ASSESSING THE PERFORMANCE OF HETEROGENEOUS SYSTEMS: A PRACTITIONER’S APPROACH

Jayesh Parameswaran
CONTENTS

Executive Summary .................................................. 3

Introduction ................................................................ 4

Surveying the System Landscape ................................. 5

Identifying an Effective Approach .............................. 6

Identify Specifics

Define Boundaries

Building the Testing Solution ................................... 11

Choosing a Framework or Toolset .......................... 11

Case Studies: Tool Selection .................................... 11

Simulating an Application Usage Model ................. 11

Accelerators for Load Model Design ...................... 11

Using the Testing Solution ....................................... 16

Execution Considerations ....................................... 16

Finding Patterns and Analyzing Results ................. 16

Post-Production Measures ....................................... 18

Production Issue Triage .......................................... 18

Continuous Production Monitoring ........................ 19

Conclusion .............................................................. 19
EXECUTIVE SUMMARY

As organizations increasingly rely on their applications—and as these applications become ever more complex and interdependent—the ability to design for and test performance has become critical. This white paper discusses some of the challenges that organizations face when testing the performance of heterogeneous systems with a slight focus on systems that support trading and investment operations within asset management organizations. The paper also presents practical recommendations to address those challenges and shares case studies illustrating how Sapient devised effective solutions for testing the performance of desktop-based client-server applications.
INTRODUCTION

In many enterprises, the application landscape has become larger and more complex, with a proliferation of heterogeneous systems that continuously exchange information. In many cases, organizations have more tightly integrated applications to support enterprise resource planning (ERP) solutions. They have also adapted to the introduction of new technology platforms, such as Java, and the resulting emphasis on web-based applications. More recently, IT organizations are facing the need to support increasing global connectivity using different types of user interfaces and an array of devices—from traditional workstations to tablets and smartphones. In the financial sector specifically, organizations must also address the demand for reliable, real-time information exchange amid high volumes of transactions.

As organizations become more reliant on diverse and interdependent IT systems, assessing and improving system performance is both more challenging and more important than ever before. Some of the most common challenges to assessing system performance include the lack of:

- Published standards for testing heterogeneous systems and architectures
- Proper planning and a sound, well-defined approach
- Off-the-shelf testing solutions that meet all performance testing requirements; despite the availability of numerous tools, many organizations are challenged to choose the right framework and toolset
- Expertise in defining the right statistics to measure and interpreting the results from the statistics collected
- In-house skills and talent to fine-tune performance, especially of financial applications. Compounding these challenges are the realities of service-oriented architecture (SOA), in which most communication occurs in an asynchronous manner. In these systems, it can be difficult to maintain the continuity of performance measurement tokens. Finally, there is the challenge inherent in how many organizations perceive performance testing. It is often viewed as a reactive step rather than a critical part of continuous delivery. What’s more, seldom do organizations proactively design for performance, establish benchmarks or continually measure system performance. It can be difficult to maintain the continuity of performance measurement tokens. Finally, there is the challenge inherent in how many organizations perceive performance testing. It is often viewed as a reactive step rather than a critical part of continuous delivery. Seldom do organizations proactively design for performance, establish benchmarks or continually measure system performance.

Often, performance tuning and system optimization are integral parts of the performance engineering discipline. While a very thin line distinguishes performance testing and profiling, profiling should not be done at the same time as testing. Both offer ways to collect and assess system statistics, but testing generally provides high-level statistics ("black box testing") while profiling focuses on specific areas within the system ("white box testing"). Thus, testing should be the first step to understanding system performance at a high level. Testing results offer insights on gaps between current and target performance, which can then be explored with more focused profiling and tuning. Indeed, testing, profiling and tuning should be performed one after the other in a cyclic fashion until optimal system performance is achieved. It is important to note that profiling and tuning are detailed topics that fall outside the scope of this paper. The focus of this paper is measuring and benchmarking system performance.
SURVEYING THE SYSTEM LANDSCAPE

