FUNDAMENTALS OF PERFORMANCE TESTING

Ross Collard, Collard & Company

The Problem We are Trying to Solve

Today's personal computers run a thousand times faster than IBM's first personal computer. Network speeds have increased even more, and database storage costs have dropped by a factor of more than ten thousand. System performance engineering and software optimization have advanced from a few incomplete rules of thumb to respectable professional fields. So why bother to test system performance and robustness?

Our story begins … You have made a heroic effort to test a system under unreasonable deadlines and with a limited test staff and equipment. You scrupulously ensure the features work as expected, and then release the system. A few days later, you receive a call from a senior user. You are expecting words of appreciation, but he can only moan about what you have done to him.

Our story ends on a sad note. The features do work, but the live response time is way too slow and the throughput is too low. Or the system cannot handle peak loads. The system cannot recover from routine errors, and seems to crash whenever someone blinks. Or the system does not work on all the users' platforms and configurations. The system worked fine in the test lab, but does not scale up very well. Or its resource utilization is prohibitive.

If you identify with this story -- you have "been there, done that" -- and would prefer not to re-live the experience, then this book is for you. Or if you have never been there and want to keep it that way, this book is for you too.

Performance Testing Issues

This book examines the major issues of performance testing, such as:

1. Determining the right mix of demands to place on the system during performance evaluation.

2. Determining what and how to measure.

3. Establishing the test environment, including scalability from the test lab to the live operational environment.

4. Evaluation and selection of load testing tools.

5. Interpretation of performance measurements and test results.

6. The relationship between performance testing and system tuning.

7. How to determine if the system performance and robustness requirements are realistic and testable.

8. How the testers can provide early feedback about likely performance bottlenecks and robustness vulnerabilities during the system design and development, instead of at the end of the project.

The Performance Testing Track Record

The majority of performance and robustness testing projects fail. Many are not worth the time and cost, and in the worst case their results are dangerously misleading. Testing often does not predict the live system's behavior within a tolerable margin of error. Conclusion: testing has a better record in heading off catastrophic implementation decisions, by detecting under-performance in the test lab.

Making precise predictions is a mistake. The critical outcome of testing is a one-line recommendation: "Yes, go live" or "No, don't". The objective of performance testing is to predict a system's performance in live operation, and in a timely fashion so that performance problems can be fixed before the system goes live. We can make a similar point about testing a system's robustness.

By these definitions, in my observation the majority of performance and robustness testing projects fail. Many are not worth the time and cost, and in the worst case their results are dangerously misleading. In these projects, the performance testing does not predict the live system's performance, such as its response time and throughput, etc., within a tolerable margin of error.

Performance testing has a better track record in heading off catastrophic system implementation decisions, by detecting gross under-performance in the test lab. Perhaps the lack of precise predictions of performance is less important than avoiding live performance problems.

Robustness testing is often not done as an activity in itself, with its own center (i.e., test objectives, project staffing, etc.) Instead, oftentimes robustness is a side issue of other types of testing and is decentralized among various people who are focusing primarily on other interests, such as feature correctness, database integrity, and so on.

Informal performance measurement and byproduct robustness testing have value, but this value is highest when the system we're testing is built like a house of cards – no one needs subtle methods to test its limits. Various wits have named this approach "Gone with the Wind" or "Three Little Pigs" testing, as in:

"I'll huff and I'll puff and I'll blow your house down." We do not need to measure precisely how many meters the wind carry the house, it is sufficient to observe that the pig is without protection.

We will explore the reasons why these types of testing fail, and what can be done to avoid failure, in this book. In the words of Nobel prize winner Neils Bohr, prediction is difficult, especially about the future.

The Status of Performance and Robustness Management

It is helpful to consider testing within the context of running and using a system. Managing a system's performance and robustness encompasses system design, system integration, resource sizing, capacity planning, monitoring the live operation, and system tuning, as well as testing.

