12

Service Testing

Overview

Companies build e-business Web sites to provide a service to customers. In many cases, these sites also have to make money. If a Web site provides poor service, customers will stop using it and find an alternative. Quality of service as provided by a Web site could be defined to include such attributes as functionality, performance, reliability, usability, security, and so on. For our purposes, however, we are separating out three particular Web service objectives that come under the scrutiny of what we will call "Service Testing":

1. *Performance:* The Web site must be responsive to users while supporting the loads imposed upon it.
2. *Reliability:* The Web site must be reliable or continue to provide a service even when a failure occurs if it is designed to be resilient to failure.
3. *Manageability:* The Web site must be capable of being managed, configured, and changed without a degradation of service noticeable to end users.

We have grouped these three sets of service objectives together because there is a common thread to the testing we do to identify shortcomings. In all three cases, we need a simulation of user load to conduct the tests effectively. Performance, reliability, and manageability objectives exist in the context of live

customers using the site to do business. Table 12.1 lists the risks addressed by service testing.

The responsiveness of a site is directly related to the resources available within the technical architecture. As more customers use the Web site, fewer technical resources are available to service each user's requests, and response times degrade. Although it is possible to model performance of complex environments, it normally falls to performance testers to simulate the projected loads on a comparable technical environment to establish confidence in the performance of a new Web service. Formal performance modeling to predict system behavior is beyond the scope of this book. See [1, 2] for a thorough treatment of this subject.

Table 12.1
Risks Addressed by Service Testing

ID	Risk	Test Objective	Technique
S1	The technical infrastructure cannot support the design load and meet response time requirements.	Demonstrate that infrastructure components meet load and response time requirements.	Performance and stress testing
S2	The system cannot meet response time requirements while processing the design loads.	Demonstrate that the entire technical architecture meets load and response time requirements.	Performance and stress testing
S3	The system fails when subjected to extreme loads.	Demonstrate that system regulates and load balances incoming messages or at least fails gracefully as designed.	Performance and stress testing
S4	The system's capacity cannot be increased in line with anticipated growth in load (scalability).	Demonstrate scalability of key components that could cause bottlenecks.	Performance and stress testing
S5	The system cannot accommodate anticipated numbers of concurrent users.	Demonstrate that the system can accommodate through load balancing and regulation the anticipated numbers of concurrent users.	Performance and stress testing

Table 12.1 (continued)

ID	Risk	Test Objective	Technique
S6	Partial system failure causes the entire system to fail or become unavailable.	Simulate specified failure scenarios and demonstrate that failover capabilities operate correctly.	Reliability/failover testing
S7	The system cannot be relied upon to be available for extended periods.	Demonstrate that throughput and response-time requirements are met without degradation over an extended period.	Reliability/failover testing
S8	System management procedures (start-up, shutdown, installation, upgrade, configuration, security, backup, and restoration) fail.	Demonstrate that system management procedures (start-up, shutdown, installation, upgrade, configuration, security, backup, and restoration) work correctly.	Service management testing
S9	Backup and recovery in case of minor failure are not in place or fail.	Demonstrate that service can be recovered from selected modes of failure.	Service management testing
S10	Disaster recovery plans are not in place or fail.	Demonstrate that contingency planning measures restore service in the case of catastrophic failure.	Service management testing

Obviously, a Web site that is lightly loaded is less likely to fail. Much of the complexity of software and hardware exists to accommodate the demands for resources within the technical architecture when a site is heavily loaded. When a site is loaded (or overloaded), the conflicting requests for resources must be managed by various infrastructure components, such as server and network OSs, database management systems, Web server products, object request brokers, middleware, and so on. These infrastructure components are usually more reliable than the custom-built application code that demands the resource, but failures can occur in two ways:

1. Infrastructure components fail because application code (through poor design or implementation) imposes excessive demands on resources.

2. Application components fail because the resources they require may not always be available (in time).

By simulating typical and unusual production loads over an extended period, testers can expose flaws in the design or implementation of the system. When these flaws are resolved, the same tests will demonstrate the system to be resilient.

Service management procedures are often the last (and possibly) the least tested aspects of a Web service. When the service is up and running, there are usually a large number of critical management processes to be performed to keep the service running smoothly. On a lightly used Web site or an intranet where users access the service during normal working hours, it might be possible to shut down a service to do routine maintenance, like making backups or performing upgrades, for example. A retail-banking site, however, could be used at any time of the day or night. Who can say when users will access their bank accounts? For international sites, the work day of the Web site never ends. At any moment, there are active users. The global availability of the Web means that many Web sites never sleep. Inevitably, management procedures must be performed while the site is live and users are on the system. It's a bit like performing open-heart surgery on an athlete running a marathon. Management procedures need to be tested while there is a load on the system to ensure they do not adversely impact the live service.

Performance and associated issues, such as resilience and reliability, dominate many people's thinking when it comes to nonfunctional testing. Certainly, everyone has used Web sites that were slow to respond, and there have been many reports of sites that failed because large numbers of people visited them simultaneously. In such cases, failures occur because applications are undersized, poorly designed, untuned, and inadequately tested. We have split our discussion of performance testing into five sections:

1. What Is Performance Testing?
2. Prerequisites for Performance Testing
3. Performance Requirements
4. The Performance Test Process
5. Performance Testing Practicalities

These five sections provide a broad overview of the full range of performance test activities. The three remaining sections in the chapter discuss other aspects of service testing.

What Is Performance Testing?

In some respects, performance testing is easy. You need to simulate a number of users doing typical transactions. Simultaneously, some test transactions are executed and response times are measured as a user would experience them. Of course, we could use real people to simulate users in production, and on more than one occasion, we have been involved in tests like that; however, manual performance testing is labor intensive, difficult to automate, unreliable (in terms of measurement), and boring for those involved. In this chapter, we are only concerned with tests performed using automated tools.

The principles of performance testing are well-documented. Our paper "Client/Server Performance Testing" [3] provides an overview of the principles and methods used to simulate loads, measure response times, and manage the process of performance testing, analysis, and tuning. In this chapter, we won't provide a detailed description of how performance tests are planned, constructed, executed, and analyzed. See [4–7] for excellent technical advice, tips and techniques for performance testing, and advice on the selection of tools and methods for Web site tuning and optimization. In many ways, the technical issues involved in conducting performance testing are the easiest to deal with. (We are not saying that solving performance problems is easy; that is a topic for another book.) Here, we will focus particularly on the challenges of performance testing from a nontechnical point of view and on some specific difficulties of Web performance testing. In our experience, once the tools have been selected, organizational, logistical, and sometimes political issues pose the greatest problems.

