LOAD TESTING: POINTS TO PONDER

Alexander Podelko
Aetna

Testing of multi-user applications under realistic and stress loads remains the only way to ensure appropriate performance and reliability in production. The author outlines some issues to consider for performance testing of distributed business applications and presents the typical pitfalls from the practical point of view. While the original objective was to contrast load testing with functional testing, the paper touches many important points of performance testing.

Introduction

Much has been written about how to design scalable software, what best practices and design patterns to use, and even how to build models to predict performance (for example, [SMITH02] or [MICR04]). While these topics are very important to create scalable software, theories and best practices can't guarantee a required level of performance. Testing multi-user applications under realistic, as well as stress, loads remains the only way to ensure appropriate performance and reliability in production.

There are many terms to define such kinds of testing: load, performance, stress, scalability, reliability, and many others. Despite many efforts to define clear distinctions between all types of testing, none of them are widely accepted [STIR02]. One approach can be that there are no clear distinctions, because these terms describe testing from somewhat different points of view, so they are not mutually exclusive.

While goals of each kind of testing can be different, in most cases they use the same approach: applying multi-user workload to the system. We mostly use the term "load testing" further in that paper 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.

This paper outlines some issues to consider for performance testing of distributed business applications and presents the typical pitfalls from a practical point of view. The original list of topics was chosen to contrast load testing with functional testing and highlights points that are often missed by people moving into this field from functional testing (as well as development). Most performance specialists attending CMG may find some statements below to be trivial, but the paper still maybe interesting for two reasons. First, it gives a testing view of these points and, second, it lists areas that probably should be discussed if people new to performance management are running load tests.

This paper is a collection of observations, mainly related to the performance testing of distributed business applications.

Load Testing Process Overview

Load testing is emerging as an engineering discipline of its own, based on "classic" functional testing from one side, and system performance analysis from another side. The typical load testing process is depicted on figure 1 (some variations are in [BARB04], [MICR04]).

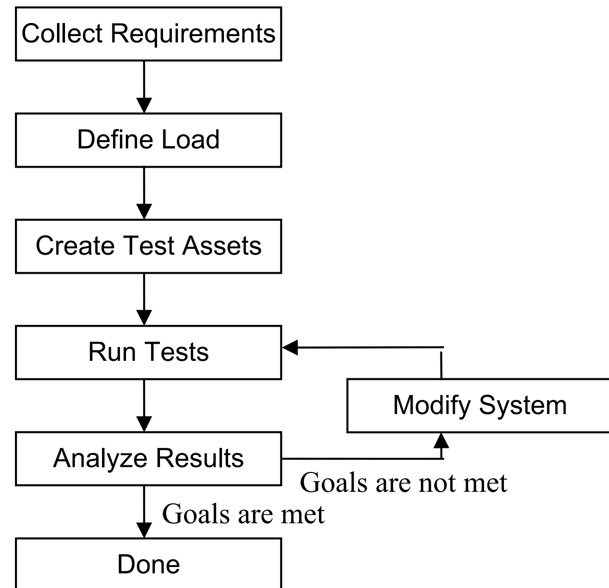

Fig.1 Load testing process

We explicitly define two different steps; "define load" and "create test assets". The "define load" step is the 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 second 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. Sometimes it is difficult to separate them. For example, performance testing of a mistuned system isn't too meaningful. Really 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 ([SMITH02], [MICR04]). 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 could have an 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 against 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 and the most probable for users to follow. It is a good idea to identify several classes of users – for example, "administrators", "operators", "users", and "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" 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 has significant 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 need to 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 include both kinds of printing in the testing scenarios.

Requirements

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

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

All classes are vital. Good throughput with long response times often is unacceptable as well as good response times for a few users.

Acceptable response times should be defined in each particular case. A response time of 30 minutes can be excellent for a big batch job, but absolutely unacceptable for getting a web page in an on-line store. Although it is often difficult to draw the line here, this is rather a common sense decision. Keep in mind that for multi-user testing we get a lot of 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 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 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 there are the same 500 users, but a 60-time difference between

loads and respectively of the hardware requirements for the system.

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 the first estimation. It is important to understand what users you are speaking about. For example, according [COGN04] for analytical reporting 10% of named (registered in the system) users are active (logged on) and 10% of active users run concurrent requests (so 1,000 named users matches 100 active users and matches 10 concurrent users). Of course, it heavily depends on the system.

Workload Implementation

If we work with a new system and never ran 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 well organized it will introduce some variation in each test, making the test less reproducible. 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 can become a very serious research task, not to mention having to deal with proprietary protocols. Creating a test harness requires more knowledge about the system (for example, an API) and some programming. 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 check that this way really works. More deeply available options are covered in [PODE05] and [LOAD06].

