<u>**DRAFT Chapter from Paul Gerrard's Forthcoming Book:**</u>

Risk-Based E-Business Testing

Chapter 13 - Service Testing

Risks to Be Addressed

ID	*Risk*	*Test Objective*	*Technique*
	The technical infrastructure cannot support fast loads up to the design load and response time requirements.	Demonstrate that infrastructure components meet load and response time requirements.	Performance and Stress Testing
	The system cannot support loads up to (any beyond) the design load.	Demonstrate that the entire technical architecture meets load requirements.	Performance and Stress Testing
	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
	The system fails when subjected to extreme loads.	Demonstrate that system regulates/load balances incoming messages or at least fails 'gracefully' as designed.	Performance and Stress Testing
	The system's capacity cannot be increased in line with anticipated growth in load (scalability).	Demonstrate that the key components that could cause bottlenecks can be scaled.	Performance and Stress Testing
	The system cannot accommodate anticipated numbers of concurrent users.	Demonstrate that the system can accommodate through load balancing/regulation the anticipated numbers of concurrent users.	Performance and Stress Testing
	Partial system failure causes the entire system to fail/be unavailable.	Simulate specified failure scenarios and demonstrate failover capabilities operate correctly.	Reliability/Resilience Testing
	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 of time.	Reliability/Resilience Testing

ID	*Risk*	*Test Objective*	*Technique*
	System management procedures fail.	Demonstrate that system management procedures (start-up, shutdown, installation, upgrade, configuration, security, backup and restoration) work correctly.	Service Management Testing
	Backup and recovery from 'minor' failures	Demonstrate that service can be recovered from selected modes of failure.	Service Management Testing
	Disaster recovery scenarios	Demonstrate that contingency planning measures restore the service in the case of catastrophic failure.	Service Management Testing

Overview

Companies build E-Business web sites to provide a service to customers. In many cases, these sites have also to make money. If a web site provides a poor service, customers will stop using it and find an alternative. The quality of service that a web site provides could be defined to include all the attributes such as functionality, performance, reliability, usability, security and so on. However, for our purposes, we are separating out three particular web service objectives that come under the scrutiny of what we will call Service Testing. These objectives are:

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

We have grouped these three sets of performance 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.

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 will 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 behaviour is beyond the scope of this book. References [1] and [2] provide a thorough treatment of this subject.

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 operating systems, database management systems, web server products, object requests 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 either.

- Infrastructure components fail because application code (through poor design or implementation) imposes excessive demands on resources.
- The application components may 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 large number of critical management processes to be performed to keep the service running smoothly. On a lightly used website or an intranet where users access the service in normal working hours, it might be possible to shut down a service to do routine maintenance, backups or upgrades for example. However, a retail-banking site could be used at any time of the day or night. Who can say when your users will access their bank accounts? For sites that are international, the working day of the website never ends. At any moment in time, there are active users of your web site. 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 doing 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 non-functional 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 sites simultaneously. In these cases, failures occur because applications are undersized, poorly designed, untuned and inadequately tested.

Performance Testing

In some respects, performance testing is easy. You need to simulate a number of users doing typical transactions and there are many tools available to do this. Simultaneously, some test transactions are executed and response time measurements taken, as a user would experience them. There are many tools available to do this too. What's all the fuss about? It sounds simple, but of course there is a lot more to think about than just the automated tools.

The principles of performance testing are well documented. Our paper, Client/Server Performance Testing (reference [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. References [4], [5], [6] and [7] all provide excellent technical advice, tips and techniques for performance testing, including the selection of tools and options for web site tuning and optimization. In many ways, the technical issues of 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). But here, we will focus particularly on the particular challenges of performance testing from a non-technical point of view and some specific difficulties that the web brings. In our experience, once the tools have been selected, organizational, logistical and sometimes political issues pose the greatest problems.

An Analogy

Figure 13.1 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 the athlete's 'vital signs' as they perform the test. The vital signs might include the pulse rate, breathing rate and volumes of oxygen inhaled and carbon dioxide exhaled. Other probes might measure blood pressure, lactic acid level, perspiration and even brainwave activity. The test starts with the athlete at rest and quiescent measurements taken. The treadmill is started. The athlete walks slowly and measurements are taken again. The speed is increased to 4 miles an hour. The athlete copes easily with the brisk walk and measurements are taken. The speed is increased to 6 then 8 then 10 miles an hour. The athlete is running steadily, breathing hard but well in control. The speed is increased to 12, 14 and 16 miles an hour, the athlete is panting hard now, showing signs of stress but coping. The speed is increased to 17, 18 and 20 miles an hour. The athlete is sprinting hard now gasping for air barely in control but managing the pace well. The speed is increased to 21, 22 and 23 miles an hour the athlete staggers, falls, collapses and rolls off the end of the treadmill. And dies.

