of rows in the grid, replace combo boxes by lookup screens and so on. In ASP.NET, we can also turn off the view state for the page / individual controls where possible. View state consumes significant amount of data. Changes for bandwidth
need not wait for the performance test to be done. All we need is to measure the size of all pages with data and then make appropriate changes to the screens to reduce the bandwidth.

- Code changes are rarely needed as long as we have a good design in place. Code changes are expensive and is disruptive in nature (presume bandwidth changes are done in advance) as multiple releases need to be done etc. It is a good idea to measure the time spent in activities using various tools like AQTime or VTune. AQtime and VTune are code profilers which can tell the time spent in each function. Ideally for high performance requirement project (>10 TPS), all code should document the timings as part of the unit test results. Given this information as well as the response time expected, a call can be made right then as to whether the code needs to be further tuned. Almost all code can be tuned for better performance but we really need to make a call as to whether we need to do it as it is expensive. A dual proc machine is around $10,000 and sometimes adding in extra machine might be cheaper compared to the development cost or maintenance cost.

- For most enterprise applications, this will be the key area to tune. In SQL Server for example, we need to ensure that we make use of stored procedure, have appropriate indexes created and ensure that the time spent in the database server is as less as possible (ideally 100 ms or less). The time spent in the server is usually a reflection of the time spent in the database. We will need a SQL Server specialist for handling / tuning Sql server for projects where quick response time
is crucial. SQL Server comes along with SQL profiler using which we can get to know the time spent in each stored procedure. Once we know which item to tune, SQL analyzer can be used to tune the SQL statements – the analyzer lets us know the cost of operation and we need to identify a solution.

For optimizing further, code was profiled and database was tuned to achieve acceptable performance. Some of the common techniques that improved performance are: Algorithm Optimization, Low-level program optimization (Use of String Builder, reference type, structured exceptions.), Addition of relevant indexes in the database and Addition of hints to SQL Query Plan.

6.5 Summary

Verification of performance and scalability requires the correctness of a performance test execution and analysis. The chapter highlights certain quantitative techniques and insights gained in analyzing performance and scalability of a .NET application using simple techniques.