For a number of years, performance testing has been evolving for both traditional, non-interactive systems and newer, web-based systems that are highly interactive. Today, even standard performance testing tools (such as HP LoadRunner, Micro Focus SilkPerformer and open-source tools, such as Grinder and JMeter) offer advanced capabilities for testing web-based systems in which end users access the system mainly via browsers or other thin clients. In years prior—and particularly among asset management firms—many applications were developed on non-web platforms. Given the complexities involved in automated performance testing, such applications were generally rolled out to production with only basic manual performance tests. Further, extensive system characterization and profiling exercises were generally not completed. Consequently, chances were high that stakeholders would notice performance degradations only after actively using the actual production systems. In addition, such systems were not designed for the massive user base that is now customary for external business-to-consumer web applications. Even so, these systems must handle high volumes of transactions. Without appropriate system profiling, resolving performance bottlenecks in production is akin to finding the proverbial needle in a haystack.

Today, stakeholders are increasingly unified in their desire to have robust performance testing strategies across heterogeneous application environments (see Figure 1 for an application environment typical to asset management organizations). As a result, IT managers’ mind sets are gradually shifting; they are now willing to allocate increased budget and attention to performance testing.

Given the complexity of today’s application landscapes, a generic, one-size-fits-all approach to performance testing is simply not viable. Thus, if the ultimate goal is an enterprise-level performance testing capability, each system should be addressed individually before advancing toward wider coverage and optimization.

Figure 1: Heterogeneous application environment typical to asset management organizations

Each block represents interconnected subsystems; output of some systems serves as input for other downstream systems.
IDENTIFYING AN EFFECTIVE APPROACH

By addressing performance testing challenges system by system or component by component, an organization can achieve a holistic testing solution at each step. However, for individual systems, initial tasks are not always the same. As a general practice, Sapient recommends working iteratively on three main areas for a sound performance testing approach (see Figure 2):

**Table 1: Sample performance indicators generated during performance testing**

<table>
<thead>
<tr>
<th>Business Performance Indicators</th>
<th>Technical Performance Indicators</th>
<th>Financial Performance Indicators</th>
</tr>
</thead>
<tbody>
<tr>
<td>Response time</td>
<td>Memory usage</td>
<td>Lifespan of the system</td>
</tr>
<tr>
<td>Throughput (transaction/unit time)</td>
<td>CPU Usage</td>
<td>New hardware/software procurement</td>
</tr>
<tr>
<td>Availability</td>
<td>Page I/O</td>
<td>System maintenance budget</td>
</tr>
</tbody>
</table>

**Identify Specifics**

Each business system has its own characteristics and target performance criteria. Thus, it is imperative that performance engineers understand each system’s distinctive nature and characteristics in order to develop an optimum strategy. For an asset management firm, those factors would include the business problem or problems that the system addresses (see Table 2).
Step 1: Ask Questions

In many cases, performance engineers may not be part of a given system’s application development team and therefore may lack in-depth knowledge of the application. Consequently, such engineers need to interview representatives from various stakeholder groups, including the development team, infrastructure team, quality assurance team, management team and business analysis team, to become familiar with each system. Because stakeholders’ perspectives are likely to differ, the knowledge collected may be in “bits and pieces.” Thus, it should be synthesized and documented, preferably in the form of a reference document. Sample questions for each type of stakeholder follow:

Table 2: Assessing systems and identifying performance criteria

<table>
<thead>
<tr>
<th>Description</th>
<th>Users</th>
<th>Transaction Volume</th>
<th>Peak Load Event</th>
<th>Important Performance Criteria</th>
</tr>
</thead>
<tbody>
<tr>
<td>Front Office Systems</td>
<td>• Desktop or web interface user interface</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Sensitive to message exchange and connectivity</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Trading staff ranging from 25 to 100 users.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Other staff, such as trade administrators and broker/dealers</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Maximum and potentially volatile</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Direct correlation with real-time events</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Examples include:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Notification day for mortgage trades</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Month-end close</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Roll for future trades</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mid Office Systems</td>
<td>• Mostly interactive systems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Back Office Systems</td>
<td>• Mostly batch processes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Few interactive systems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Moderate base of users from accounting</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>and operations departments</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Potentially huge transaction volumes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• May also include data processing</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Inter connectedness of various processes</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• is more likely to affect timely</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>performance than particular market events</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Central Shared Systems</td>
<td>• Proprietary users</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Third Party</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Interactive users</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Requests via application protocol</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Interface (API) calls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Moderate to maximum transaction volume</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Depending on service requests from</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• external systems</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Any market event might affect these</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• systems performance</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Response time</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Scalability</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Questions Pertaining to Business Service Level Agreements (SLAs)

- Based on previous experience, what is the length of response time at which users begin complaining about completion of critical business transactions? (The response time identified in the objective-setting phase should be well within this threshold level and other technical constraints.)