This sounds brutal doesn't it? Of course, we would not contemplate this kind of test with a real person (at least not to the degree that they die). However, this is exactly the approach we adopt when performance testing computer systems.

- Performance testing consists of a range of tests at varying load where the system reaches a steady state (loads and response times at constant levels).
- We measure load and response times for each load simulated for a 15-30 minute period to get a statistically significant number of measurements.
- We 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 I/O rates etc.

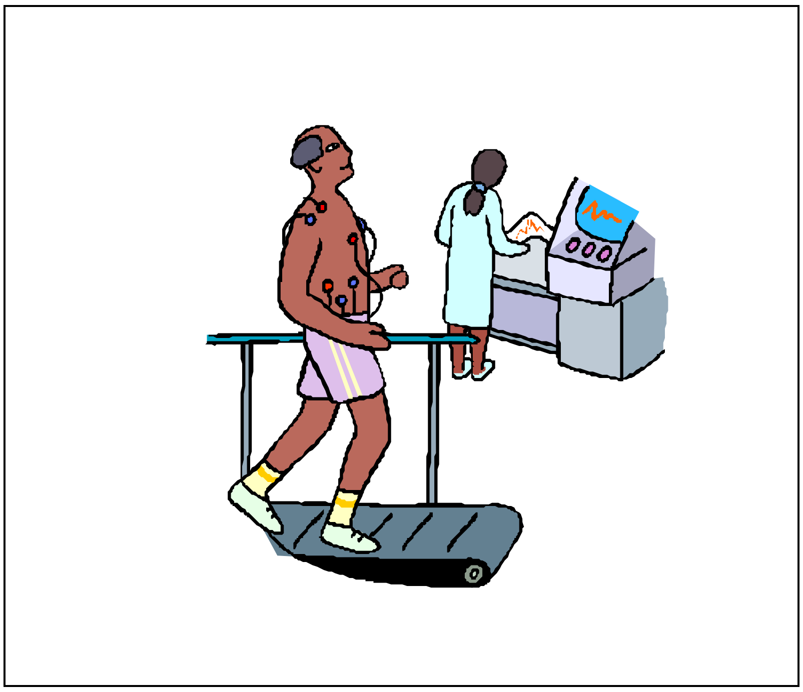

Figure 13.1 Performance testing an athlete.

We plot a graph of these varying loads against the response times experienced by our end users. When plotted, our graph looks something like figure 13.2 below. At zero-load, where there is only a single user on the system, they have the entire resource to themselves and the response times are fast. As we introduce increased loads and measure response times, they get progressively worse 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.

