The Workshop on Performance and Reliability 11: Reliability…what can we do?

Reliability Testing: Kill the Server!

Document Version 0.8.6

Last Updated: October 24th, 2008

Eric Proegler

ericp@onbase.com

Performance Team Lead

Hyland Software, Inc.

28500 Clemens Road

Westlake, Ohio 44145

Ph: (440) 788-5000

Fax: (440) 788-5100

www.onbase.com

COPYRIGHT AND DISCLAIMERS

Intended Audience

Attendees of the Workshop on Performance and Reliability, or others examining performance testing from a context-driven perspective. It is expected that the reader has experience and familiarity with using automated testing tools test reliability of software systems.

Abstract

This paper describes reliability testing techniques that have been useful for identifying server vulnerabilities in a highly agile context. By focusing on what can be done immediately with available tools, these techniques may be helpful when time for measuring a system's reliability is limited.

Table of Contents

Original Call for Papers (CFP)

This is the request for contrribution that prompted the writing of this paper, for distribution in October 2008 at WOPR 11 in San Diego. It appeared on the WOPR web site, http://www.performance-workshop.org.

> The IEEE defines reliability as "The ability of a system or component to perform its required functions under stated conditions for a specified period of time." Mean Time Between Failure is often described as a measure of reliability. The next question that follows - How is that measure to be used or applied, such that it has meaning?
>
> In the space of computing, 'reliable' is an adjective often with ambiguous meaning, because the term can be applied in several ways. Nearly every driver in the world has a sense of whether their car is reliable. Factors such as how the motor sounds, the warning-light panel, how well it drove the last trip, even the smell all contribute to the sense of reliability. However, what metrics can be applied to understand just how reliable a car is? More importantly, is it possible to identify metrics applicable to the computing space? We're looking for people with experience and passion to contribute papers & experience reports based on actual first person experiences that address one or more of the following aspects of reliability…
>
> - Requirements – How are they stated? What makes a good reliability requirement?
> - Modeling – Do you create mathematical models that try to predict reliability? How accurate are the models?
> - Risk Management – How reliable does a given system need to be? How to identify risks?
> - Testing – How do we go about testing the reliability of a system?
> - Verification vs. Validation
> - Field Experience – What happened post-ship?
> - Overall Design – Are there design or test patterns that dramatically improve (or drastically lower!) reliability?
> - Reliability Engineering in the Real World – how is it applied? Are measures like MTBF and MTTR useful outside of marketing literature? Are there useful alternatives?
>
> WOPR11 is seeking experience reports (ERs) of your relevant experiences and innovations from past projects and from your current initiatives (as-yet unfinished and unproven projects). For a description and samples of ERs, see the Paper Guidance and Case Studies & Papers pages.
>
> We are more interested in effective presentations and enlightening exchanges than in formal papers. A detailed paper is welcome though not required. For your presentation, an organized outline is enough. We are looking for informative, in-depth storytelling by experienced practitioners. Your proposal to present should contain enough substance for us to understand and evaluate it. Content is more important than format. Your presentation should omit any confidential data (anything that requires an NDA).
>
> Reports and presentations are welcome over a broad range of topics related to performance testing. The test domain is broad and may include real-time embedded devices, web sites, and international telecom networks.

Narrative

I lead a small test team of three people in a software development company. In our aggressively agile shop, more than one hundred developers and seventy-plus functional testers are constantly adding and refining functionality. It is difficult to add value to the work of this many people, and to provide timely information for a project moving this fast. Information we gather quickly becomes stale because of the development pace.

Our testing projects have timelines of hours, not weeks. Traditional models of a performance testing do not fit well here, not just because of the effort necessary to reach expected levels of thoroughness and rigor, but also because the breadth of the product means that we cannot test all of it. Our focus on test design falls on acquiring the most useful data as fast as possible; we need to achieve a high yield of testing for the "worst" types of problems, such as stability issues encountered on probably code paths.

It helps that we have only development stakeholders, which means that the kinds of tests we consider valuable can be very different from what acceptance testers might try. Even in different contexts, the techniques we've developed can be useful for quickly getting information.

In our context, we work with a sizeable functional testing group that finds defects. Additional background on how and where is in the "More on Context" section of this paper.

