Load and Stress Testing

Mark Kampe
$Id: loadstress.html 7 2007-08-26 19:52:08Z Mark $

1. Introduction

Load and Stress Testing is very different from most other testing activities.

For most products, the vast majority of all tests exercise positive functional assertions (if situation x, the program will y). Such assertions may describe either positive (functionality) or negative (error handling) behavior. A typical suite is collection of test cases, each of which has the general form:

- establish conditions (c_1, c_2, ... c_n)
- perform operations (o_1, o_2, ... o_n)
- ascertain that assertions (a_1, a_2, ... a_n) are satisfied.

The art of creating such a test suite is being able to define a set of test cases that simultaneously:

a. adequately exercises the program's capabilities
b. adequately captures and verifies the program's behavior
c. is small enough to be practically implementable

If all operations are performed, and all of the required assertions were satisfied, the program has passed the test. Such "functional validation" suites form the foundation for automated software testing, and are the primary basis for determining whether or not a product "works". As such, repeatability of results is considered to be very important.

Load and stress testing are quite different:

- the number of enumerated test cases is relatively small, and the particulars of each test case may be particularly important.
- we will run these test cases in pseudo-random orders for unspecified periods of time.
- we have no expectation that results will be repeatable (we are, in fact, depending on this).
- in many cases, there is no definitive pass indication. Rather, we can only say that the test has run for a period of time.
- we may not take the trouble to define a complete set of assertions to determine the correctness with which any particular operation has been performed. In some cases we may not even look at returned results.

This note is a brief introduction to the goals and methods of load and stress testing.

2. Load Testing

The initial (and perhaps still primary) purpose of load testing is to measure the ability of a system to provide service at a specified load level. A test harness issues requests to the component under test at a specified rate (the offered load), and collects performance data. The collected performance data depends on what is to be measured, but typically includes things like:

- response time for each request
- aggregate throughput
- CPU time and utilization

- disk I/O operations and utilization
- network packets and utilization

The resulting information can be used to:

- measure the system's speed and capacity
- analyze bottlenecks to enable improvements

The key ingredient in this process is a tool to generate the test traffic. Such tools are called **load generators**.

2.1 Load Generation

A load generator is a system that can generate test traffic, corresponding to a specified profile, at a calibrated rate. This traffic will be used to drive the system that is being measured. Such testing is normally performed on a "whole system", and is typically performed through the system's primary service interfaces:

- If the system to be tested is a network server, the load generator will pretend to be numerous clients, sending requests over a network.
- If the system to be tested is an application server, the load generator will create multiple test tasks to be run.
- If the system to be tested is an I/O device, the load generator will generate I/O requests.

In all cases, the test load is broadly characterized in terms of:

- request rate
  the number of operations per second
- request mix
  Different types of clients use a system in different ways. A database might do a large number of small (and relatively random) disk reads and writes, while an streaming video server would do a much smaller number of huge contiguous transfers, all reads. If different types of requests exercise very different code paths, it is important that the load generator be able to accurately emulate all of the various types of clients.
- sequence fidelity
  In many situations, it is sufficient to merely generate the right mix of read and write operations. In other cases it may be critical to simulate particular access patterns (timed sequences operations on related objects). In some situations it may be necessary to simulate realistic scenarios against predetermined or random objects.

A good load generator will be tunable in terms of both the overall request rate and the mix of operations that is generated. Some load generators may merely generate random requests according to a distribution. Others may have rule grammars that describe typical scenarios, and use these to generate complex realistic usage scenarios.

Most such load generators are proprietary tools, developed and maintained by the organizations that build the products they measure. Some have been turned into products (or open source tools) and are widely used. Some have become so widely used that they have been adopted as "standard performance benchmarks".

2.2 Performance Measurement

There are a few typical ways to use a load generator for performance assessment:

1. Deliver requests at a specified rate and measure the response time.
2. Deliver requests at increasing rates until a maximum throughput is reached.
3. Deliver requests at a rate, and use this as a calibrated background for measuring the performance other system services.

4. Deliver requests at a rate, and use this as a test load for detailed studies performance bottlenecks.

In the first two usages, the load generator is a measurement tool. In the other usages, it provides a calibrated activity mix to exercise the system.

2.3 Accelerated Aging

Many errors (e.g. memory leaks) can have trivial consequences, and only accumulate to measurable effects after long periods of time. Load generators can be used to create realistic traffic to simulate normal traffic for long periods of time. Alternatively, they can be cranked up to much higher rates in order to simulate accelerated aging. It is common for test plans to require products to undergo (at least) months of continuous load testing to look for the accumulation of such problems.

3. Stress Testing

Load generators are (fundamentally) intended to generate request sequences that are representative of what the measured system will experience in the field. Accelerated Aging uses cranked up load generators to simulate longer periods of use. If we go further (into simulating scenarios far worse than anything that is ever expected to really happen) we enter the realm of stress testing.

Any error that results from a simple situation is likely to be easily recognized, tested, and (therefore) properly handled. In mature and well tested products, the residual errors tend to involve combinations of unlikely circumstances. These problems are much easier to find and fix in design review than they are to debug, but we still need to test for them. But how can we test for combinations of circumstances that we can't even enumerate?

The answer is random stress testing.

- use randomly generated complex usage scenarios, to increase the likelihood of encountering unlikely combinations of operations.
- deliberately generate large numbers conflicting requests (e.g. multiple clients trying to update the same file).
- introduce a wide range randomly generated errors, and simulated resource exhaustions, so that the system is continuously experiencing and recovering from errors.
- introduce wide swings in load, and regular overload situations.

Such testing takes situations that might normally occur only once or twice per year, and makes them occur (in combinations) hundreds of times per minute. Take a large number of such systems, and run them in this mode for several months. **This** will give us some serious confidence about the robustness and stability of our systems. Such testing is extremely demanding, and (in fact) very few software products receive (or survive) such testing. This is, however, typical methodology for mission critical and highly available products.

4. Conclusions

In the early stages of the product, most of the bugs are found by reviews and functional validation tests. These remain valuable (as regression tools) throughout the life of the product, but they are not actually expected to find many more bugs once they have initially succeeded.

Once a product has passed this "adolescent" stage, most of the bug reports result from new usage scenarios associated with adoption by real customers. There is a wider diversity of use here, and it may take a while so shake out all of these problems. But again, once the product has been brought up to the demands of every-day customer use, the number of bug reports resulting from this falls off sharply.

If we ignore new features (which are, in some sense, new software), the improvements in mature software tend to be in performance and robustness. The tools and techniques for finding functionality problems are neither designed nor adequate for driving improvements in these areas. Load and stress testing are very different from functional testing. The goals are different, the tools are different, the techniques are different, and the testing programs are different.

Functional quality starts with good design, and is then complemented by good review, and secured by a complete and well considered set of functional test suites. Performance and robustness also start with good design, are complemented by review, and secured by testing. Functionality tools and testing are, for the most part, done when the product ships. Load and stress testing will continue to be a critical element of product quality throughout its lifetime, and (unlike functional test cases) the load and stress testing tools may receive almost as much design and development attention as the tested product does.

In mature products, the difference between a good product and a great one is often found it the seriousness of the ongoing performance and robustness programs. **Performance** and **availability** don't just happen. They must be **earned**.