18

June 2012

testing experience

The Magazine for Professional Testers

printed in Germany

print version 8,00 €

free digital version

www.testingexperience.com

ISSN 1866-5705

Test Center of Excellence

How can it be set up?

Load Testing: Respect the Difference

by Alexander Podelko

Talking about Test Centers of Excellence, it is important to remember that the skills and processes needed for load testing are quite different from functional testing. Often that results in creating a separate Performance Center of Excellence, but regardless of specific organizational structure, it is important to understand and respect the difference.

This article tries to contrast load testing with functional testing and highlights points that are often missed by people moving into load testing from functional testing or development. Applying the best practices and metrics of functional testing to load testing quite often results in disappointments, unrealistic expectations, sub-optimal test planning and design, and misleading results. While people who were involved in load testing or performance analysis may find many statements below to be trivial, it still may be beneficial to highlight the differences when we discuss processes and skills needed for Test Centers of Excellence.

Testing multi-user applications under realistic as well as stress loads remains the main way to ensure appropriate performance and reliability in production. There are many terms defining such kind of testing: load, performance, stress, scalability, concurrency, reliability, and many others. There are different (and sometimes conflicting) definitions of these terms, and these terms describe testing from somewhat different points of view, so they are not mutually exclusive.

While each kind of performance testing may be somewhat different, in most cases they use the same approach: applying multi-user synthetic workload to the system. We mostly use the term "load testing" further in that article because we try to contrast multi-user load testing with single-user functional testing. Everything mentioned here applies to performance, stress, scalability, reliability and other kinds of testing as far as these features are tested by applying load.

Load Testing Process Overview

Load testing is emerging as an engineering discipline of its own, based on "classic" testing from one side, and system performance analysis from another side. A typical load testing process is shown in figure 1.

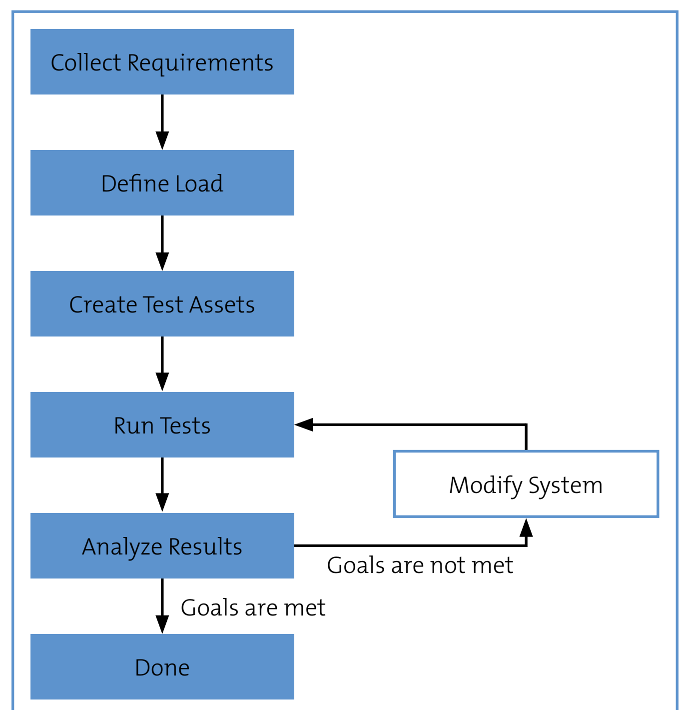

Fig.1 Load testing process

We explicitly define two different steps here: "define load" and "create test assets". The "define load" step (sometimes referred to as workload characterization or workload modeling) is a logical description of the load we want to apply (like "that group of users login, navigate to a random item in the catalog, add it to the shopping cart, pay, and logout with average 10 seconds think time between actions"). The "create test assets" step is the implementation of this workload, and conversion of the logical description into something that will physically create that load during the "run tests" step. While for manual testing that can be just the description given to each tester, usually it is something else in load testing – a program or a script.

Quite often load testing goes hand-in-hand with tuning, diagnostics, and capacity planning. It is actually represented by the back loop on the fig.1: if we don't meet our goal, we need optimize the system to improve performance. Usually the load testing process implies tuning and modification of the system to achieve the goals.

Load testing is not a one-time procedure. It spans through the whole system development life cycle. It may start from technology or prototype scalability evaluation, continue through component / unit performance testing into system performance testing before deployment and follow up in production (to troubleshooting performance issues and test upgrades / load increases).

What to Test

Even in functional testing we have a potentially unlimited number of test cases and the art of testing is to choose a limited set of test cases that should check the product functionality in the best way with given resource limitations. It is much worse with load testing. Each user can follow a different scenario (a sequence of functional steps), and even the sequence of steps of one user versus the steps of another user could affect the results significantly.