An Analogy

Figure 12.1 shows an athlete on a treadmill being monitored by a technician. The system under test is the athlete. The source of load is the treadmill. The technician monitors vital signs, such as pulse rate, breathing rate, and volumes of oxygen inhaled and carbon dioxide exhaled, as the athlete performs the test. Other probes might measure blood pressure, lactic-acid level, perspiration, and even brain-wave activity. The test starts with the athlete at rest and quiescent measurements are taken. The treadmill is started. The athlete walks slowly, and measurements are taken again. The speed is increased to 4 mph. The athlete copes easily with the brisk walk and measurements are taken. The speed is increased to 6 mph, 8 mph, and 10 mph. The athlete is running steadily, breathing hard, but well in control. The speed is increased to 12 mph, 14 mph, and 16 mph. The athlete is panting hard now, showing

Figure 12.1 Performance testing an athlete.

signs of stress, but coping. The speed is increased to 17 mph, 18 mph, and 20 mph. The athlete is sprinting hard now, gasping for air, barely in control, but managing the pace. The speed is increased to 21 mph, 22 mph, and 23 mph. The athlete staggers, falls, collapses, and rolls off the end of the treadmill. And dies! Well, maybe not.

This sounds brutal doesn't it? Of course, we would never perform this kind of test with a real person (at least not to the degree that they die). This is, however, exactly the approach we adopt when performance testing computer systems. Performance testing consists of a range of tests at varying loads where the system reaches a steady state (loads and response times reach constant levels). We measure load and response times for each load simulated for a 15 to 30 minute period to get a statistically significant number of measurements. We also monitor and record the vital signs for each load simulated. These are the various resources in our system (e.g., CPU and memory usage, network bandwidth, and input-output rates, and so on).

We then plot a graph of these varying loads against the response times experienced by our virtual users. When plotted, our graph looks something like Figure 12.2. At zero-load, where there is only a single user on the system, that user has the entire resource to him- or herself, and response times are fast. As we increase the load and measure response times, they get

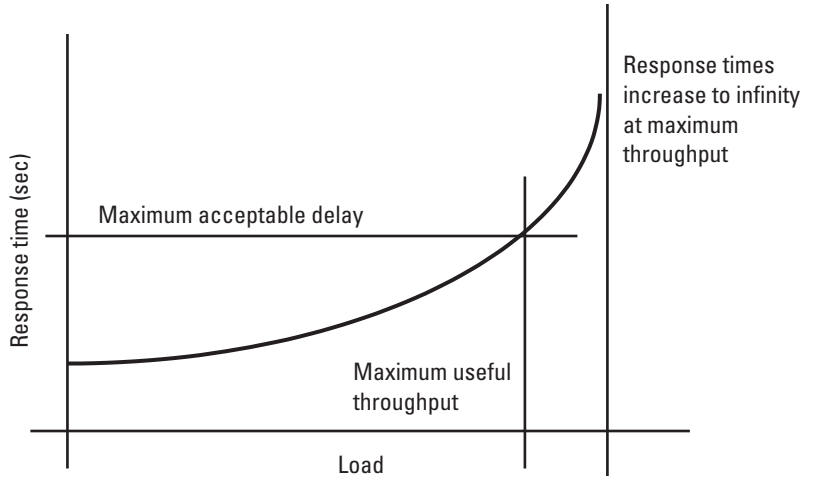

Figure 12.2 Load-response time graph.

progressively slower until we reach a point where the system is running at maximum capacity. It cannot process any more transactions beyond this maximum level. At this point, the response time for our test transactions is theoretically infinite because one of the key resources of the system is completely used up and no more transactions can be processed.

As we increase the loads from zero to the maximum, we also monitor the usage of various resource types. These resources include, for example, server processor usage, memory usage, network bandwidth, database locks, and so on. At the maximum load, one of these resources is 100% used up. This resource is the limiting resource, because it runs out first. Of course, at this point response times have degraded to the point that they are probably much slower than would be acceptable. Figure 12.3 shows a typical resource-usage load graph. To increase the throughput capacity or reduce response times for a system, we must do one of the following:

- Reduce the demand for the resource, typically by making the software that uses the resource more efficient (usually a development responsibility);
- Optimize the use of the hardware resource within the technical architecture, for example, by configuring the database to cache more data in memory or by prioritizing some processes above others on the application server;

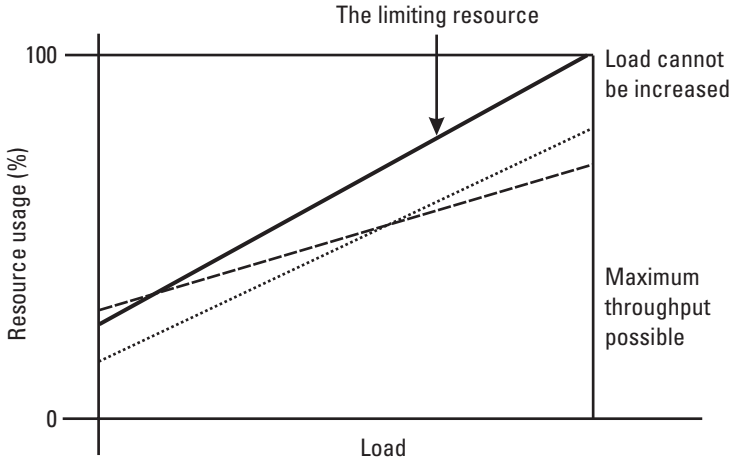

Figure 12.3 Resource usage-load graph.

- Make more of the resource available, normally by adding processors, memory, or network bandwidth, and so on.

Performance testing normally requires a team of people to help the testers. These are the technical architects, server administrators, network administrators, developers, and database designers and administrators. These technical experts are qualified to analyze the statistics generated by the resource monitoring tools and judge how best to adjust the application or to tune or upgrade the system. If you are the tester, unless you are a particular expert in these fields yourself, don't be tempted to pretend that you can interpret these statistics and make tuning and optimization decisions. It is essential to involve these experts early in the project to get their advice and commitment and, later, during testing, to ensure that bottlenecks are identified and resolved.

Performance Testing Objectives

The objective of a performance test is to demonstrate that the system meets requirements for transaction throughput and response times simultaneously. More formally, we can define the primary objective as follows:

> To demonstrate that the system functions to specification with acceptable response times while processing the required transaction volumes on a production sized database [3].

The main deliverables from such a test, prior to execution, are automated test scripts and an infrastructure to be used to execute automated tests for extended periods. This infrastructure is an asset and an expensive one too, so it pays to make as much use of this infrastructure as possible. Fortunately, the test infrastructure is a test bed that can be used for other tests with the following broader objectives:

- *Assessment of the system's capacity for growth:* The load and response data gained from the tests can be used to validate the capacity planning model and assist decision making.
- *Identification of weak points in the architecture:* The controlled load can be increased to extreme levels to stress the architecture and break it; bottlenecks and weak components can be fixed or replaced.
- *Tuning of the system:* Repeat runs of tests can be performed to verify that tuning activities are having the desired effect of improving performance.
- *Detection of obscure bugs in software:* Tests executed for extended periods can cause memory leaks, which lead to failures and reveal obscure contention problems or conflicts. (See the section on reliability/failover testing in this chapter.)
- *Verification of resilience and reliability:* Executing tests at production loads for extended periods is the only way to assess the system's resilience and reliability to ensure required service levels are likely to be met. (See the section on reliability/failover testing.)