Over time, the performance, robustness and cost-effectiveness of systems have unquestionably and radically improved. Hardware costs have dropped and continue to drop dramatically, and software prices also are declining as software increasingly becomes a commodity.

So the need to manage performance and robustness should be declining. Who needs performance management when the equipment costs are approaching zero? We can just throw more "iron" at our performance problems.

Actually, we are seeing the opposite trend – a sizeable and continuing upsurge in the demand for performance and robustness management. One reason is the voracious growth in the demand for services, which shows little sign of slowing in the next five to ten years. Another reason: many software packages and components suffer from creeping feature-itis and become "bloatware", exhibiting worse performance with each new release.

Probably the major reason for this upsurge is the complex and fluid nature of today's multi-technology, multi-vendor distributed systems:

o System integration has become a major field in its own right, and performance and robustness are not easy to predict when we bring products from different vendors together.

o Every system is unique in the galaxy – nobody else has compiled quite the same mix of hardware, networks, databases, support software and applications. Like it or not, this is like Star Trek, "going where no person has gone before".

o We do not have good conceptual models with which to understand how subsystems interact, interfere and otherwise behave within systems of even moderate complexity. The traditional, centralized approach to managing a single-platform system like a mainframe does not work very well in decentralized

environments.

o Vendors test their own products (maybe), but there is little or no integration testing across vendors.

o The skills needed to monitor and tune systems are fragmented, which each group of specialists only seeing part of the picture. When tuning decisions are made at the subsystem level and are not well coordinated, it is easy to de-tune a system and not know it.

o The tools available for monitoring and tuning systems are equally fragmented. Usually no one tool can tell us everything we need to know to manage the system, and the readings available from the different tools are difficult to correlate into a coherent picture. (Despite some tool vendors' wonderful claims.)

o The knowledge base for managing systems must continually evolve, and often reactively, because of frequent upgrades, changes to the architecture and interfaces, and advancing technology.

Managing performance and robustness is certainly not impossible but it is challenging.

As the demand has grown for better management, the testing has become even more prominent within the whole area of managing performance and robustness, because of the shortcomings I have mentioned here.

Basic Definitions and Concepts

Performance testers use a wide diversity of names for the same concepts, and the same word often is used for several different things, indicating the immaturity of the field. We have no universal consistency in how people use terms like performance test and robustness test. I can say that the definitions provided in this book are as much or more in the mainstream as any others.

What do we mean by the following terms? Performance engineering vs. management vs. testing; load vs. stress; robustness; response time vs. throughput; operational profile vs. benchmark vs. baseline; concurrency; scalability vs. bottleneck; self-tuning system; and service level agreement? For now, we will define just enough to get started (with the rest coming later).

Basic Terms

A system's performance is its speed or responsiveness, its ability to handle loads and its efficient use of resources. The load or work load is the mix of demands placed on a system.

System Performance Testing

Performance engineering is the field of defining performance requirements, and designing and implementing systems to meet these requirements. Performance management is monitoring and troubleshooting on-going performance in live operation.

Definition of Performance Testing

The purpose of performance testing is to measure a system's performance under load. As Humpty Dumpty said, a word can mean whatever one chooses it to mean, so it is worth our time to examine what we mean by the words "measure", "performance" and "load".

Performance testing is a measurement of performance characteristics, although sometimes the use of the word "testing" confuses people. Some performance professionals feel strongly that it is important to not use the term "performance testing", but to call it performance measurement instead. They are concerned that this measurement will get confused with feature testing and debugging, which it is not. They point out that measurement is only testing if the collected measurements are checked against pre-established goals for performance, and that measurement is often done without preconceptions of required performance.

These people have a good point: clarity of terminology is important. But since most people use the term "performance testing" we will go with the majority and use it too.