Load testing can't be comprehensive. Several scenarios (use cases, test cases) should be chosen. Usually they are the most typical scenarios, the ones that most users are likely to follow. It is a good idea to identify several classes of users – for example, administrators, operators, users, or analysts. It is simpler to identify typical scenarios for a particular class of users. With that approach, rare use cases are ignored. For example, many administrator-type activities can be omitted as far as there are few of them compared with other activities.

Another important criterion is risk. If a "rare" activity presents a major inherent risk, it can be a good idea to add it to the scenarios to test. For example, if database backups can significantly affect performance and should be done in parallel with regular work, it makes sense to include a backup scenario in performance testing.

Code coverage usually doesn't make much sense in load testing. It is important to know what parts of code are being processed in parallel by different users (that is almost impossible to track), not that particular code path was executed. Perhaps it is possible to speak about "component coverage", making sure that all important components of the system are involved in performance testing. For example, if different components are responsible for printing HTML and PDF reports, it is a good idea to add both kinds of printing to testing scenarios.

Requirements

In addition to functional requirements (which are still valid for performance testing – the system still should do everything it is designed to do under load), there are other classes of requirements:

- Response times – how fast the system handles individual requests or what a real user would experience
- Throughput – how many requests the system can handle
- Concurrency – how many users or threads can work simultaneously

All of them are important. Good throughput with long response times often is as unacceptable as good response times, but just for a few users.

Acceptable response times should be defined in each particular case. A response time of 30 minutes may be excellent for a big batch job, but it is absolutely unacceptable for accessing a Web page for an online store. Although it is often difficult to draw the line here, it is rather a usability or common sense decision. Keep in mind that for multi-user testing we get multiple response times for each transaction, so we need to use some aggregate values like averages or percentiles (for example, 90% of response times are less than this value).

Throughput defines the load on the system. Unfortunately, quite often the number of users (concurrency) is used to define the load for interactive systems instead of throughput. Partially because that number is often easier to find, partially because it is the way how load testing tools define load. Without defining what each user is doing and how intensely (i.e. throughput for one user), the number of users is not a good measure of load. For example, if there are 500 users running short queries each minute, we have throughput of 30,000 queries per hour. If the same 500 users are running the same queries, but one per hour, the throughput is 500 queries per hour. So with the same 500 users we have a 60-fold difference between loads and, probably, at least 60-fold difference in hardware needed.

The intensity of load can be controlled by adding delays (often referred as "think time") between actions in scripts or harness code. So one approach is to start with the total throughput the system should handle, then find the number of concurrent users, get the number of transactions per user for the test, and then try to set think times to ensure the proper number of transactions per user.

Finding the number of concurrent users for a new system can be tricky too. Usually information about real usage of similar systems can help to make an initial estimate. Another approach may be to assume what share of named (registered in the system) users are active (logged on). So if that share is 10%, 1,000 named users results in 100 active users. These numbers, of course, depend greatly on the nature of the system.

Workload Implementation

If we work with a new system and have never run a load test against it before, the first question is how to create load. Are we going to generate it manually, use a load testing tool, or create a test harness?

Manual testing could sometimes work if we want to simulate a small number of users. However, even if it is well organized, manual testing will introduce some variation in each test, making the test difficult to reproduce. Workload implementation using a tool (software or hardware) is quite straightforward when the system has a pure HTML interface, but even if there is an applet on the client side, it may become a serious research task, not to mention dealing with proprietary protocols. Creating a test harness requires more knowledge about the system (for example, an API) and some programming skills. Each choice requires different skills, resources, and investments. Therefore, when starting a new load-testing project, the first thing to do is to decide how the workload will be implemented and to check that this way will really work. After we decide how to create the workload, we need to find a way to verify that the workload is really being applied.

Workload Verification

Unfortunately, an absence of error messages during a load test does not mean that the system works correctly. An important part of load testing is workload verification. We should be sure that the applied workload is doing what it is supposed to do and that all errors are caught and logged. The problem is that in load

testing we work on the protocol or API level and often don't have any visual clues that something doesn't work properly. Workload can be verified directly (by analyzing server responses) or, in cases where this is impossible, indirectly (for example, by analyzing the application log or database for the existence of particular entries).

Many tools provide some ways to verify workload and check errors, but you need understanding what exactly is happening. Many tools report only HTTP errors for Web scripts by default (such as 500 "Internal Server Error"). If we rely on the default diagnostics, we could still believe that everything is going well when we are actually getting "out of memory" errors instead of the requested reports. To catch such errors, we should add special commands to our script to check the content of HTML pages returned by the server.