Your test strategy should define the requirements for a test infrastructure to enable all these objectives to be met.

Prerequisites for Performance Testing

We can identify five prerequisites for a performance test. Not all of these need be in place prior to planning or preparing the test (although this might be helpful). Rather, the list below defines what is required before a test can be executed. If any of these prerequisites are missing, be very careful before you proceed to execute tests and publish results. The tests might be difficult or impossible to perform, or the credibility of any results that you publish may be seriously flawed and easy to undermine.

Quantitative, Relevant, Measurable, Realistic, and Achievable Requirements

As a foundation for all tests, performance requirements (objectives) should be agreed on beforehand so that a determination of whether the system meets those requirements can be made. Requirements for system throughput or response times, in order to be useful as a baseline for comparing performance results, must be:

- Quantifiable;
- Relevant to the task a user wants to perform;
- Measurable using a tool (or stopwatch);
- Realistic when compared with the duration of the user task;
- Economic.

These attributes are described in more detail in [3]. Often, performance requirements are vague or nonexistent. Seek out any documented requirements if you can, and if there are gaps, you may have to document them retrospectively. Performance requirements are discussed in more detail in the next section.

A Stable System

A test team attempting to construct a performance test of a system with poor-quality software is unlikely to be successful. If the software crashes regularly, it will probably not withstand the relatively minor stress of repeated use. Testers will not be able to record scripts in the first instance, or may not be able to execute a test for a reasonable length of time before the application, middleware, or operating system crashes. Performance tests stress all architectural components to some degree, but for performance testing to produce useful results, the system and the technical infrastructure have to be reasonably reliable and resilient to start with.

A Realistic Test Environment

The test environment should ideally be identical to the production environment, or at least a close simulation, dedicated to the performance test team for the duration of the test. Often this is not possible. For the results of the test to be meaningful, however, the test environment should be comparable to the final production environment. A test environment that bears no similarity to the final environment might be useful for finding obscure faults in the code, but it is useless for a performance analysis.

A simple example where a compromise might be acceptable would be where only one Web server is available for testing, but where the final architecture will balance the load between two identical servers. Reducing the load imposed to half during the test might provide a good test from the point of view of the Web server, but might understate the load on the network. In all cases, the compromise environment to be used should be discussed with the technical architect, who may be able to provide the required interpretations.

The performance test will be built to provide loads that simulate defined load profiles and scalable to impose higher loads. You would normally overload your system to some degree to see how much load it can support while still providing acceptable response times. You could interpret the system's ability to support a load 20% above your design load in two ways:

1. You have room for 20% growth beyond your design load.
2. There is a 20% margin of error for your projected load.

Either way, if your system supports a greater load, you will have somewhat more confidence in its capacity to support your business.

A Controlled Test Environment

Performance testers not only require stability in the hardware and software in terms of its reliability and resilience, but also need changes in the environment or software under test to be minimized. Automated scripts are extremely sensitive to changes in the behavior of the software under test. Test scripts designed to drive Web browsers are likely to fail immediately if the interface is changed even slightly. Changes in the operating system environment or database are equally likely to disrupt test preparation, as well as execution, and should be strictly controlled. The test team should ideally have the ability to refuse and postpone upgrades to any component of the architecture until they are ready to incorporate changes to their tests. Changes intended to improve the performance or reliability of the environment would normally be accepted as they become available.

The Performance Testing Toolkit

The five main tool requirements for our performance testing toolkit are summarized as follows:

- *Test database creation and maintenance:* It is necessary to create large volumes of data in the test database. We'd expect this to be a SQL-

based utility or perhaps a PC-based product like Microsoft Access™ connected to your test database.

- *Load generation:* The common tools use test drivers that simulate virtual clients by sending HTTP messages to Web servers.
- *Application running tool:* This tool drives one or more instances of the application using the browser interface and records response time measurements. (This is usually the same tool used for load generation, but doesn't have to be.)
- *Resource monitoring:* This includes utilities that monitor and log client and server system resources, network traffic, database activity, and so forth.
- *Results analysis and reporting:* Test running and resource monitoring tools generate large volumes of results data. Although many such tools offer facilities for analysis, it is useful to combine results from these various sources and produce combined summary test reports. This can usually be achieved using PC spreadsheet, database, and word-processing tools.

Guidance on the selection and implementation of test tools can be found in [3–7]. A listing of commercially available and freeware performance test tools is presented at the end of this chapter.

Performance Requirements

Before a performance test can be specified and designed, requirements need to be agreed on for the following:

- Transaction response times;
- Load profiles (the number of users and transaction volumes to be simulated);
- Database volumes (the number of records in database tables expected in production).

It is common for performance requirements to be defined in vague terms. On more than one project, we have (as testers) had to prepare a statement of requirements upon which to base performance tests. Often, these requirements are based on forecasted business volumes that are sometimes unrealistic. Part of the tester's job may be to get business users to think about performance requirements realistically.

Response Time Requirements

When asked to specify performance requirements, users normally focus attention on response times and often wish to define requirements in terms of generic response times. A single response time requirement for all transactions might be simple to define from the user's point of view, but is unreasonable. Some functions are critical and require short response times; others are less critical and response-time requirements can be less stringent.

The following are some guidelines for defining response-time requirements:

- For an accurate representation of the performance experienced by a live user, response times should be defined as the period between a user's requesting the system to do something (e.g., clicking on a button) to the system's returning control to the user.
- Requirements can vary in criticality according to the different business scenarios envisaged. Subsecond response times are not always appropriate!
- Generic requirements are described as catch-all thresholds. Examples of generic requirements are times to load a page, navigate between pages, and so forth.
- Specific requirements define the requirements for identified system transactions. Examples might include the time to log in, search for a book by title, and so forth.
- Requirements are usually specified in terms of acceptable maximum, average, and 95th-percentile times.

The test team should set out to measure response times for all specific requirements and a selection of transactions that provide two or three examples of generic requirements.

Jakob Nielsen, in his book *Designing Web Usability* [8] and on his Web site, useit.com (http://www.useit.com), promotes a very simple three-tiered scheme based on work done by Robert B. Miller in 1968. After many years, these limits remain universally appropriate:

- One tenth of a second is the limit for having the user feel the system is reacting instantaneously.
- One second is the limit to allow the user's flow of thought to be uninterrupted.

- Ten seconds is about the limit for keeping the user's attention. Some might say this limit is 8, or even fewer, seconds.

It's up to you to decide whether to use these limits as the basis for your response-time requirements or to formulate your own, unique requirements in discussion with business users. Some sample response-time requirements and a discussion of them are provided in [3].