- How sensitive is each use case for the users?

- How many users are expected to be active in the system across various periods of time?

- What is the expected concurrent number of users executing any specific use case (classified into low, medium and high buckets)?

- What is the peak size of users expected on a typical business day?

- On average, how many transactions are expected to be completed in a second? (In other words, what is the standard throughput that users expect?)

- What is the extrapolation regarding the rate of growth of data and users for at least a couple of years in future? (This data will be helpful while designing a realistic load model.)

Questions Pertaining to Technical Constraints

- What are the hardware configurations used?

- What are the network protocols, topology and load from other systems used?

- Are there any organizational standards to follow? If so, are those standards suitable for the application under test?

- If the application is Java based, what could be the current Java Virtual Machine (JVM) memory sizing?

Questions Pertaining to Financial Constraints

- Are you seeking to procure additional resources, or are you just looking for an assessment of current capability?

- Can additional load/network simulation software be procured if needed?

- Are the budget and need suitable for commercial simulators if required—for example, a Financial Information eXchange (FIX) protocol simulator or other utilities?

After receiving sufficient input, the performance engineers should categorize the answers to all of these questions. Ultimately, this exercise ought to lead to a greater understanding of the system—including fine-tuned goals for self-assessment.

Step 2: Visualize and Design the Simulation Environment

In addition to collecting the necessary application-related information, performance engineers are well advised to build a picture of the simulation environment and start developing guidelines on preparing the test environment. Consider, for example, the case of a trading application under test. By the very nature of a trading application, it would need to communicate with multiple systems simultaneously for various purposes, such as:

- Order creation
- Quote generation
- Trade execution and allocation
- Real-time update and price verification
- Various compliance checks
- Processing events related to parallel order executions by other traders
- Jobs that would aggregate or otherwise enhance transactional data in the background
While executing the performance tests on a trading system, these and other essential events and transactions should be simulated in the background. Such simulation details are most effective only if the simulation infrastructure is equivalent to the actual production infrastructure. Generally speaking, testing environments are scaled-down environments—an important difference to consider when analyzing results. (In “Using the Testing Solution,” we discuss testing infrastructure criteria in greater detail.)

Apart from the infrastructure, there should be absolute control over test data, as well. When framing test data, performance engineers need to ensure that the data are diverse and represent different categories. What follows are some examples of criteria that performance engineers can use to address diversity:

- Asset class. Ensure the availability of representative data from a range of asset classes, including Fixed Income and Equity and Derivatives.
- Compliance rules. Ensure that chosen data touch upon different compliance rules, such as pre-trade and legal requirements.
- Holdings. Carefully consider how many accounts and/or portfolios are holding a particular security.

Depending on the type of test data chosen, benchmarks may also need to be set for various types. For repeated execution, this “golden” set of data should not change so as to have an ideal comparison of results.

**Step 3: Choose the Representative Sample of Test Cases**

Most applications can have many parallel workflows; of those, only a few are commonly used. It is neither possible nor necessary to create performance test cases for every possible scenario. As with most test cases, the 80-20 rule applies. Accordingly, 20 percent of the cases that are commonly used and are performance sensitive need to be identified. Once enough familiarity with the system is developed, one should also have a clear understanding of all the workflows the system supports and which are relevant for testing. Considering the case of a portfolio management tool, some of the main workflows could be loading the portfolio, editing the portfolio, managing cash flow and submitting orders. In addition, the tool might provide other features, such as transaction history lookup and audit trial view, which are not commonly used. Such cases could be removed from the list of workflows chosen to simulate. However, there are certain other cases that need not be measured but are repeatedly executed in the system. An example is a service that maintains user-view updates after receiving events from a central cache with information on order submissions from other users. Such cases should definitely be included in the list for automation since they are necessary to simulate a realistic scenario.