The definitions I've found for reliability testing tended to be similar:

- "*software is subjected to the same statistical distribution of inputs that is expected in operation. Instead of actively looking for failures, the tester in this case waits for failures to surface spontaneously, so to speak*." (1)
- "*The ability of a system or component to perform its required functions under stated conditions for a specified period of time*." (2)
- "*software unreliability is the result of unanticipated results of software operations*." (3)

All of these definitions focus on defect densities as measures of reliability. For us, reliability is not a measure or proportion of the number of software defects, but another category of testing that is seeking to alleviate a different set of risks than typical functional testing.

Incorrect software response is not being reliable, but in the context of a web application or other multi-user system, states where the system becomes unstable and affects all users of it are more significant reliability risks. A shared system like one of these should have a much higher reliability requirement than an individual execution that is only one user's experience. Can we take the server down?

Our testing with respect to reliability focuses on faults caused and/or detectable by load. The most common are thread safety issues and object/memory leaks; these are the risks focused on for this paper. We are sometimes checking for these in the context of software performance, and sometimes for themselves.

Web server testing with a load tool requires creation of load scripts and data pools. We get reuse out of these for reliability testing purposes by simply changing the load profile. Some of the techniques that use our automated load tools to measure reliability are described in the Test Techniques section of this paper, but in short, with unrealistic load profiles, we challenge the software's reliability.

Following are the techniques that focus on leveraging existing load test scripts for measuring reliability of a web server software product. They function by stretching or compressing one or more load scenario characteristics, such as length of test, number of threads, or length of sleep delays.

If the software continues to execute under these types of loads without errors, then they have been evaluated for reliability while performing the functionality in the test. Since this functionality usually includes many common code paths, some sense of reliability can be had, "minefield style" or no.

Longevity Test

This test attempts to complete a large number of iterations over an extended period of time. This is often called a "soak test", but usually does not run this long. This test does not particularly challenge thread management, but it does challenge memory management and is a way to check for memory leaks.

This is implemented by dedicating a server for our web server product, and installing a database and file store on the same server. Though these are typically distributed in the field, any communications disruptions could interfere with the actual software test. A client machine running a load tool is set up next to it.

The load tool is configured to exercise the server at a relatively low rate; our current run has 100 threads executing transactions system-wide at a rate of 0.49 transactions/sec, or at about 204 seconds/thread. This is a fairly "realistic" rate, if not overall model. After 72 days, 3,041,100 transactions have been completed. We'll likely stop soon to update the software to a newer build, and may increase the rate.

A memory leak of even a few bytes could be detectable after this many iterations, not only in measurement of consumption, but in any reliability effects typically expected from a memory management problem. It turns out that we need to use our custom load tool for this kind of test; our commercial tool leaks memory and can't stay up this long.

Slam Test

This test throws loads at the web server with no user sleep time or delays. This test was initially requested by developers that wanted a quantity to measure for regression purposes. A workload requested at the start is 10 threads executing 10 transactions each. These might complete in 25 seconds on typical hardware for a simple login, ((do something and then something) x10), logout loop. This is a number of iterations that lends itself to comparing and contrasting with other measurements.

This type of test has value in testing a server's response time for completing a given quantity of work, and in deliberately stressing the software for reliability. Thread safety issues that are load-based can be flushed out by this kind of test. If execution time increases, then the software is probably doing more work, suggested a decline in efficiency. Queuing effects and other variables could be in play, but those should be discovered.

Our testing experience has shown that we can find issues this way: additional database access, memory consumption, etc. The known quantity helps in analyzing the number of times something happened.

A variation on this technique is sometimes used for determining a maximum throughput, with similar second-order reliability testing effects. A small number of threads (say, 4) are executed without sleep time, and then one thread is added at intervals until transactional throughput per second peaks. Again, by pushing the software hard, certain risks are chased out.

Fill 'Er Up Test

This test was conceived as a way of exploring the process space limitations of the IIS worker process. A 32-bit application generally is capable of addressing 2.15 GB of memory. Our experience with our web

server process has been that this limitation will be encountered by virtual memory several hundred megabytes before physical consumption nears this level.

This was especially interesting to us because it represents the hard per-server limit for how many users we can support on a server. CPU usage for rendering business content is usually the operational limitation in scaling a web server, but for some research usage (say, examining supporting documents as part of a business transaction), CPU usage may not be the limitation. The highly affordable nature of quad-core processors, and the six and twelve core processors soon coming to market, suggest that CPU will be less of a limiting factor going forward.

It also was desired to measure memory on a per-session basis. Because of the managed code (.NET) on the web server, it can be difficult to accurately measure this. A significant number of users would need to be added to the system to make any accurate measurements. The Horde tool (see Tools in More on Context) does not have the licensing handcuffs of the commercial tool, so we use it to add loads up to the breaking point, up to and above 1500.