Load Profiles

The second component of performance requirements is a schedule of load profiles. A load profile is a definition of the level of system loading expected to occur during a specific business scenario. There are three types of scenarios to be simulated:

1. *Uncertain loads:* The mix of users and their activities is fixed, but the number of users and the size of load vary in each test. These tests reflect our expected or normal usage of the site, but the scale of the load is uncertain.
2. *Peak or unusual situations:* These include specific scenarios that relate, for example, to the response to a successful marketing campaign or special offer or to an event (which may be outside your control) that changes users' behavior.
3. *Extreme loads:* Although we might reasonably expect this never to occur, stress tests aim to break the architecture to identify weak points that can be repaired, tuned, or upgraded. In this way we can improve the resilience of the system.

We normally run a mix of tests, starting with a range of normal loads, followed by specific unusual-load scenarios, and finally stress tests to harden the architecture.

Requirements Realism

We probably know the load limits of an internal intranet. Intranets normally have a finite (and known) user base, so it should be possible to predict a workload to simulate. With Internets, however, there is no reasonable limit to how many users could browse and load your Web site. The calculations are primarily based on the success of marketing campaigns, word-of-mouth recommendations, and, in many cases, luck rather than judgment.

Where it is impossible to predict actual loads, it is perhaps better to think of performance testing less as a test with a defined target load, but as a

measurement exercise to see how far the system can be loaded before selected response times become unacceptable. When management sees the capability of its technical infrastructure, it may realize how the technology constrains its ambitions; however, management might also decide to roll the new service out in a limited way to limit the risk of performance failures.

In our experience of dealing with both established firms and start-up companies, the predicted growth of business processed on their Web sites can be grossly overestimated. To acquire budget or venture capital and attention in the market, money is invested based on high expectations. High ambition has a part to play in the game of getting backing. Being overly ambitious, however, dramatically increases the cost of the infrastructure required and of performance testing. Ambitious targets for on-line business volumes require expensive test software licenses, more test hardware, more network infrastructure, and more time to plan, prepare, and execute.

On one occasion, a prospective client asked us to prepare a proposal to conduct a 2,000 concurrent-user performance test of their Web-based product. Without going into detail, their product was a highly specialized search engine that could be plugged into other vendor's Web sites. At our first meeting, we walked through some of the metrics they were using. They predicted the following:

- 100,000 users would register to use their search engine.
- 50,000 registered users would be active and would access the site once a week.
- Each user would connect to their service for an average of 10 minutes.
- Most users would access their system between 8 A.M. and 12 P.M.

We sketched out a quick calculation that surprised the management around the table:

- In an average week, if all active users connect for 10 minutes, the service would need to support 500,000 session minutes.
- If users connect between the hours of 8 A.M. to 12 P.M., these sessions are served in a live period of seven 16-hour days (equal to 126 hours or 7,560 minutes).
- The average number of concurrent sessions must therefore be 500,000 session minutes spread over 7,650 minutes of uptime.
- On average, the service needs to support 66.1 concurrent-user sessions.

The management team around the table was surprised to hear that their estimates of concurrent users might be far too high. Of course, this was an estimate of the average number of sessions. There must be peak levels of use at specific times of the day. Perhaps there are, but no one around the table could think of anything that would cause the user load to increase by a factor of 30! Be sure to qualify any requests to conduct huge tests by doing some simple calculations—just as a sanity check.

Database Volumes

Data volumes relate to the numbers of rows that should be present in the database tables after a specified period of live running. The database is a critical component, and we need to create a realistic volume of data to simulate the system in real use. Typically, we might use data volumes estimated to exist after one year's use of the system. Greater volumes might be used in some circumstances, depending on the application.

The Performance Test Process

There are four main activities in performance testing. An additional stage, tuning, accompanies the tester's activities and is normally performed by the technical experts. Tuning can be compared with the bug-fixing activity that usually accompanies functional test activities. Tuning may involve changes to the architectural infrastructure, but doesn't usually affect the functionality of the system under test. A schematic of the test process is presented in Figure 12.4. The five stages in the process are outlined in Table 12.2.

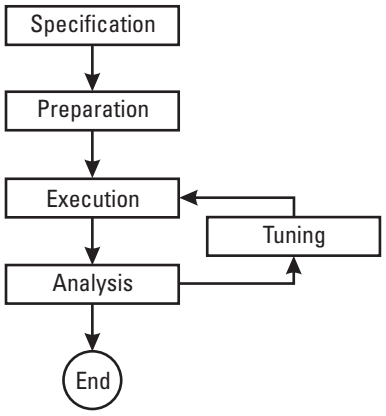

Figure 12.4 Performance test process.

Table 12.2
Performance Test Process Outline

Specification	• Documentation of the following performance requirements:
	  • Database volumes;
	  • Load profiles;
	  • Response time requirements.
	• Preparation of a schedule of load profile tests to be performed;
	• Inventory of system transactions comprising the loads to be tested;
	• Inventory of system transactions to be executed and response times measured;
	• Description of analyses and reports to be produced.

Preparation	• Preparation of a test database with appropriate data volumes; • Scripting of system transactions to comprise the load; • Scripting of system transactions whose response is to be measured (possibly the same as the load transactions); • Development of workload definitions (i.e., the implementations of load profiles); • Preparation of test data to parameterize automated scripts.
Execution	• Execution of interim tests; • Execution of performance tests; • Repeat test runs, as required.
Analysis	• Collection and archiving of test results; • Preparation of tabular and graphical analyses; • Preparation of reports, including interpretation and recommendations.
Tuning	• Performance improvements to application software, middleware, and database organization; • Changes to server system parameters; • Upgrades to client or server hardware, network capacity, or routing.

Incremental Test Development

Performance test development is usually performed incrementally in four stages:

1. Each test script is prepared and tested in isolation to debug it.
2. Scripts are integrated into the development version of the workload, and the workload is executed to test that the new script is compatible.

3. As the workload grows, the developing test framework is continually refined, debugged, and made more reliable. Experience and familiarity with the tools also grow.
4. When the last script is integrated into the workload, the test is executed as a dry run to ensure it is completely repeatable, reliable, and ready for the formal tests.

Interim tests can provide useful results. Runs of the partial workload and test transactions may expose performance problems. These can be reported and acted upon within the development groups or by network, system, or database administrators. Tests of low volume loads can also provide an early indication of network traffic and potential bottlenecks when the test is scaled up. Poor response times can be caused by poor application design and can be investigated and cleared up by the developers earlier. Inefficient SQL can also be identified and optimized. Repeatable test scripts can be run for extended periods as soak tests (see later). Such tests can reveal errors, such as memory leaks, which would not normally be found during functional tests.

Test Execution

The execution of formal performance tests requires some stage management or coordination. As the time approaches to run the test, team members who will execute the test need to liaise with those who will monitor the test. The test monitoring team members are often working in different locations and need to be kept very well informed if the test is to run smoothly and all results are to be captured correctly. They need to be aware of the time window in which the test will be run and when they should start and stop their monitoring tools. They also need to be aware of how much time they have to archive their data, preprocess it, and make it available to the person who will analyze the data fully and produce the required reports.