The term performance can mean response time, throughput, availability, error rate, resource utilization, or another system characteristic (or group of them), which we are interested in measuring. "All promise outruns performance." Ralph Waldo Emerson

Performance testing simulates the typical user experience under normal working conditions. The load is a typical, representative mix of demands on the system. (And, of course, there can be several different representative loads -- the work load at 2 p.m., at 2 a.m., etc.) Another name sometimes used for a performance test is a capacity test, though there is a minor difference in these terms as we will see later.

First, the performance testers need to define what the term performance means in a specific test situation -- that is, what the objectives are and what we need to measure in the test.

The answer to this question is that we measure performance usually as a weighted mix of three characteristics of a system: throughput, response time and availability. In real-time systems, for example, the users need a guarantee that a task will always be completed within a fixed time limit. Performing a task

correctly but a millisecond too late could literally be fatal.

The term load simply means the mix of demands placed on a system while we measure its performance and robustness characteristics. In practice, most loads vary continually, so later we will address the challenge of determining the most appropriate load(s) for testing. The terms work load and benchmark are sometimes used as synonyms for load. A benchmark usually means a standard load, one used to compare the performance of systems, system versions, or hardware environments, but the benchmark is not necessarily the actual mix of demands at any one user installation. The term work load is a synonym for a load, and you see both of the terms in this book: they are interchangeable.

Definition of Load Testing

In contrast to a performance test, a load test is a measurement of performance under heavy load: the peak or worst-case conditions. Because loads can have various sizes, more precise terms for this type of testing are peak-load testing or worst-case-load testing.

A performance test usually is done with a typical, representative load, but this measurement may not tell us much about the system's behavior under heavy load. For example, let's assume that the peak load on a system is only 15% more than the average load. The system performance may degrade gracefully – the system runs 15% slower at peak load. Often, though, the performance under load is non-linear: as the load increases by a moderate amount (in this case, 15%), the response time does not increase by a comparable percentage but instead becomes infinite because the system fails under the increased load.

The reliability goal (MTBF) determines the horizontal time duration, which determines the peak load for testing. An imprecise shortcut: multiply the average load by a fixed factor – 3 to 5 times more for client/server; 10 to 25 times more for Web sites with sudden spikes.

The idea is to impose an unreasonable load on the system, an overload, without providing the resources which the system needs to process that load. In contrast to a performance test, a load test measures performance under heavy load: the peak or worst-case conditions.

Definition of Stress Testing

A stress test is one which deliberately stresses a system by pushing it beyond its specified limits. The idea is to impose an unreasonable load on the system, an overload, without providing the resources which the system needs to process that load.

In a stress test, one or more of the system resources, such as the processor,

memory, or database I/O access channel, often “maxes out” and reaches saturation. (Practically, saturation can happen at less than 100% of the theoretical usable amount of the resource, for many reasons.)

This means that the testware (the test environment, test tools, etc.) must be sufficiently robust to support the stress test. We do not want the testware to fail before we have been able to adequately stress the system.

Many bugs found in stress testing are feature bugs which we cannot see with normal loads but are triggered under stress. This can lead to confusion about the difference between a feature bug and a stress bug. We will address this issue in the upcoming section entitled: “Testing Performance and Robustness versus Features”.

Some testers prize stress testing because it is so fruitful in finding bugs. Others think it is dangerous because it misdirects projects to fix irrelevant bugs. Stress testing often finds many bugs, and fixing these bugs leads to significant delays in the system delivery, which in turn leads to resistance to fixing the bugs. If we find a bug with a test case or in a test environment which we can’t connect to actual use, people are likely to dismiss it with comments like: “The users couldn’t do that”, “.. wouldn’t do that” or “... shouldn’t do that.”

Stress, Robustness and Reliability

Although stress, robustness and reliability are similar, the differences among them mean that we test them in related but different ways.

We stress a system when we place a load on it which exceeds its planned capacity. This overload may cause the system to fail, and it is the focus of stress testing.