As soon as 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, a lack of error messages during a load test does not mean that the system worked correctly. A very 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. It can be done directly by analyzing server responses or, in cases when this is impossible, indirectly. For example, by analyzing the application log or database for the existence of particular entries.

Many tools provide some way to verify workload and check errors, but a complete understanding of what exactly is happening is necessary. For example, Mercury Interactive’s LoadRunner reports only HTTP errors for Web scripts by default (like 500 “Internal Server Error”). If we rely on the default diagnostics, we could still believe that everything is going well when we get “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.

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 the 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 get much better results than in production, or it could cause concurrency issues and we 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 to add new data during the tests without special care. Each new test would create additional data, so each test would be done with different amount of data. One way of running such tests is to restore the system to the original state after each tests. Or additional tests could be done to prove that varying amounts of data does not change the outcome of that particular test.

Exploring the System

At the beginning of a new project, it is 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 the overall performance. Modeling can help here to find the projected level of performance, 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? Is there any performance issue just with one or several users? Are there enough computer resources to support the requested scenarios? If we have a functional or performance problem with one user, it should be fixed before starting performance testing with that scenario.

Even if there are big plans for performance testing, an iterative approach fits better here. As soon as a new script is ready – run it. That gives an understanding how well the system will handle the specific load. The results we get can help to improve plans and find many issues early. By running tests we are learning the system and can 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. Issues may not be discovered until later resulting in a lot of work needing to be redone.

Unspecified Requirements

Usually when people are talking 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. Some tuning activities are usually necessary at the beginning of the testing to be sure that the system is properly tuned and the results are meaningful. In most cases, if a performance problem is found, it should be diagnosed further up to the point when it is 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 it unrealistic from the beginning.

Time

Each performance test usually takes more time than a functional test. Regularly 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 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. Considering that is especially important during tuning or diagnostics, when the number of tests to run is unknown and can be big enough.

Simulating real users requires time, especially if it isn't just repeating actions like entering orders, but some kind of process with some actions following others. We can't just squeeze several days of regular work in fifteen minutes for each user. This is not a simulation of real work. It should be a slice of work, not a squeeze.

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

Still "squeezing" should 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 each ten minutes doesn't guarantee that the system supports 600 users running that report each hour. Moreover, tuning for 600 users can differ significantly from tuning for 100 users. The larger the difference between the number of simulated and real users, the more 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.

Process

Three specific features of load testing affect the testing process and often require more close work with development to fix problems than when doing functional testing. First, a reliability or performance problem quite 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. Keeping the full setup for a long time can be expensive or even impossible. Third, debugging performance problems is a quite sophisticated diagnostic process usually requiring close collaboration between a performance engineer running tests and analyzing the results and a developer profiling and altering code. Special tools

may be necessary: many tools, like debuggers, work fine in a single-user environment, but do not work in the multi-user environment, due to huge performance overheads.

These three features make it difficult to use an asynchronous process in load testing (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.

Take 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. It could be, for example, the traditional approach of "one change at a time" (also often referred as "one factor at a time" - OFAT) or using design of experiments (DOE) theory. "One change at a time" here does not mean 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 can be wrong. A system's reaction can be quite the opposite under heavy load. So changing several things at once without a systematic approach will not give an understanding 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 usually 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 like average response times or percentiles), but also other metrics like resource utilization. Result analysis of load testing for enterprise-level systems can be quite difficult and should be based on a good working knowledge of the system and the requirements and involve all possible sources of information: measured metrics, results of monitoring during the test, all available logs, and profiling results (if available). Not only for all components of the system under test, but also for 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 is worth the effort to determine why and adjust tests accordingly. For example, restart the program, or even re-boot the system, before each test to eliminate caching effects.

References

[BARB04] S.Barber, "Beyond Performance Testing", 2004. http://www.perftestplus.com/pubs.htm

[COGN04] "Cognos ReportNet Scalability Benchmark", 2004. http://www.cognos.com/products/whitepapers/wp_reportnet_scalability_01.pdf

[JAIN91] R. Jain, "The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation and Modeling", Wiley, 1991.

[LOAD06] "Load Testing Strategy", Borland white paper, 2006. http://www.borland.com/resources/en/pdf/white_papers/load_test_whitepaper.pdf

[MCWH04] M.McWhinney, "SEI Load Test Planning Process", 2004. http://www.portata.com/seiplanningprocess.htm

[MICR04] Improving .NET Application Performance and Scalability, Microsoft Press, 2004. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnpag/html/scalenet.asp

[PODE05] A.Podelko, "Workload Generation: Does One Approach Fit All?" CMG, 2005.

[SMITH02] C.U. Smith, L.G.Williams, "Performance Solutions", Addison-Wesley, 2002.

[STIR02] S.Stirling, "Load Testing Terminology", Quality Techniques Newsletter, September 2002. http://www.soft.com/News/QTN-Online/qtnsep02.html

*All mentioned brands and trademarks are the property of their owners