Beyond the coordination of the various team members, performance test execution tends to follow a standard routine:

1. Preparation of database (restored from backup, if required);
2. Preparation of test environment as required and verification of its state;
3. Start of monitoring processes (network, clients and servers, and database);
4. Start of load simulation and observation of system monitor(s);

5. If a separate tool is used, when the load is stable, start of the application test running tool and response time measurement;
6. Close supervision for the duration of the test;
7. Termination of the test when the test period ends, if the test-running tools do not stop automatically;
8. Stopping of monitoring tools and saving of results;
9. Archiving of all captured results and assurance that all results data is backed up securely;
10. Production of interim reports; conference with other team members concerning any anomalies;
11. Preparation of analyses and reports.

When a test run is complete, it is common for some tuning activity to be performed. If a test is a repeat test, it is essential that any changes in environment are recorded, so that any differences in system behavior, and hence performance results, can be matched with the changes in configuration. As a rule, it is wise to change only one thing at a time so that when differences in behavior are detected, they can be traced back to the changes made.

Results Analysis and Reporting

The application test running tool will capture a series of response times for each transaction executed. The most typical report for a test run will summarize these measurements and report the following for each measurement taken:

- Count of measurements;
- Minimum response time;
- Maximum response time;
- Mean response time;
- 95th-percentile response time.

The 95th percentile, it should be noted, is the time within which 95% of the measurements occur. Other percentiles are sometimes used, but this depends on the format of the response time requirements. The required response times are usually presented in the same report for comparison.

The other main requirement that must be verified by the test is system throughput. The load generation tool should record the count of each

transaction type for the period of the test. Dividing these counts by the duration of the test gives the transaction rate or throughput actually achieved. These rates should match the load profile, but might not if the system responds slowly. If the transaction load rate depends on delays between transactions, a slow response will increase the delay between transactions and slow the rate. The throughput will also be lower than planned if the system cannot support the transaction load.

It is common to execute a series of test runs at varying loads. Using the results of a series of tests, a graph of response times for a transaction plotted against the load applied can be prepared. Such graphs provide an indication of the rate of degradation in performance as load is increased and the maximum throughput that can be achieved while providing acceptable response times.

Resource monitoring tools usually have statistical or graphical reporting facilities that plot resource usage over time. Enhanced reports of resource usage versus load applied are very useful and can assist in the identification of bottlenecks in a system's architecture.

Performance Testing Practicalities

Nowadays, there is a choice of approach to performance testing. There are many companies offering to do your performance testing for you, and performance testing portals that offer remote testing opportunities are emerging in importance. In planning, preparing, and executing performance tests, there are several aspects of the task that cause difficulties. The problems encountered most often relate to the software and environment. The predominant issue that concerns the performance tester is stability. Unfortunately, performance testers are often required to work with software that is imperfect or unfinished. These issues are discussed in the following sections.

Which Performance Test Architecture?

There are four options for conducting performance testing. All require automated test tools, of course. To implement a realistic performance test, you need the following:

- Load generation tool;
- Load generation tool host machine(s);
- High-capacity network connection for remote users;

- Test environment with fully configured system, production data volumes, and a security infrastructure implemented with production-scale Internet connection.

The cost and complexity of acquiring and maintaining the test tool licenses, skills, hardware, and software environments can be extremely high. For some years, the tool vendors and specialist testing companies have provided on-site consultants to perform the work in your environment. Other services are emerging that make it easy to hire resources or test facilities. There are four basic choices as to how you implement performance tests:

1. Doing it yourself;
2. Outsourcing (bring external resources into your environment);
3. Remote testing (use external resources and their remote testing facility);
4. Performance test portals (build your own tests on a remote testing facility).

Some of the advantages and disadvantages of each are summarized in Table 12.3.

Table 12.3
Four Methods for Performance Testing

Doing It Yourself	**Outsourcing**	**Remote Testing**	**Performance Test Portals**
Test tool license			
You acquire the licenses for performance test tools.	You acquire the licenses for performance test tools.	Included in the price.	You rent a timeslot on a portal service.
Test tool host			
You provide.	You provide.	Included in the price.	Included in the price.
Internet connections			
You organize.	You organize.	Use of the service provider's is included. You liaise with your own ISP.	Use of the service provider's is included. You liaise with your own ISP.

Table 12.3 (continued)

Doing It Yourself	Outsourcing	Remote Testing	Performance Test Portals
Simplicity			
Complicated—you do everything.	You manage the consultants. You build the tool environment.	Simplest—the service provider does it all.	A simple infrastructure and tools solution, but you are responsible for building and running the test.
Cost			
Resources are cheaper, but tools, licenses, and hosts are expensive. You still need to buy training and skills.	Probably most expensive: you buy everything and hire external resources.	Lower tool and infrastructure costs, but expensive services. May be cheaper as a package.	Low tool and infrastructure costs—your resources are cheaper.
Pros and cons			
You acquire expensive tools that may rarely be used in the future.	Most expensive of all potentially.	Potentially cheaper—if you would have hired consultants anyway.	Lowest tool and infrastructure costs, but you still need to buy and acquire skills.
You need to organize, through perhaps two ISPs, large network connections (not an issue for intranets).	You need to organize, through perhaps two ISPs, large network connections (not an issue for intranets).	Simpler—you need one arrangement with your own ISP only.	Simplest infrastructure solution—but you must manage and perform the tests.
You are in control.	You are in control at a high level.	You can specify the tests; the service provider builds and executes them. You manage the supplier.	You are in control.
Complicated—you do everything.	Less complicated—the outsourcer manages the consultants.	Simplest solution—the service provider does it all.	Simpler technically—but you must build and execute the tests.
You can test as many times as you like.	You can test as many times as you like.	Pricing per test makes repeat tests expensive.	Price per test depends on the deal. You may get "all you can test" for a fixed period.

Compressed Timescales

One serious challenge with e-business performance testing is the time allowed to plan, prepare, execute, and analyze performance tests. We normally budget 6 to 8 weeks to prepare a test on a medium-to-high complexity environment. How long a period exists between system testing's elimination of all but the trivial faults and delivery into production? Not as long as you might need. There may be simplifying factors that make testing Web applications easier, however:

- Web applications are relatively simple from the point of view of thin clients sending messages to servers. The scripts required to simulate user activity can be very simple so reasonably realistic tests can be constructed quickly.
- Because the HTTP calls to server-based objects and Web pages are simple, they are much less affected by functional changes that correct faults during development and system testing. Consequently, work on the creation of performance test scripts can often be started before the full application is available.
- Tests using drivers (and not using the browsers user interface) may be adequate to make the test simpler.

Normally, we subject the system under test to load using drivers that simulate real users by sending messages across the network to the Web servers, just like normal clients would. In the Internet environment, the time taken by a browser to render a Web page and present it to an end user may be short compared with the time taken for a system to receive the HTTP message, perform a transaction, and dispatch a response to a client machine. It may be a reasonable compromise to ignore the time taken by browsers to render and display Web pages and just use the response times measured by the performance test drivers to reflect what a user would experience.