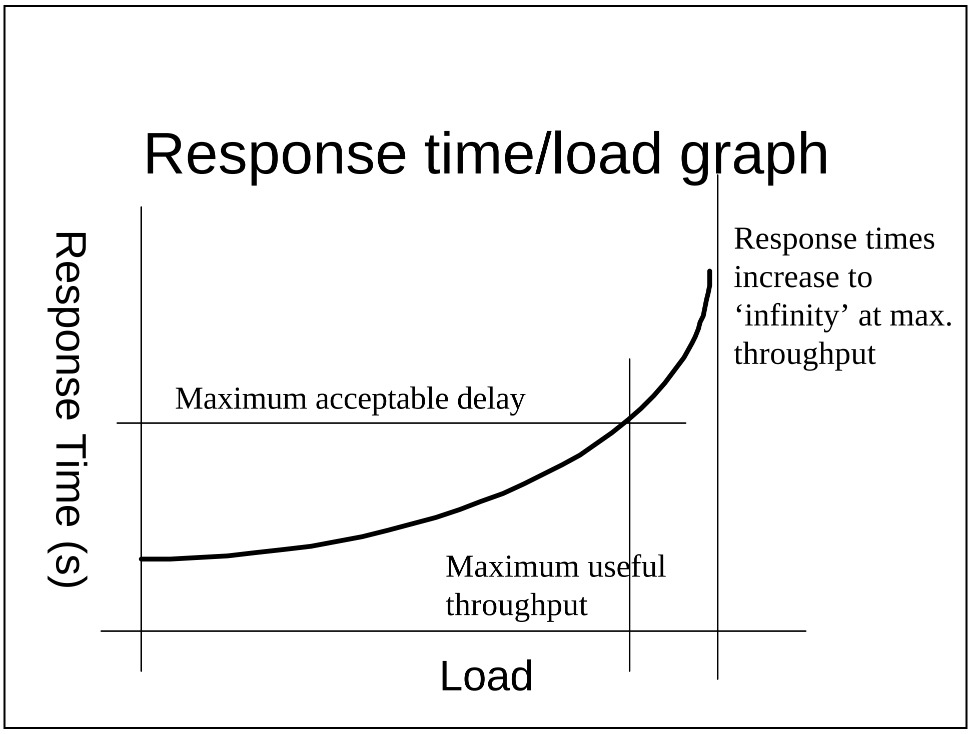

Figure 13.2 Response Time/Load Graph

As we increase the loads from zero up to the maximum, we monitor the usage of various resource types. These resources are for example, server processor usage, memory usage, network bandwidth, database locks and so on. At the maximum load, one of these resources is used up 100%. This resource is the limiting resource, because it runs out first. Figure 13.3 shows how a typical resource/load graph.

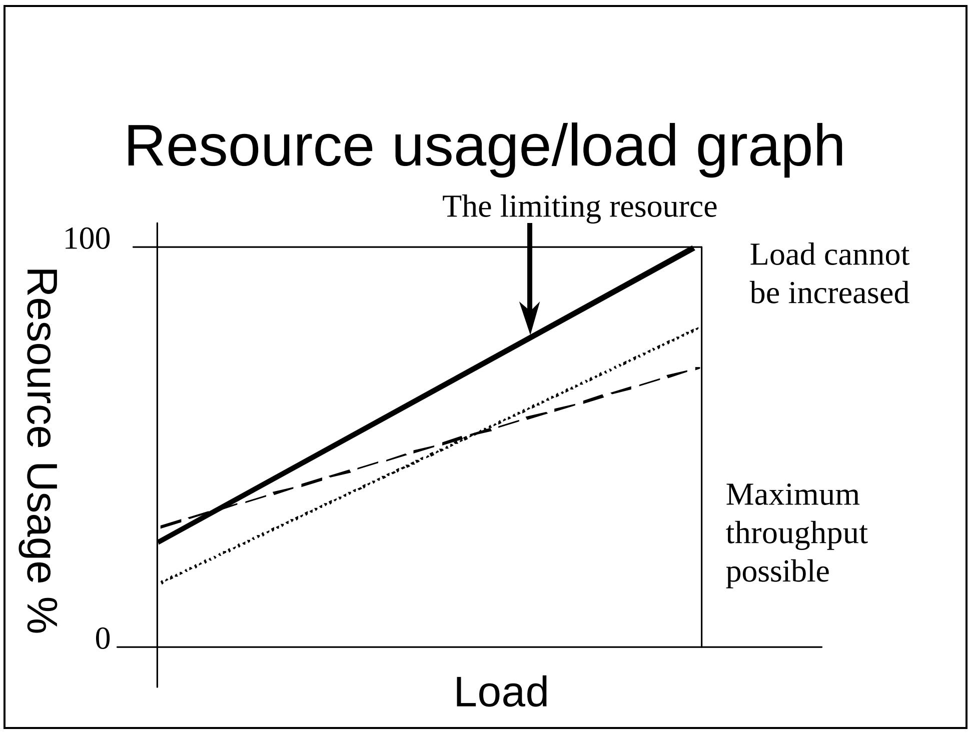

Figure 13.3 Resource Usage/Load Graph

To increase the throughput capacity and/or reduce response times for a system we must do one of the following:

- Reduce the demand for resource. Typically by making the software that uses the resource more efficient. This is usually a development responsibility.
- Optimize the use of the hardware resource within the technical architecture. By configuring the DBMS to cache more data in memory or by prioritising some processes above others on the application server.
- Make more resource available. Normally by adding processors, memory or network bandwidth.