**Step 4: Baseline and Benchmark**

Baselining is the activity in which application performance statistics are measured under ideal conditions and then recorded as a comparative reference for the future. Sometimes, performance engineers prefer to have two sets of baseline measurements—one with a single user load and another with an ideal production load.

Consider the response time of certain transactions. Typically, statistics (such as average time, maximum time, minimum time and response time) are captured at a certain percentile of the sample distribution. Engineers can use these percentile values to cut the “tail end,” thereby removing potential outliers. Depending on the distribution, the average or percentile values could be used as the baseline. In general, when there is minimum standard deviation and no outliers, the average value could be considered as the baseline.

In the case of a “golden” set of test data, performance engineers need to have a data snapshot while executing the baseline tests. Doing so helps in reusing the data set during performance testing of the application at a later time. A simple Insert/Update/Delete script could be used to create this golden data snapshot.
Benchmarking is the process of calibrating the given application performance against industry standards. Various standards are available for this purpose. The Standard Performance Evaluation Corporation is a well-established standards body for performance benchmarks.

**Define Boundaries**

In addition to studying system architecture and prospective usage in production, performance engineers must also consider other factors and set appropriate boundaries for them. These boundaries define the scope of effort and may vary depending on the system’s phase of life, source control aspects and scripting aspects. We describe these and other factors below.

**Application in production**

In many cases, an application is already in production and the need for performance testing comes much later. In such cases, performance engineers must either quickly develop a holistic performance testing strategy or focus on the already known top-five vulnerability areas while scripting the use cases. When facing a quick turnaround, it is prudent to go with the latter option.

**Brand-new application**

For a new application to be released into production, performance testing activity should start as early as possible. Such an approach will help ensure ample time to devise a holistic testing strategy or focus on the already known top-five vulnerability areas while scripting the use cases. When facing a quick turnaround, it is prudent to go with the latter option.

Performance engineers also need to closely work with system architects to brainstorm areas in which performance is critical and can be proactively optimized. Some examples may include:

- Sending data over the network and ensuring that data is compressed wherever possible
- Reading data from hard drives and making use of in-memory cache and other optimizations
- Interacting with databases and ensuring that any object relational mapping (ORM) is not causing bottlenecks
- Providing real-time updates and making sure these updates are processed incrementally
- Performing front-end data translation and ensuring that only the data relevant to the specific user is being translated
- Delivering pagination techniques and making sure that only the data the user can handle at one time is being shown on screen

Ultimately, the goal is to keep tabs on any potential loopholes that could degrade performance in the system—and to address those areas much earlier in system development.

Performance engineers are also responsible for generating awareness within the team about guidelines to build a system that performs well. During development tasks, the development team should be advised to code performance test cases. Best practices suggest instrumenting the code base and having a well-designed metrics collection capability within the system itself. Frameworks, such as JUnitPerf and PerfTest, can be used to instrument the code and to get single-user-response measurements. However, it is important to note that these frameworks are not capable of simulating load; they are to be used with other load testing solutions to provide measurement under real scenarios.
BUILDING THE TESTING SOLUTION

Once performance engineers have determined the main criteria for performance assessment, it is time to think through execution. At this stage, various considerations could include these and other “whats” and “hows”:

- Framework and toolset for achieving the goals
- Usage model that would be simulated
- Monitoring model or statistical data that might be collected during execution
- Ensuring the availability of representative data from a range of asset classes, including Fixed Income and Equity and Derivatives.

Choosing a Framework or Toolset

Buy vs. Build

An abundance of commercial and open-source performance testing tools are now available. All aim to address the challenges inherent in testing complex applications for non-functional attributes, such as performance. However, our experience suggests that no single tool fully addresses all performance testing requirements. Therefore, the key to any successful performance testing engagement is devising a sound strategy in advance and then creatively selecting the suitable and required tools or framework from the available set.

Performance engineers can consider three different categories of assisting tools:

- Commercial Tools. Commercial tools are highly advanced and arguably the best in terms of capabilities for recording use cases. Such tools help even non-programmers to simulate simple cases easily and very quickly. The depth of reporting coverage and analysis is another area where commercial tools score much higher than other options.

