What Is Specific in Load Testing?

Testing of multi-user applications under realistic and stress loads is really the only way to ensure appropriate performance and reliability in production. Load testing is emerging as an engineering discipline of its own, based on "classic" functional testing from one side, and system performance analysis and capacity planning from another side. The terminology is still vague in this field and the borders are fuzzy. From a practical point of view, we probably can break down all kinds of testing into two big classes: with a multi-user load (load, performance, stress, volume, etc.) or without it (functional, regression, etc.). The different terms inside each of these classes specify why we are testing and for what result we are looking rather than how we do it. There is no precise definition for each term; the exact meaning can differ from source to source. For example, performance testing could mean that we are more interested in response time; load testing could mean that we want to see the system behavior under a specified load, and stress testing could mean that we want to find the system break point. We still do the same things: apply a multi-user load and get some measurements. The difference is only in the details: what precise load we apply and what measurements are more important to us.

As all these terms reflect goals of testing, these two classes do not match the terms completely: we can do performance testing for one user measuring response times with a stopwatch, or test the functionality of the system while having a 100-user load in the background. Moreover, performance is a part of functionality in some sense. We would still refer to testing under multi-user workload as load or performance testing here and contrast it to classical one-user functional testing.

Both classes are considered testing and have a lot in common. Still, load testing has significant specifics and requires some special approaches and skills. Quite often, applying the best practices and metrics of functional testing to load testing results in disappointments. It looks like many things that look trivial for an experienced performance engineer could drop from the attention of a person less experienced in this field that quite often results in unrealistic expectations, not optimal test planning and design, and misleading results.

Here we try to outline some issues to be taken into consideration for performance testing and point to typical pitfalls from the practical point of view. The list was chosen to contrast load and functional testing; the things in common for both of them are not discussed here. Although most of the recommendations are still valid for functional testing (as mentioned, even the border between them is somewhat fuzzy), they are much more important for performance testing. The selection is based on the extensive experience of Hyperion's Performance Engineering group, multiple discussions with experts in load testing, and best papers and books about Performance Engineering.

The list is illustrated by the example of Hyperion Analyzer project. Hyperion Analyzer is a powerful interactive query and reporting tool. Analyzer allows analyzing sophisticated multidimensional and relational data in an easy-to-use graphical interface. There are four main components relevant to the project:

- *The central part is a standard JSP-based J2EE application, the Analysis Server. Runs inside an Application Server (can be TomCat, WebLogic, or WebSphere).*
- *Relational repository to keep all information (can be SQL Server, Oracle or DB2).*
- *Data source (can be Hyperion Essbase, Hyperion Financial Management, or relational data).*
- *The Java client – a very powerful applet making a lot of processing on the client side.*

There are also other components (Thin HTML client, Windows client, Administrator tools, API).

The project consisted of performance testing of Hyperion Analyzer against multidimensional database Hyperion Essbase for a large financial company. WebSphere (including vertical and horizontal clustering) was used as an Application Server and Microsoft SQL Server was used as a repository.

Workload Implementation – If you work with a new system (i.e. you never ran a load test against it before), the first question (as soon as you get some kind of testing requirements) is how you will create load. Are you going to generate it manually, use an available load-testing tool, or create a test harness?

Manual testing could sometimes works if you want to simulate a small number of users, but even if well organized, it will probably introduce more variation in each test and make 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 could 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.

As soon as you decided how to create the workload, you need to find a way to verify that the workload is really being applied.

Hyperion Analyzer was completely re-designed in version 6. Version 5 was a client-server application; version 6 is a standard J2EE three-tier application. The Analyzer Java client is a very powerful applet communication with the Analyzer server using HTTP. The server response came in the body of HTTP requests as a serialized Java object including session ID.

The first problem was to record the communication – the applet is starting in a separate window and LoadRunner, recording communication between the Internet explorer and the server, created almost empty scripts. Using multiprotocol recording allowed to solve the problem.