Performance testing normally requires a team of people to help the testers. These are the technical architects, server administrators, networks administrators, developers and database designers/administrators. Only these technical experts are qualified to analyse the statistics generated by the resource monitoring tools and judge how best to adjust the application, tune or upgrade the system. It is essential to involve these experts early in the project to get their advice and commitment and later, during testing, to ensure bottlenecks are identified and resolved.

Performance Testing Objectives

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

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

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 broader objectives that we can summarise as:

- **Assessing 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.
- **Identifying 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 the system** - repeat runs of tests can be performed to verify that tuning activities are having the desired effect - improving performance.
- **Detect obscure bugs in software** - tests executed for extended periods can cause failures caused by memory leaks and reveal obscure contention problems or conflicts. (See the next section for Reliability/Resilience Testing).
- **Verifying 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 next section for Reliability/Resilience Testing).

A comprehensive test strategy would define a test infrastructure to enable all these objectives to be met.

Pre-Requisites for Performance Testing

We can identify five pre-requisites for a performance test. Not all of these need be in place prior to planning or preparing the test (although this might be helpful), but rather, the list below defines what is required before a test can be executed.

Quantitative, Relevant, Measurable, Realistic, Achievable Requirements

As a foundation to all tests, performance objectives, or requirements, should be agreed prior to the test so that a determination of whether the system meets requirements can be made. Requirements for system throughput or response times, in order to be useful as a baseline to compare performance results, should have the following attributes. They must be:

- Expressed in quantifiable terms.
- Relevant to the task a user wants to perform.
- Measurable using a tool (or stopwatch) and at reasonable cost.
- Realistic when compared with the durations of the user task.
- Achievable at reasonable cost.

These attributes are described in more detail in reference [3].

Stable System

A test team attempting to construct a performance test of a system whose software is of poor quality 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(s) crash.

Performance tests stress all architectural components to some degree, but for performance testing to produce useful results, the system infrastructure has to be reasonably reliable and resilient to start with.

Realistic Test Environment

The test environment should ideally be the production environment or a close simulation and be dedicated to the performance test team for the duration of the test. Often this is not possible. However, for the results of the test to be useful, 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 is useless for a performance test.

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. If the system supports load 20% above your design load, you could read this in two ways:

- You have room for growth of 20% beyond your design load or
- 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.

Controlled Test Environment

Performance testers require stability in not only the hardware and software in terms of its reliability and resilience, but also need changes in the environment or software under test to be minimised. Automated scripts are extremely sensitive to changes in the behaviour of the software under test. Test scripts designed to drive client software GUIs are prone 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 in any component of the architecture until they are ready to incorporate changes to their tests. Changes intended to improve performance or the reliability of the environment would normally be accepted as they become available.

Performance Testing Toolkit

The five main tool requirements for our ‘Performance Testing Toolkit’ are summarised here:

- **Test Database Creation/Maintenance** - to create the large volumes of data on the database that will be required for the test. We’d expect this to be an SQL-based utility or perhaps a PC based product like Microsoft Access™, connected to your server-based database.

- **Load generation** – the common tools use test drivers that simulate ‘virtual clients’ by sending HTTP messages to web servers.
- **Application Running Tool** – this 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** - utilities that monitor and log client and server system resources, network traffic, database activity etc.
- **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.

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 mi ght be simple to define from the users point of view, but is unreasonable. Some functions are critical and require short response times, but others are less critical and response time requirements can be less stringent.

Here 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 requesting the system to do something (e.g. clicking on a button) to the system returning control to the user.
- Requirements can vary in criticality according to the different business scenarios envisaged. Sub-second response times are not always appropriate!
- *Generic requirements* are described as ‘catch all’ thresholds. Examples of generic requirements are times to ‘perform a page refresh’, ‘navigate between pages’.
- *Specific requirements* define the requirements for identified system transactions. Examples might be the time ‘to log in’, ‘search for a book by title’ etc.
- Requirements are usually specified in terms of acceptable maximum, average or 95 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 his web site 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 seem to be universally appropriate.

- One tenth of a second (0.1) is the limit for having the user feel the systems is reacting instantaneously.
- One second (1.0) is the limit to allow the user's flow of thought to be uninterrupted.
- Ten seconds (10.0) 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 of your response time requirements or formulate your own, unique, requirements.