If an organization already has commercial tools, performance engineers should strongly consider using them. In other cases, it is probably better to go for such tools only if the need is for detailed and extensive performance classification and characterization. Generally, the limitations of such tools are that the licensing model would make the investment more expensive. For example, vendors may charge additional fees for each individual protocol for tools. Thus, organizations need to complete an updated cost-benefit analysis before choosing them.

- Open-Source Tools. Open-source tools may not offer as many features as their commercial counterparts. However, performance engineers should consider open-source tools when their needs are not extensive. Open-source tools are also helpful in developing exploratory performance test assets and in maintaining continuous, automated testing.

- In-House Tools. Another possibility is to use frameworks developed in house. When a specific application team has already developed a particular performance testing or monitoring framework, it should be considered for other applications of similar architecture. However, such frameworks do not always scale well when considered for distinct applications. Enhancing and supporting such frameworks also introduces requirements around specific in-house technical expertise.

Variety of Categories

Another classification of tools is based on the various functions they provide during testing. As an example, consider some of the out-of-the-box tools and framework offerings and the general categories they fall under (see Figure 3).
Usability Factors

In addition to availability and suitability, certain usability considerations are also critical to making a final decision on tools and framework:

- **Learning Curve.** Most commercially available tools are relatively easy to learn and use. However, there are still some with a steep or gradual learning curve. Thus, organizations need to consider this factor vis-à-vis the immediacy of the need, as well as the expertise of the performance testing team.

- **Interoperability.** Because it is nearly impossible for a single tool to address all performance testing needs, the ability of selected tools to work with each other is another important consideration.

- **Maintainability.** Scripts and other test assets often need modification, enhancement and rework. Consequently, it important to consider the effort required to maintain the test suite and other assets.

Based on these considerations, performance engineers can formulate a clearer vision around which tools or frameworks will be used. With a proper understanding of all the factors discussed—along with some additional development considerations—performance engineers can acquire the vital clues required for an optimized approach.

Maintenance of Test Assets and Reports

Maintaining scripts is critical to repeated iterations of executions. Therefore, enough attention should be paid to matters such as the script repository to be used...
and how frequently to test-run the scripts. Overall script maintenance is as important as the application source code control and maintenance, and it should be treated accordingly. It is a good idea to run the scripts at regular time periods to ensure that they work whenever required for testing. The results are also recommended to be saved at a location for later reference. Although the results need to be re-benchmarked over time, the old results can help in understanding performance improvement patterns.

Good Practices for Scripting
While script development processes can share some maintenance similarities with application code development, there are certain differences to keep in mind. Typically, a good testing tool provides the capability to record and replay the workflow, and the performance engineer usually modifies the generated scripts for parameterization, correlation and more. In such cases, detailed documentation and reusability need not be given focus to the same extent as application source code. It is important that scripts should be readable and very simple. Also, the objective of the test suite should be to execute tests with various virtual user behavior configurations and the ability to rerun and report results consistently.

Case Studies: Tool Selection
Case Study 1: Thick Client Application
The application had a Java swing client at the portfolio managers’ desktops communicating with a Java server application deployed on an Oracle WebLogic server. The transactional data was ultimately stored in an Oracle database at the back end. Portfolio managers were using this application for such activities as portfolio edits and views and trade proposals. Consequently, the performance engineer required an automated performance testing solution that would report the response times experienced by portfolio managers for main workflows. In addition, the solution had to be able to report hardware performance statistics.

Application Architecture
There were many hurdles for automating performance testing of this thick client application. What follows are some of the common methods of testing performance:

- Measuring the response time manually for single-user experience
- Measuring the response time manually under load by many people accessing the system at a predefined time window
- Measuring the time taken for relevant application code by iterative execution of integrated API calls. In this case, performance engineers needed a test suite that would cater to all manual requirements while reducing the manual effort in test execution.

Solution Overview
Clearly, the tools had to offer the ability to:

- Apply load
- Measure from the end-user perspective
- Measure correlated hardware performance

For the specific application under test, it was observed that there were two different types of communication happening between client and server. One was the Remote Method Invocation (RMI) and the other was Java Messaging Services (JMS). The proposed solution was as shown in the diagram below:
HP LoadRunner: HP LoadRunner offered a suitable protocol for recording and replaying Java methods using both RMI and JMS. These scripts were used to simulate the load on the server and to generate various graphs, such as response time pattern under load and throughput.