Efforts to record and playback the HTTP communication still failed - it didn't work without session ID parameterization. The existing API was too fat – the effort to make a test harness failed due to contention on the client side even with the limited number of users. Finally

Mercury provided us with a utility to convert the script from C to Java and after that, using the Analyzer API calls, we de-serialize the object and extract the session ID.

Workload verification – Unfortunately, a lack of error messages during a load test does not mean that system works correctly. A very important part of load testing is workload verification – you should be sure that the applied workload is doing what it is supposed to do -- that all errors are caught and logged. It can be done directly (analyzing server responses) or, in cases when this is impossible, indirectly (for example, analyzing the application log 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 you rely on default diagnostics, you could still believe that everything is going well when you get "out of memory" errors instead of requested reports. To catch such kind of errors you should add the special command to check the content of HTML pages returned by the server to your script and enable such kind of checks in run-time options.

As far as we get serialized Java objects from the Analyzer server we haven't any easy way to check that it worked correctly. De-serializing and analysis of each object require a lot of work and put significant overheads on the client (LoadRunner). Finally, we ended up using a utility to analyze Essbase log. It calculates the number of queries processed by Essbase. If this number matches what should be, we can suppose that it works.

Unspecified requirements – Usually when people are talking about performance (load) testing, they do not make distinctions among tuning, diagnostics, or capacity planning. Pure "performance testing", just getting performance metrics, unfortunately is possible only in rare cases like benchmarking, when the system and all optimal settings are well known. Therefore, some tuning activities are 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 (or reliability, etc.) problem were found, it should be diagnosed further up to the point when it would be clear how to handle it. Generally speaking, "performance testing", "tuning", "diagnostics", and "capacity planning" are quite different processes and not including explicitly any of them (if they are really assumed) in the test plan would make it unrealistic from the start.

The Analyzer project was formulated as quick performance check, but we still return to it from time to time until now, after almost 2 years. Necessity to tune application server parameters, what-if analysis (especially concerning WebSphere clusters), tracing down and fixing found problems took much more time than testing itself.

Exploring the System – At the beginning of a new project, it is good 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 ways to predict how many users the system can support and how each scenario will affect overall performance. Do we have any functional problems - can we do all

requested scenarios manually? Will we run into any problem with several users? Do we have enough computer resources to support the requested scenarios?

The customer did some performance testing in parallel (it was quite big and sophisticated deployment). They called and told that Analyzer hangs with database X. When we came on-site and ran just one user monitoring all involved servers we found that running a single report takes two minutes and completely occupies two processors (from 4) on the Essbase (database) server. Starting even 10 users completely clogged the system and created an impression that Analyzer hangs, but really the database was overloaded. The problem was the bad database design and any further load testing had no sense until the design was changed.

Time – Each performance test takes some time, often significantly more than a functional test. Generally, we are interested in the steady mode during load testing, i.e. when all requested users are in the system. It means that we need time for all users to login and work for a meaningful period to be sure that we are in the steady mode. Moreover, some kinds of testing (reliability, for example) could require significant amount of time – from several hours to several days. Therefore, the number of tests that could be run per day is limited. This could be especially important during tuning or diagnostics, when the number of iterations is unknown.

The customer specified very complicated user scenarios. They wanted to include almost each their report into the scenario. Even after significant simplification a run took 2.5 hours. As soon as we started to tune WebSphere parameters, it became a real disaster. So we end up in simplified version to run during tuning.

Take a systematic approach to changes – The tuning (and often diagnostic) process consists 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 paradoxical under heavy load. So changing several things at once without a systematic approach will not give you an understanding how each change affects results and could mess up the testing process and lead to incorrect conclusions. Of course, all changes and their impacts should be logged to allow rollback and further analysis.

We needed to move tests to another set of equipment (as far as they took much more time and we needed to return machines we used). We also get advice from IBM about WebSphere tuning and "slightly modified" repository from the customer. Due to urgency, we implemented everything on the new equipment and got much worse results. We spent a lot of

time backing up configuration setting and analyzing hardware performance until figured out that the problem was due to increased number of users in the repository. Making one of this three changes in a time, although required a little more time, finally would saved us significantly more time.