The Effect of Data

The size and structure of data could affect load test results drastically. Using a small sample set of data for performance tests is an easy way to get misleading results. It is very difficult to predict how much the data size affects performance before real testing. The closer the test data is to production data, the more reliable are test results.

Running multiple users hitting the same set of data (for example, playback of an automatically created script without proper modifications) is an easy way to get misleading results. This data could be completely cached, and we will get much better results than in production. Or it could cause concurrency issues, and we will get much worse results than in production. So scripts and test harnesses usually should be parameterized (fixed or recorded data should be replaced with values from a list of possible choices) so that each user uses a proper set of data. The term "proper" here means different enough to avoid problems with caching and concurrency, which is specific for the system, data, and test requirements.

Another easy trap with data is adding new data during the tests without sufficient considerations. Each new test will create additional data, so each test would be done with a different amount of data. One way of running such tests is to restore the system to the original state after each test or group of tests. Or additional tests can be performed to prove that a change of data volume inside a specific range does not change the outcome of that particular test.

Exploring the System

At the beginning of a new project, it is a good practice to run some tests to figure out how the system behaves before creating formal plans. If no performance tests have been run, there is no way to predict how many users the system can support and how each scenario will affect overall performance. Modeling can help here to find the potential limits, but a bug in the code or an environmental issue can dwarf scalability.

It is good to check that we do not have any functional problems. Is it possible to run all requested scenarios manually? Are there any

Certified Agile Tester

Book your CAT training in USA!

CAT is no ordinary certification, but a professional journey into the world of Agile. As with any voyage you have to take the first step. You may have some experience with Agile from your current or previous employment or you may be venturing out into the unknown. Either way CAT has been specifically designed to partner and guide you through all aspects of your tour.

Open Seminar in USA:
July 2–6, 2012 in Orlando, Florida

Díaz & Hilterscheid GmbH / Kurfürstendamm 179 / 10707 Berlin, Germany
Tel: +49 30 747628-0 / Fax: +49 30 747628-99
www.diazhilterscheid.de training@diazhilterscheid.de

pic: © Sergejy Galushko – Fotolia.com grahpic: © Wikimedia Commons

performance issues with just one or with several users? Are there enough computer resources to support the requested scenarios? If we have a functional or performance problem with one user, usually it should be fixed before starting performance testing with that scenario.

Even if there are extensive plans for performance testing, an iterative approach will fit better here. As soon as a new script is ready, run it. This will help to understand how well the system can handle a specific load. The results we get can help to improve plans and discover many issues early. By running tests we are learning the system and may find out that the original ideas about the system were not completely correct. A "waterfall" approach, when all scripts are created before running any multi-user test, is dangerous here: issues would be discovered later and a lot of work may need to be redone.

Assumed Activities

Usually when people talk about performance testing, they do not separate it from tuning, diagnostics, or capacity planning. "Pure" performance testing is possible only in rare cases when the system and all optimal settings are well known. Usually some tuning activities are necessary at the beginning of testing to be sure that the system is properly tuned and the results will be meaningful. In most cases, if a performance or reliability problem is found, it should be diagnosed further until it becomes clear how to handle it. Generally speaking, performance testing, tuning, diagnostics, and capacity planning are quite different processes, and excluding any of them from the test plan if they are assumed will make the plan unrealistic from the beginning.

Test Environment

Conducting functional testing in virtualized and cloud environments is quite typical and has many advantages. While many companies promote load testing in the cloud (or from the cloud), it makes sense only for certain types of load testing. For example, it should work fine if we want to test how many users the system supports, would it crash under load of X users, how many servers we need to support Y users, etc., but are not too concerned with exact numbers or variability of results (or even want to see some real-life variability).

However, it doesn't quite work for performance optimization, when we make a change in the system and want to see how it impacts performance. Testing in a virtualized or cloud environment with other tenants intrinsically has some results variability as far as we don't control other activities and, in the cloud, usually don't even know the exact hardware configuration.

So when we talk about performance optimization, we still need an isolated lab in most cases. And, if the target environment for the system is a cloud, it probably should be an isolated private cloud with hardware and software infrastructure similar to the target cloud. And we need monitoring access to underlying hardware to see how the system maps to the hardware resources and if it works as expected (for example, testing scaling out or evaluating impacts to/from other tenants – which probably should be one more kind of performance testing to do).

Time Considerations

Performance tests usually take more time than functional tests. Usually we are interested in the steady mode during load testing. It means that all users need to log in and work for some time to be sure that we see a stable pattern of performance and resource utilization. Measuring performance during transition periods can be misleading. The more users we simulate, the more time we will usually need to get into the steady mode. Moreover, some kinds of testing (reliability, for example) can require a significant amount of time – from several hours to several days or even weeks. Therefore, the number of tests that can be run per day is limited. It is especially important to consider this during tuning or diagnostics, when the number of iterations is unknown and can be large.