All required client libraries in the VuGen editor were to be provided and the workflows on the application under test were to be recorded. LoadRunner could load the Java hooking libraries to inspect any calls via RMI or JMS between client and server while executing the workflow. Later, these calls were generated into a source script that could be further analyzed and modified. The dynamic objects that were passing back and forth between client and server were serialized into data files, and LoadRunner provided APIs for reading and modifying them. A little knowledge of Java and the application-specific client libraries made the script replay successfully with a moderate level of effort.

LoadRunner VuGen’s regular features, such as parameterization and correlation, were used to further enhance the scripts. Hence, the scripts developed as described were capable of being replayed iteratively and repeatedly. By creating a load scenario with required number of users executing this script, sufficient load was assured in the server. The usual LoadRunner analysis can be used to assess response times for executing the API calls.

HP QTP: One shortcoming in the LoadRunner scenario mentioned above was that it could not give the exact user perceived timings. Therefore, QTP scripts were used to measure response time from the GUI on the user desktop. Using the LoadRunner GUI license, the QTP scripts from the LoadRunner controller and regular VuGen scripts were invoked. The functional tools, such as QuickTest Professional, helped to address the requirements.

HP SiteScope: Finally, performance counters—such as CPU usage, memory usage and IO usage on the application and database servers—were to be monitored during testing. HP SiteScope is a tool that can monitor system usage remotely without any agent installation on the server. The counters that were measured using SiteScope were imported to LoadRunner so that correlated graphs for further analysis could be generated. Another advantage with SiteScope was that operating system-specific scripts could be written and configured, enabling it to monitor any performance counter other than off-the-shelf monitors provided.

Case Study 2: Thin Client Application

If an application’s front end is developed using any web framework, the complexity of the performance testing solution may be reduced. Suppose that the application has used Web 2.0 technologies. One of the relatively complex tasks would be measuring processing time of JavaScript and AJAX call simulations. Such applications can range from those handling simple CRUD (Create, Read, Update and Delete) functionality to complex trading functionalities.

Solution Overview

Providing advanced capabilities using multi-protocol support, the HP tools mentioned above can be leveraged in such cases. In addition, other measurements could have been there as described below.

HP LoadRunner and HP SiteScope: A combination of HP tools—namely, LoadRunner and SiteScope—provide excellent coverage for measuring web applications’ performance statistics. They are very advanced and offer system measurement and knowledge that are difficult, if not impossible, to match.

Browser performance measurement tools: Even though HP LoadRunner could be useful to measure response time for web applications, at times it would not measure JavaScript processing time and page rendering time at the user end. HP LoadRunner typically measures server processing time and network round trip time, but not browser rendering time. In these instances, tools such as dynaTrace AJAX, HTTPWatch and Fiddler, could be used to measure total response time as perceived by the end user.
Simulating an Application Usage Model

The simulation of application utilization during a typical performance test scenario depends on two variables: the user load that must be simulated on the system during the test and the total duration of the test execution. Both variables vary depending on which performance attribute is being measured and analyzed (see Table 3).

<table>
<thead>
<tr>
<th>Ideal Response Time Test</th>
<th>User Load</th>
<th>Duration of the Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single User</td>
<td>4-5 Iterations</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stability Test (Endurance)</th>
<th>User Load</th>
<th>Duration of the Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation of virtual users on a typical business day</td>
<td>5 hours to 1-2 days</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Scalability Test (Stress)</th>
<th>User Load</th>
<th>Duration of the Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-4 times the number of virtual users on a typical business day</td>
<td>2-3 hours</td>
<td></td>
</tr>
</tbody>
</table>

This table depicts the variability and desired range of user load and total duration of the test.

Response Time: When the objective is to measure the response time of certain transactions performed by the application under ideal conditions, it is recommended to keep minimum load in the system and a limited number of iterations of test suite executions that would provide enough data to analyze and validate. This data should be a good candidate for baseline measurement.