Data – The size and structure of data usually affect load test results dramatically. Using a small sample set of data for performance tests is an easy way to get misleading results. There is no way to predict how the data size affects performance before real testing. The closer the test data is to production data, the better (although testing with larger data sets makes a lot of sense to ensure that the system will work when more data had been accumulated).

Running multiple users hitting the same set of data is an easy way to get misleading results. This data, for example, can be completely cached and you get much better results than in production. However, it can cause concurrency issues and you will get much worse results than in production. Scripts and test harnesses usually should be parameterized so that each user uses the proper set of data ("proper" could be different depending on the test requirements).

Another easy trap with data is to add new data during the tests without special care. Each time it would be different tests with a different amount of data in the system. The proper way of running such tests is to restore the system to the original state after each test (or make additional tests to prove that it does not matter in this particular case).

Further analysis of the problem with the increased number of users in the repository using SQL server profiler pinpointed a SQL statement making outer join of three tables and the number of records in each table was directly proportional to the number of users. When there were 200 users in the repository, it worked fine. As soon as the number of users was increased to 800, this SQL statement became the main bottleneck.

Process – Three specific features of load testing affect the testing process and often require more close work with development to fix problems than in functional testing. First, quite often, a reliability or performance problem blocks further performance testing until the problem is fixed or a workaround is found. Second, usually the full setup "as is" (that often is very sophisticated) should be used to reproduce the problem. Keeping the full setup for a long time could 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). Many tools (like debuggers) work fine in a one-user environment, but do not work in the multi-user environment (for example, due to huge performance overheads).

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

Returning to the SQL statement problem with three outer join, we weren't able to test the full customer setup until the problem was fixed. A workaround was to remove most of users to test performance of other functionality. Although the problem was fully diagnosed, it took some time to communicate it. It was difficult to explain until we communicate it to the development manager and the developer responsible for this code.

Combinatory explosion – Even in functional testing, we have an unlimited number of test cases and the art of testing is to choose a limited set of test cases that could check 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 how the steps of one user relate to the steps of another user during the test could affect results drastically. So generally, it is the same art, but you have significantly more choices and significantly less efforts to check them (see time note above). Load testing cannot be comprehensive, so you should choose a subset of tests limited by your resources that would provide you with maximal amount of information (criteria can be quite different – typical scenarios, risk minimization, etc.).

We ended up with a few "realistic" scenarios (running existing reports and creating ad-hoc reports) that covers a small part of functionality but still allow to pinpoint performance problems and get understanding of performance limits. In parallel, we investigated a separate performance problems reported by the customer.

Result analysis – Usually test results of load testing are difficult to interpret as passed/failed. Even if we do not need to tune/diagnose anything, we usually should consider not only transaction response times for all different transactions (usually using aggregating metrics like average response time or 90% percentile), but also other metrics like resource utilization. Result analysis of load testing for enterprise-level systems could be quite difficult and should be based on 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). For example, heavy load on load generator machines could 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 this difference is large, it is worth checking why and adjusting tests accordingly – for example, re-start the program (or even re-boot the system) before each run to eliminate caching effects. If the system is overloaded, the difference could be very significant.

The SQL statement problem with three outer join was really found during the result analysis. We noticed that the pattern of load changed: before the main load was on the application server with 10-15% of cpu utilization on the database server, then it became vice versa – heavy load on the database server and moderate load on the application server. So attention was moved to the database server and the problem was soon pinpointed.

As was mentioned in discussion about workload verification, result analysis for Analyzer included the analysis of the Essbase log that each query was really processed. Other data

included results from LoadRunner (with statistical information) and monitoring results from all tiers: the application server, the database server (repository), the HTTP server, the Essbase server (data source) as well as client machines with LoadRunner (we make additional operations on this client machine with parsing and processing of server responses so need to keep an eye on it).

The above content is not instructions to follow literally, just some things you should keep in mind while conducting performance testing. It is possibly to cut corners in many particular cases, but it would probably be good to check your plan against the suggestions above and explain to yourself why you doing it in a different way.