Systems can fail in many ways, not just from overloading. We define the robustness of a system by its ability to recover from problems; its survivability. Robustness testing tries to make a system fail, so we can observe what happens and whether it recovers. Robustness testing includes stress testing but is broader, since there are many ways in which a system can fail as well as from overloading.

Reliability is most commonly defined as the mean time between failure (MTBF) of a system in operation, and as such it is closely related to availability. Reliability testing measures MTBF in test mode and predicts what the system reliability will be in live operation.

Robustness and reliability testing are discussed in the companion volume to this book, entitled “System Robustness Testing”.

System Performance Testing

Work Load and Work Flow Testing

A work load is a mix of demands on a system. The term work flow is sometimes used to mean the same thing, or we simply call it the load. Depending on the situation, these demands can be events, transactions, queries, interrupts, or other demands, or a mix of these.

We could claim that by this definition any mix of demands qualifies as a work load– and it does. The critical question is what the work load represents.

Having the appropriate test work load (or more likely, set of related work loads) is crucial, because most system characteristics can vary significantly with the work load – performance, reliability, and so on.

This means that the critical success factor for a work load is realism. If the work load used in testing is reasonably close to the work load of a particular user, then we can say with confidence that the conclusions formed by testing in our lab are likely to apply to that user. If not, it is anybody's guess how well the lab findings predict what will happen in that user's environment.

The term "work load" test does not imply any particular set of test objectives or measurements (though internally an organization might use the term as a label for their own particular flavor of testing) Depending on what we want to accomplish, a work load can be used in many different types of testing. For example, we could run a work load while we measure the system's performance (response time, throughput, availability, etc.) Or we could run a heavy work load to stress the system, attempt to overload it and see if it recovers from failures.

Work loads are used in interoperability testing (checking if subsystems connect and communicate), configuration testing (checking if features are compatible across different system versions, and across different support hardware, databases and networks), and so on.

A complicating fact is that many systems are highly configurable – they can be modified through changes in switch settings, patches and upgrades to behave differently, and can run with different variations of hardware, etc.

Another complication is the huge number of users of a major system, such as an operating system (OS) from a major vendor, all of whom use the system in their own ways. We need a scheme to manage the numbers of variations of support environments, system settings and system uses.

The common way to resolve being overwhelmed by the number of variations is a categorization model, which allows us to group together similar users by type of work load (usage pattern) and by type of support environment.

System Performance Testing

Dependability

The term “dependability” is often used to summarize the degree to which we can rely on a system. This term is broader than just reliability and recoverability – it encompasses security, safety and data integrity too.

Operational Profile

An operational profile (OP) is a list of the demands placed on a system, together with frequency of occurrence for each one. Depending on the situation, the demands could be events, transactions, messages, database accesses, etc. The operational profile of the test load should match the profile expected in live operation.

Benchmark

A benchmark is a standard work load which used in testing, instead of one based on an OP. Benchmarks are often industry-wide, e.g., the TPC-C and TPC-W benchmarks. Benchmarks can be used where OPs are not available, infeasible to build or not trusted. Benchmarks are convenient, but the question is how well the benchmark can substitute for an OP – how well it represents reality. Benchmarks are best used for a comparison of two or more systems’ behavior under the same load, or system versions’ behavior.

Baseline

The most common use of the term baseline is the measurement of the performance of an existing system or operation, as the “before” part of a before-and-after comparison. We measure the baseline in a live environment which will disappear (or at least be modified), when the change is implemented. The baseline helps us understand the current behavior if the metrics available are informal, anecdotal (“it runs slowly”), untrustworthy or politicized, and if it is important to later show evidence of the claimed improvements caused by the change.

Concurrency

Concurrency is the occurrence of two or more events at the same instant or during the same brief time interval. If events are not simultaneous, to be considered concurrent they should occur closely enough in time to have a non-trivial chance of competing for the same resources, or of interacting and possibly interfering with each other.