Stability: When the testing objective is to forecast application stability, the load should be as much as that on a typical day in the production system and the tests need to be executed for a longer duration (for example, more than half a day or so).

Scalability: To estimate the current capacity and to understand the system’s scalability, performance engineers should conduct specific scalability tests. In this case, the load should be increased gradually starting with regular ideal load (such as 2x or 3x, where \( x \) is the anticipated ideal user load in production). Subsequently, the user load could be ramped up until the system buckles and begins to collapse.

Accelerators for Load Model Design

Performance engineers can employ various tools and techniques to come up with a system usage model. Scientifically, engineers can build their models using such methods as queue theory or operational laws.

Because detailed descriptions of these theories are beyond the scope of this paper, performance engineers may benefit from referring to a good performance testing guide to get a deeper understanding of these methods. One of the more straightforward methods is to parse the log files generated in the production environment. Certain tools, such as Splunk, have the capability to parse and index the log files and help deduce patterns of application usage. This could further assist in developing a realistic usage model.

Load Calculation Based on Future Usage Prediction

In setting objectives, the span of system usage is among the questions to be addressed. Typically, most financial systems have very long lives. A minimum of four to five years is the standard timeframe in which these systems are expected to function without major architectural changes. Therefore, it is good to extrapolate the number of current users for a few years ahead and determine load accordingly. To project future usage, performance engineers should read general industry trends along with any available data regarding past usage of similar systems. Particularly while conducting scalability tests, such an understanding of usage will help to define the load. If the current infrastructure is not capable of handling that load, infrastructure enhancement will be required to accommodate future needs.
Execution Considerations

Building the testing solution is just one part of the solution; defining various parameters that determine the execution context is another critical part of any performance testing exercise. Based on experience, some important factors need to be considered before initiating execution.

Test Infrastructure

- **Test Environment.** To the greatest extent possible, the testing environment should be built and configured identically to the production environment.

- **Test Machines.** It is important for performance engineers to possess a good understanding of the test machines on which scripts are running. Such an understanding should include hardware configuration, memory footprint and any additional overhead.

- **Test Scripts.** Ensure that scripts are properly measuring the defined transactions while filtering out think time, delay time and wasted time.

- **Iterations of Test Execution.** Whether conducting a baseline test, load test or stress test, performance engineers are well advised to execute multiple iterations of the tests (five times, at minimum) to ensure conclusive results.

- **Known Tool Limitations.** Even the most suitable tools can have certain limitations. For example, in one of the case studies mentioned above, QuickTest Professional, which is used to measure response time, takes certain CPU cycles, thereby requiring measurement of corresponding overhead.

Continuous Execution

The most important advantage of automated testing is its ability to iteratively execute the test suite once developed. An obvious question—especially among people accustomed to agile methods—is “How can we execute the test suite in an automated manner with every build?” In answering this question, it is important to note that performance engineering resources are generally shared across the organization. Often, it is not practical for the team to perform full-fledged performance testing execution and further analysis on consistent and frequent demands, especially when commercial products such as HP LoadRunner are being used. In such situations, organizations should consider alternative non-commercial solutions with minimal dependencies.

Open-source tools that are actively developed and maintained, such as JMeter, may prove very helpful if measurements need to be taken under load. JMeter provides a feature in which regular JUnit test cases developed as part of the development of application can be converted to JUnit samplers or requests in JMeter and test plans could be created (see Figure 5).
What if there is no wide JUnit test case coverage for the application under test? Most of the time, organizations need tests performing complete business transactions rather than modular unit test cases. That, in turn, may require extended time and effort to devise such integrated test cases for a workflow. Therefore, performance engineers should seek out other creative solutions to tackle this problem. One example: checking the possibility of converting scripts developed by any commercial tool to JUnit test cases. (This is certainly possible with HP LoadRunner Java Vuser scripts.)

Finding Patterns and Analyzing Results
Pattern detection and analysis are arguably the most important part of the entire performance testing effort. With data collection complete, the next step is to analyze and extract knowledge about the system. Generally, most testing tools provide various views of the captured data. Recommendations for analyzing the results follow:

• **Look at Errors First.** Error details should be examined before any other statistics. (As a practice, never stop a test when an error occurs; let the test finish.)