Simulating real users requires time, especially if it isn't just repeating actions like entering orders, but a process when some actions follow others. We can't just squeeze several days of regular work into fifteen minutes for each user. This will not be an accurate representation of the real load. It should be a slice of work, not a squeeze.

In some cases we can make the load from each user more intensive and respectively decrease the number of users to keep the total volume of work (the throughput) the same. For example, we can simulate 100 users running a small report every five minutes instead of 300 users running that report every fifteen minutes. In this case, we can speak about the ratio of simulated users to real users (1:3 for that example). This is especially useful when we need to perform a lot of tests during the tuning of the system or trying to diagnose a problem to see the results of changes quickly. Quite often that approach is used when there are license limitations.

Still "squeezing" should only be used in addition to full-scale simulation, not instead of it. Each user consumes additional resources for connections, threads, caches, etc. The exact impact depends on the system implementation, so simulation of 100 users running a small report every ten minutes doesn't guarantee that the system will support 600 users running that report every hour. Moreover, tuning for 600 users may differ significantly from tuning for 100 users. The higher the ratio between simulated and real users, the more is the need to run a test with all users to be sure that the system supports that number of users and that the system is properly tuned.

Testing Process

Three specific characteristics of load testing affect the testing process and often require closer work with development to fix problems than functional testing does. First, a reliability or performance problem often blocks further performance testing until the problem is fixed or a workaround is found. Second, usually the full setup, which often is very sophisticated, should be used to reproduce the problem. However, keeping the full setup for a long time can be expensive or even impossible. Third, debugging performance problems is a sophisticated diagnostic process that usually requires close collaboration between a performance engineer running tests and analyzing the results and a developer profiling and altering code. Special tools may be needed: many tools, such as regular debuggers and profilers, work fine in a single-user environment, but do not work in the multi-user environment due to huge performance overheads.

These three characteristics make it difficult to use an asynchronous process in load testing (which is often used in functional testing: testers look for bugs and log them into a defect tracking system, and then the defects are prioritized and independently fixed by development). What is often required is the synchronized work of performance engineering and development to fix the problems and complete performance testing.

A Systematic Approach to Changes

The tuning and diagnostic processes consist of making changes in the system and evaluating their impact on performance (or problems). It is very important to take a systematic approach to these changes. This could be, for example, the traditional approach of "one change at a time" (sometimes referred as "one factor at a time", or OFAT) or using the design of experiments (DOE) theory. "One change at a time" here does not imply changing only one variable; it can mean changing several related variables to check a particular hypothesis.

The relationship between changes in the system parameters and changes in the product behavior is usually quite complex. Any assumption based on common sense could be wrong. A system's reaction under heavy load could differ drastically of what was expected. So changing several things at once without a systematic approach will not give the understanding of how each change affects results. This could mess up the testing process and lead to incorrect conclusions. All changes and their impacts should be logged to allow rollback and further analysis.

Result Analysis

Load testing results bring much more information than just passed/failed. Even if we do not need to tune the system or diagnose a problem, we usually should consider not only transaction response times for all different transactions (usually using aggregating metrics such as average response times or percentiles), but also other metrics such as resource utilization. The systems used to log functional testing results usually don't have much to log all this information related to load testing results.

Result analysis of load testing for enterprise-level systems can be quite difficult and should be based on a good knowledge of the system and its performance requirements, and it should involve all possible sources of information: measured metrics, results of monitoring during the test, all available logs and profiling results (if available). We need information not only for all components of the system under test, but also for the load generation environment. For example, a heavy load on load generator machines can completely skew results, and the only way to know that is to monitor those machines.

There is always a variation in results of multi-user tests due to minor differences in the test environment. If the difference is large, it makes sense to analyze why and adjust tests accordingly. For example, restart the program, or even reboot the system, before each test to eliminate caching effects.

Summary

To wrap up, there are serious differences in processes and required skills between load and functional testing. Some of them were discussed in this article, but these are rather examples than a comprehensive list. It is important to understand these differences when talking about Test Centers of Excellence, regardless of specific organizational structures: while load and functional testing both are testing, trying to fit them into the same organizational structure without consideration of their specifics may be problematic.

› biography

Alexander Podelko

For the last fifteen years Alex Podelko has worked as a performance engineer and architect for several companies. Currently he is Consulting Member of Technical Staff at Oracle, responsible for performance testing and optimization of Hyperion products. Alex serves as a director for the Computer Measurement Group (CMG) http://cmg.org, a volunteer organization of performance and capacity planning professionals. He blogs at http://alexanderpodelko.com/blog and can be found on Twitter as @apodelko.