Scripting performance test tool drivers is comparatively simple. Scripting GUI-oriented transactions to drive the browser interface can be much more complicated. If you do decide to ignore the time taken by browsers themselves, be sure to discuss this with your users, technical architect(s), and developers to ensure they understand the compromise being made. If they deem this approach unacceptable, advise them of the delay that additional GUI scripting might introduce into the project.

Software Quality

In many projects, the time allowed for functional and nonfunctional testing (including performance testing) is squeezed. Too little time is allocated overall, and development slippages reduce the time available for system testing. Under any circumstance, the time allowed for testing is reduced, and the quality of the software is poorer than required.

When the test team receives the software to test and attempts to record test scripts, the scripts themselves will probably not stretch the application in terms of its functionality. The paths taken through the application will be designed to execute specific transactions successfully. As a test script is recorded, made repeatable, and then run repeatedly, bugs that were not caught during functional testing may begin to emerge.

One typical problem found during this period is that repeated runs of specific scripts may gradually absorb more and more client resources, leading to a failure when a resource, usually memory, runs out. Program crashes often occur when repeated use of specific features within the application causes counters or internal array bounds to be exceeded. Sometimes these problems can be bypassed by using different paths through the software, but more often, these scripts have to be postponed until these faults can be fixed.

Dedicated Environment

During test preparation, testers will be recording, editing, and replaying automated test scripts. These activities should not disturb or be disturbed by the activities of other users on a shared system. When a single test script is integrated into the complete workload and the full load simulation is run, however, other users of the system will probably be very badly affected by the sudden application of such a large load on the system.

If at all possible, the test team should have access to a dedicated environment for test development. It need hardly be stated that when the actual tests are run, there should be no other activity on the test environment. Testing on a dedicated environment is sensible if the target production infrastructure is dedicated to the system under test. However, most systems are implemented on shared infrastructure. You should discuss with your technical architect(s) what the implication of this is for your testing.

Other Potential Problems

Underestimation of the effort required to prepare and conduct a performance can lead to problems. Performance testing a large Web application is

a complex activity that usually has to be completed in a very limited timescale. Few project managers have direct experience of the tasks involved in pre- paring and executing such tests. As a result, they usually underestimate the length of time it takes to build the infrastructure required to conduct the test. If this is the case, tests are unlikely to be ready to execute in the time available.

Overambition, at least early in the project, is common. Project managers often assume that databases have to be populated with perfect data, that every transaction must be incorporated into the load, and every response time measured. Usually, the Pareto rule applies: 80% of the database volume will be taken up by 20% of the system tables; 80% of the system load will be generated by 20% of the system transactions; only 20% of system transactions need to be measured and so on. When you discuss the number of transactions to simulate and the tables to populate, you should discuss the practicalities of implementing a perfect simulation with business users, designers, and developers. Because you are always constrained by time, you need to agree on the design of the test that is achievable in time, but still meaningful.

The skills required to execute automated tests using proprietary tools are not always easy to find, but as with most software development and testing activities, there are principles that, if adhered to, should allow competent functional testers to build a performance test. It is common for managers or testers with no test automation experience to assume that the test process consists of two stages: test scripting and test running. As should be clear by now, the process is more complicated and, actually, is more like a small software development project in its own right. On top of this, the testers may have to build or customize the tools they use.

When software developers who have designed, coded, and functionally tested an application are asked to build an automated test suite for a performance test, their main difficulty is their lack of testing experience. Experienced testers who have no experience of the system under test, however, usually need a period to gain familiarity with the system to be tested. Allowance for this should be made in the early stages of test development; testers will have to grapple with the vagaries of the system under test before they can start to record scripts.

Building a performance test database may involve generating thousands or millions of database rows in selected tables. There are two risks involved in this activity. The first is that in creating the invented data in the database tables, the referential integrity of the database is not maintained. The second risk is that your data may not obey business rules, such as the reconciliation

of financial data between fields in different tables. In both cases, the load simulation may be perfect, but the application may not be able to handle such data inconsistencies and fail. In these circumstances, test scripts developed on a small coherent database might no longer work on a prepared, production-size database. Clearly, it is important that the person preparing the test database understand the database design and the operation of the application.

This problem can of course be helped if the database itself has the referential constraints implemented and will reject invalid data (often, these facilities are not used because they impose a significant performance overhead). When using procedural SQL to create database rows, the usual technique is to replicate existing database rows with a new unique primary key. Where primary keys are created from combined foreign keys, you must generate new primary keys by combining the primary keys from the referenced tables.

Presenting Results to Customers and Suppliers

On a previous project, we were hired as consultants to specify and supervise performance testing to be done by a third party. The performance testing took place towards the end of the project and was going badly (the measured response times were totally unacceptable). Our role was to advise both client and supplier and to witness and review the testing. Biweekly checkpoint meetings were held in two stages. In the first stage, the customer and their consultants discussed the progress. We reported to the project board the interim test results and our interpretation of them. Then, the suppliers were invited to join in for the second half of the meeting. The test results were presented and discussed, and quite a lot of pressure was applied on the supplier to emphasize the importance of the current shortcomings.

The point to be made here is this: It is very common to have the customer, the supplier, and the testers discussing performance issues late in the project. The situation is usually grim. The testers present the results, and the suppliers defend themselves. Typically, they suggest the tests are flawed: The load requirements are excessive. The numbers of concurrent users, the transaction rates, and the database volumes are all too high. The testers aren't simulating the users correctly. Real users would never do that. This isn't a good simulation. And so on.

The suppliers may try to undermine the testers' work. As a tester, you need to be very careful and certain of your results. You need to understand the results data and be confident that your tests are sound. You must test your tests and make sure the experts (and not you) perform analyses of the

resource monitoring statistics because it's likely that you will come under attack. Suppliers aren't above suggesting the testers are incompetent. Remember, successful performance tests are usually a major hurdle for the suppliers, and a lot of money may depend on the successful outcome of the performance testing. You have been warned.

Reliability/Failover Testing

Assuring the continuous availability of a Web service may be a key objective of your project. Reliability testing helps to flush out obscure faults that cause unexpected failures so they can be fixed. Failover testing helps to ensure that the measures designed for anticipated failures actually work.

Failover Testing

Where sites are required to be resilient, reliable, or both, they tend to be designed with reliable systems components with built-in redundancy and failover features that come into play when failures occur. These features may include diverse network routing, multiple servers configured as clusters, middleware, and distributed object technology that handles load balancing and rerouting of traffic in failure scenarios. The features that provide diversity and redundancy are also often the mechanisms used to allow the performance of backups, software and hardware upgrades, and other maintenance activities (see the section on service management testing).

These configuration options are often trusted to work adequately, and it is only when disaster strikes that their behavior is seen for the first time. Failover testing aims to explore the behavior of the system under selected failure scenarios before deployment and normally involves the following:

- Identification of the components that could fail and cause a loss of service (looking at failures from the inside out);
- Identification of the hazards that could cause a failure and loss of service (looking at threats from the outside in);
- Analysis of the failure modes or scenarios that could occur where you need confidence that the recovery measures will work;
- An automated test that can be used to load the system and explore its behavior over an extended period and monitor its behavior under failure conditions.