By examining memory and how it is/is not recovered across this many sessions, we have learned a lot about memory utilization and management. Some degree of precision is achievable in measuring per-session memory usage. Dogma about how garbage collection works can be tested in a well-understood memory usage context. By testing this way, leaked objects and memory can be detected.

More on Context

I work for a software development firm that develops an ECM solution named OnBase, developed in C++/C#/ASP.Net, using Windows and IIS with a variety of database and storage platforms. Development/quality assurance staff is around 200. Development is aggressive Agile, with a sizable number of features added between semi-annual major releases, with service packs being distributed year-round.

Approximately 7500 customers have been installed by the company's services group, channel partners, and OEM re-labelers. Imaging, parsing and import processing, workflow, a thick client, a web server and web services, and exposed APIs are all components of the software of concern to us. Database/web/application servers, storage hardware, networks, custom middleware, and widely varying line of business uses are the context in deployment.

I lead a team of three that serves a number of purposes. We report to a senior manager with 35 years of experience, an essential architect of the software that now serves as the CTO's lead technical resource, guiding us and others. For now, I call us the Performance Team, though I'm still taking impressive name suggestions.

We performance test software under development; we work with Development to isolate and fix bugs in areas such as thread safety, in addition to identifying unnecessary resource allocation and consumption in the software. We analyze software's effects on resources directly; poring over SQL traces and FileMon logs are typical activities. We also provide troubleshooting and analysis in escalated support situations (frequently leading to the previous activity), evaluate technologies and their deployments with our software, present training to sales engineers, teach at our vendor conferences, and occasionally earn services revenue through performance testing services.

This broad portfolio requires executing quickly and simply. We must choose a reasonable scope and stick to it. We generally first perform a small number of uncomplicated experiments with thorough instrumentation. We then apply rapid heuristics and group analysis, report, and then test further into areas of interest. We report written, orally, and/or through change requests. Methodology and conclusions may be challenged, so we need to show data, but documentation is generally quite light and often not more than email and a couple of spreadsheets.

This pace is kept successfully because we have an accumulated foundation of knowledge concerning the software, supporting technologies and test lab hardware. Mentorship is readily available, intermediate levels of confidence in results can be expressed, mistakes are tolerated, and we are permitted to simply state conclusions and move quickly to the next task.

We definitely make trade-offs for this agility. We load test only some of the time, reducing our opportunity to become truly familiar with our tools. We do not have resources and time to invest in making our toolbox as robust and repeatable as it could be. The commercial load tool experience we have is significantly affected by this. We do not reach very high levels of justified confidence in our results, as we generally do not have the time to aggressively challenge conclusions from more than one or two directions. We live the 80/20 proportion, counting on getting enough right quickly enough to make up for the overall success percentage reflected in moving at this speed.

Consider a classic response time graph; response time is measured as load increases. Depending on the limitations of the system and how these relate, the curve upwards in the response time may be linear or exponential. Greatest efficiency for the system under test is achieved at the knee of this curve. Testing value (confidence) versus testing effort might be graphed in a similar fashion. I suggest this curve is logarithmic; though increasing effort may continue to yield value, the rate at which value is added as a test is refined declines. Our testing tries to stay on the steep part of this curve.

All in all, the fast pace and diverse focus makes for a high level of challenge and freshness, and fits our team very well. The frequent changes of priority, lack of closure, and uncertainty means that it would probably not be a good fit for everyone.

Tools

For analysis, we use whatever tools we can find for logging and aggregation. For creating and driving load, we most commonly use a commercial tool named QALoad by Compuware, followed by a homegrown product we call Horde. It has been developed over a couple of years, and is nearing a level of maturity where it can become our primary load tool.

References

1. Testing Standard Working Party. (undated). *Reliability Guidelines*. Retrieved October 20, 2008, from http://www.Testingstandards.co.uk: http://www.testingstandards.co.uk/reliability_guidelines.htm
2. Rosenberg, L., Hammer, T., Shaw, J. (Nov 1998). *Software Metrics and Reliability, citing* IEEE 610.12-1990. Retrieved October 20, 2008, from http://satc.gsfc.nasa.gov: http://satc.gsfc.nasa.gov/support/ISSRE_NOV98/software_metrics_and_reliability.html.
3. Wikipedia Authors (October 20, 2008). *Reliability Engineering*. Retrieved October 20, 2008 from http://en.wikipedia.org: http://en.wikipedia.org/wiki/Reliability_engineering#Software_reliability