• **Distinguish Between Application Errors and Test Constraints.** For example, if the test configuration has a constraint on connection time or resource download time, this might be reported as an error. Such configurations should be verified and changed if necessary.

• **Inspect Data from Different Angles.** An advantage of commercial tools is their ability to generate various graphs. Merging or overlaying such graphs can provide more relevant information than analyzing individual graphs and trying to identify meaningful patterns.

• **Apply Statistical Concepts.** Depending on the nature of data, statistical distributions can be used (normal and Poisson distributions are some of the commonly used statistical models), and aggregations can be performed. The results should be taken as guidance toward conclusions. Always keep in mind that the relevance of some of those values (such as simple averages) will depend on the pattern of occurrence of sample values being considered. Bottom line: Data and patterns should be analyzed very diligently.
**Production Issue Triage**

Even after sufficient pre-production testing, it is common to experience a slowdown in production. In such cases, meaningful and interpretable log files, along with effective auditing services, can be helpful in triaging the issues. Consequently, both logging and auditing should receive sufficient attention to ensure that post-production triage is a relatively smooth exercise. Apart from logging and auditing, another best practice is to have a service that can collect metrics and asynchronously save them into a remote system without affecting application server performance. The data collected can be read and analyzed at any point needed—a capability that may help in proactively detecting any performance issues. Later, as a best practice, the data snapshot and use case scripts should be created and made part of the performance testing suite if not already available.

**Continuous Production Monitoring**

Certain basic web traces or synthetic transactions could be configured in an organization’s Application Performance Management tool. If the organization does not have such a tool, it may reflect an organizational infrastructure monitoring capability gap that should be brought to stakeholders’ attention. What follows are examples of commercially available products:

- **HP Business Service Management.** This product can give a 360-degree view of details pertaining to high availability and performance. The end-user management component of this product can sit on a client system and help to understand user behavior. Meanwhile, the business process management component can integrate LoadRunner VuGen script or QTP script for simulating the business use case and can generate continuous monitoring graphs and reports.

- **Compuware Wily CEM.** Along with another industry-recognized product (dynaTrace), this Compuware offers an alternative product that can address this space.
CONCLUSION

Assessing the performance of applications can be as simple as manually measuring response times using a stop watch or as complex as building a robust testing ecosystem with various tools and frameworks. Regardless of the level of complexity, following an inquisitive, creative and corroborative approach throughout the various phases is necessary to extract maximum value from a testing exercise.

To be sure, system diagnosis is only as good as the data collected during the testing. The quality of data, in turn, depends on many factors as discussed in this paper, as well as many other application-specific variables. The methods described in this paper can serve as a starting point for worthwhile performance assessment. Suitable enhancements and modifications are always recommended depending upon the nature of application under test.

As performance testing slowly and steadily gains importance, the ultimate goal for organizations should be to reach a level where all of their systems can be tested in concert for any business event simulation. While this probably sounds like a tall order, the benefits of such a capability would prove worth the effort.

About Sapient Global Markets
Sapient Global Markets, a division of Sapient® [NASDAQ: SAPE], is a leading provider of services to today’s evolving financial and commodity markets. We provide a full range of capabilities to help our clients grow and enhance their businesses, create robust and transparent infrastructure, manage operating costs, and foster innovation throughout their organizations. We offer services across Advisory, Analytics, Technology, and Process, as well as unique methodologies in program management, technology development, and process outsourcing. Sapient Global Markets operates in key financial and commodity centers worldwide, including Boston, Chicago, Houston, New York, Calgary, Toronto, London, Amsterdam, Düsseldorf, Geneva, Munich, Zurich, and Singapore, as well as in large technology development and operations outsourcing centers in Bangalore, Delhi, and Noida, India.

For more information, visit sapientglobalmarkets.com.

© 2011 Sapient Corporation.
Trademark Information: Sapient and the Sapient logo are trademarks or registered trademarks of Sapient Corporation or its subsidiaries in the U.S. and other countries. All other trade names are trademarks or registered trademarks of their respective holders.
EQUITY DERIVATIVES STANDARDISATION: A Critical Element in Curbing Systemic Risk