Mostly used in higher integrity environments, a technique called fault tree analysis (FTA) can help to understand the dependencies of a service on its underlying components. FTA and fault tree diagrams are a logical representation of a system or service and the ways in which it can fail. Figure 12.5 shows the relationship between basic component failure events, intermediate subsystem failure events, and the topmost service failure event. Of course, it might be possible to identify more than three levels of failure event. Further, events relationships can be more complicated. For example, if a server component fails, we might lose a Web server. If our service depends on a single Web server we might lose the entire service. If, however, we have two Web servers and facilities to balance loads across them, we might still be able to provide a service, albeit at a reduced level. We would lose the service altogether only if both servers failed simultaneously.

FTA documents the Boolean relationships between lower-level failure events to those above them. A complete fault tree documents the permutations of failures that can occur, but still allows a service to be provided. Figure 12.6 shows an example where the loss of any low-level component causes one of the Web servers to fail (an OR relationship). Both Web servers must fail to cause loss of the entire service (an AND relationship). This is only a superficial description of the technique. Nancy Leveson's book *Safeware* [9] provides a more extensive description.

FTA is most helpful for the technical architect designing the mechanisms for coping with failures. Fault tree diagrams are sometimes complicated with many Boolean symbols representing the dependencies of services on their subsystems and the dependencies of subsystems on their

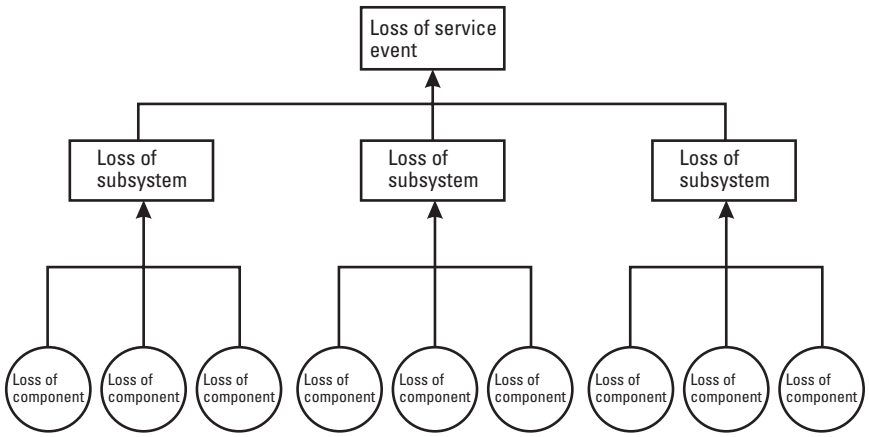

Figure 12.5 Failure event hierarchy.

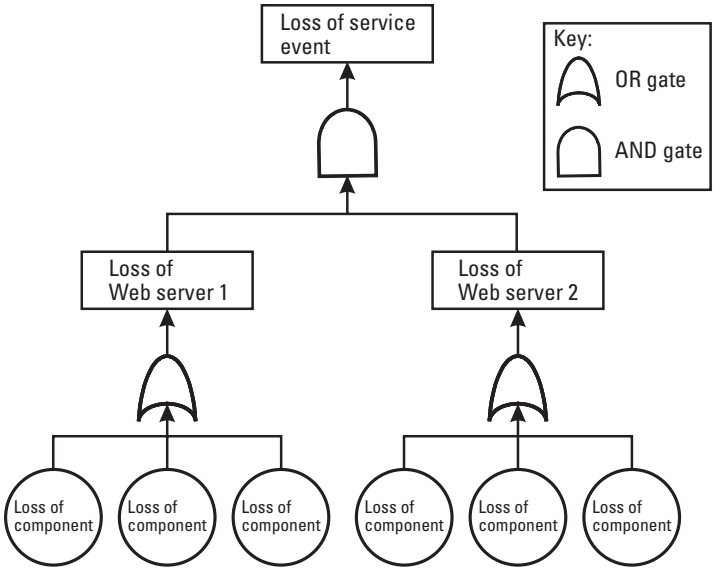

Figure 12.6 Example fault tree for a Web service.

components. These diagrams are used to identify the unusual, but possible, combinations of component failures that the system must withstand. For the tester, fault trees provide a simple model from which failover test scenarios can be derived. Typically, the tester would wish to identify all the modes of failure that the system is meant to be able to withstand and still provide a service. From these, the most likely scenarios would be subjected to testing.

Whether you use a sophisticated technique like FTA, or you are able to identify the main modes of failure easily just by discussing these with the technical experts, testing follows a straightforward approach. With an automated test running, early tests focus on individual component failures (e.g., you off-line a disk, power-down a server, break a network connection). Later tests simulate more complex (and presumably, less likely) multiple failure scenarios (e.g., simultaneous loss of a Web server and a network segment).

These tests need to be run with an automated load running to explore the system's behavior in production situations and to gain confidence in the designed-in recovery measures. In particular, you want to know the following:

- How does the architecture behave in failure situations?
- Do load-balancing facilities work correctly?

- Do failover capabilities absorb the load when a component fails?
- Does automatic recovery operate?
- Do restarted systems catch up?

Ultimately, the tests focus on determining whether the service to end users is maintained and whether the users actually notice the failure occurring.

Reliability (or Soak) Testing

Whereas failover testing addresses concern about the system's ability to withstand failures of one kind or another, reliability testing aims to address the concern over a failure occurring in the first place. Most hardware components are reliable to the degree that their mean time between failures may be measured in years. Proprietary software products may also have relatively high reliability (under normal working environments). The greatest reliability concerns typically revolve around the custom-built and bleeding-edge proprietary software that has not undergone extensive testing or use for long periods in a range of operating conditions.

Reliability tests require the use (or reuse) of automated tests in two ways to simulate the following:

1. Extreme loads on specific components or resources in the technical architecture;
2. Extended periods of normal (or extreme) loads on the complete system.

When focusing on specific components, we are looking to stress the component by subjecting it to an unreasonably large number of requests to perform its designed function. For example, if a single (nonredundant) object or application server is critical to the success of our system, we might stress test this server in isolation by subjecting it to a large number of concurrent requests to create new objects, process specific transactions, and then delete those same objects. Being critical, this server might be sized to handle several times the transaction load that the Web servers might encounter. We can't stress test this server in the context of the entire system (a stress test at the required volumes would always cause the Web servers to fail first, perhaps). By testing it in isolation, however, we gain confidence that it can support loads greater than other components whose reliability we already trust. Another example might be multiple application servers that service the

requests from several Web servers. We might configure the Web servers to direct their traffic to a single application server to increase the load several times. This might be a more realistic scenario compared with a test of the application server in isolation; however, this test sounds very similar to the situation where the application servers fail, but the last one running supports the entire service. The similarity with failover testing is clear.

Soak tests subject a system to a load over an extended period of perhaps 24 or 48 hours or longer to find (what are usually) obscure problems. Early performance tests normally flush out obvious faults that cause a system to be unreliable, but these tests are left running for less than 1 hour. Obscure faults usually require testing over longer periods. The automated test does not necessarily have to be ramped up to extreme loads; stress testing covers that. We are particularly interested, however, in the system's ability to withstand continuous running of a wide variety of test transactions to find any obscure memory leaks, locking, or race conditions. As for performance testing, monitoring the resources used by the system helps to identify problems that might take a long time to cause an actual failure. Typically, the symptoms we are looking for are increases in the amount of resources being used over time. The allocation of memory to systems without return is a called a *memory leak*. Other system resources, such as locks, sockets, objects, and so on may also be subject to leakage over time. When leaks occur, a clear warning sign is that response times get progressively worse over the duration of the test. The system might not actually fail, but given enough testing time to show these trends, the symptom can be detected.

Service Management Testing

When the Web site is deployed in production, it has to be managed. Keeping a site up and running requires that it be monitored, upgraded, backed up, and fixed quickly when things go wrong. Postdeployment monitoring is covered in Chapter 16. The procedures that Web site managers use to perform upgrades, backups, releases, and restorations from failures are critical to providing a reliable service, so they need testing, particularly if the site will undergo rapid change after deployment.

The management procedures fall into five broad categories:

1. System shutdown and start-up procedures;
2. Server, network, and software infrastructure and application code installation and upgrades;

3. Server, network, and software infrastructure configuration and security changes;
4. Normal system backups;
5. System restoration (from various modes of failure) procedures.

The first four procedures are the routine, day-to-day procedures. The last one, system restoration, is less common and is very much the exception, as they deal with failures in the technical environment where more or less infrastructure may be out of service. The tests of the routine procedures will follow a similar pattern to failover testing in that the procedures are tested while the system is subjected to a simulated load. The particular problems to be addressed are as follows:

- Procedures fail to achieve the desired effect (e.g., an installation fails to implement a new version of a product, or a backup does not capture a restorable snapshot of the contents of a database).
- Procedures are unworkable or unusable (i.e., the procedures themselves cannot be performed easily by system management or operations staff. One example would be where a system upgrade must be performed on all servers in a cluster for the cluster to operate. This would require all machines to be off-lined, upgraded, and restarted at the same time).
- Procedures disrupt the live service (e.g., a backup requires a database to operate in read-only mode for the duration of the procedure, and this would make the live service unusable).

The tests should be run in as realistic a way as possible. System management staff must perform the procedures in exactly the same way as they would in production and not rely on familiar shortcuts or the fact that they do not have real users on the service. Typically, there are no test scripts used here, but a schedule of tasks to be followed identifies the system management procedures to be performed in a defined order. The impact on the virtual users should be monitored: response times or functional differences need to be noted and explained. Further, the overall performance of the systems involved should be monitored and any degradation in performance or significant changes in resource usage noted and justified.

Tools for Service Testing

As Table 12.4 shows, a large number of tools is available for performance testing. Most of the proprietary tools run on the Windows platform and a smaller number run under Unix. The proprietary tools all have GUI front-ends and sophisticated analysis and graphing capabilities. They tend to be easier to use.

Table 12.4
Tools for Service Testing

Proprietary Tools	URL
Astra LoadTest	http://www-svca.mercuryinteractive.com
Atesto Internet Loadmodeller	http://www.atesto.com
CapCal	http://www.capcal.com
e-Load, Bean-Test	http://www.empirix.com
eValid	http://www.soft.com
forecastweb	http://www.facilita.co.uk
OpenSTA	http://www.opensta.org
Portent	http://www.loadtesting.com
PureLoad	http://www.pureload.com
QAload, EcoTOOLS, File-AID/CS	http://www.compuware.com
silkperformer	http://www.segue.com
SiteLoad	http://www.rational.com
SiteTools Loader, SiteTools Monitor	http://www.softlight.com
Test Perspective	http://www.keynote.com
Web Performance Trainer	http://www.webperformanceinc.com
Web Server Stress Tool	http://www.paessler.com/tools/
WebART	http://www.oclc.org/webart
WebAvalanche, WebReflector	http://www.cawnetworks.com
WebLOAD	http://www.radview.com
Web Polygraph	http://www.web-polygraph.org
WebSizr	http://www.technovations.com
WebSpray	http://www.redhillnetworks.com

Table 12.4 (continued)

Free Tools	URL
ApacheBench	http://www.cpan.org/modules/by-module/HTTPD/
Deluge	http://deluge.sourceforge.net
DieselTest	http://sourceforge.net/projects/dieseltest/
Hammerhead 2	http://hammerhead.sourceforge.net
http_load	http://www.acme.com/software/http_load/
InetMonitor	http://www.microsoft.com/siteserver/site/DeployAdmin/InetMonitor.htm
Load	http://www.pushtotest.com
OpenLoad	http://openload.sourceforge.net
Siege	http://www.joedog.org
SSL Stress Tool	http://sslclient.sourceforge.net
Torture	http://stein.cshl.org/~lstein/torture/
Velometer	http://www.velometer.com
Wcat	http://www.microsoft.com/downloads/
Web Application Stress Tool	http://webtool.rte.microsoft.com
WTest	http://members.attcanada.ca/~bibhasb/

Of the free tools, the majority run under Unix and many are written in Perl. They have fewer features, they produce fewer, less comprehensive reports, and their scripting languages tend to be unsophisticated. They are less flexible than the proprietary tools and many have a command-line interface. They tend to lack many of the features of proprietary tools. They are less well documented and some assume users have a technical or programming background. They are free, however, and in most cases, you get the source code to play with too. If you need to enhance or customize the tools in some way, you can.

Being short of money is one reason for choosing a free tool. Even if you have a budget to spend, however, you still might consider using one of the free tools, at least temporarily. The principles of performance testing apply to all tools, so you might consider using a free tool to gain some experience, then make a better-informed decision to buy a commercially available tool when you know what you are doing.

References

[1] Smith, C. U., *Performance Engineering of Software Systems*, Boston, MA: Addison-Wesley, 1990.

[2] Menascé, D. A., and V. A. F. Almeida, *Scaling for E-Business*, Upper Saddle River, NJ: Prentice Hall, 2000.

[3] Gerrard, P., "Client/Server Performance Testing," http://www.evolutif.co.uk, 2002.

[4] Dustin, E., J. Rashka, and D. McDiarmid, *Quality Web Systems*, Boston, MA: Addison-Wesley, 2001.

[5] Nguyen, H. Q., *Testing Applications on the Web*, New York: John Wiley & Sons, 2001.

[6] Splaine, S., and S. P. Jaskiel, *The Web Testing Handbook*, Orange Park, FL: STQE Publishing, 2001.

[7] Kolish, T., and T. Doyle, *Gain E-Confidence*, Lexington, MA: Segue Software, 1999.

[8] Nielsen, J., *Designing Web Usability*, Indianapolis, IN: New Riders, 2000.

[9] Leveson, N. G., *Safeware*, Reading, MA: Addison-Wesley, 1995.