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 scenario to be simulated:

- Where the mix of users and their activities is fixed, but the number of users and the size of load varies in each test. These tests reflect our expected or normal usage of the site, but uncertainty is the scale of the load.
- Specific scenarios relating to peak or unusual situations. For example, the response to a successful marketing campaign or special offer, or where an event (that may be outside your control) changes users' behaviour.
- Extreme loads that we might reasonably expect 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 loads of the normal mix, followed by specific scenarios and finally stress tests to 'harden' the architecture.

Requirements Realism

With an internal intranet, we probably know the limits of the load it can experience. 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 website. The calculations are primarily based on the success of marketing campaigns, word of mouth recommendations and in many cases, luck rather than judgement.

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 their technical infrastructure, they may realize how the technology constrains their ambitions. However, they 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 dotcoms, the predicted growth of business processed on their web sites can be grossly overestimated. To acquire 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. However, over ambition dramatically increases the cost of infrastructure required and the cost 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 2000 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:

- 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 ten minutes.
- Most users would access their system between 8am and 12pm.

At our first meeting, we sketched out a quick calculation on a whiteboard and this 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 8am to 12pm, these sessions are served in a 'live' period of seven sixteen-hour days (equal to 126 hours or 7560 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 thirty! 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 table rows that should be present in the database 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.

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 and doesn’t usually affect the functionality of the system under test. A schematic of the test process is presented in Figure 13.4.

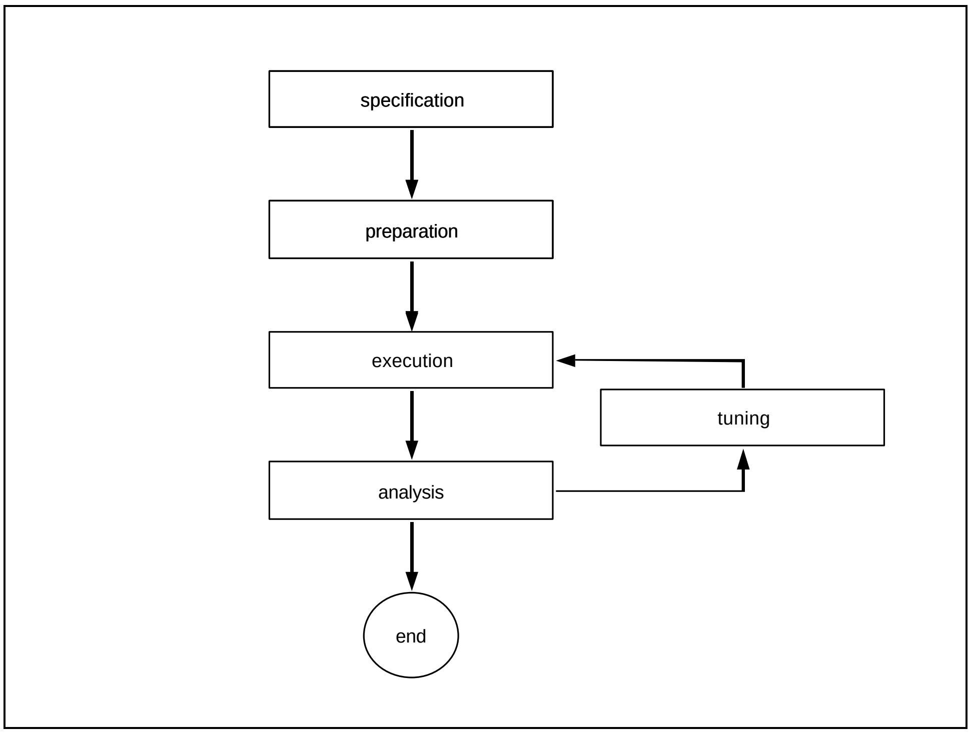

Figure 13.4 Performance Test Process.

The five stages in the process are described in outline in table 13.2.

Specification	▪ Documentation of performance requirements including: ▪ 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, database organization. ▪ Changes to server system parameters. ▪ Upgrades to client or server hardware, network capacity or routing.

Table 13.2 Performance Test Process Outline.

Incremental Test Development

Test development is usually performed incrementally in four stages:

- Each test script is prepared and tested in isolation to debug it.
- Scripts are integrated into the development version of the workload and the workload is executed to test that the new script is compatible.
- As the workload grows, the developing test framework is continually refined, debugged and made more reliable. Experience and familiarity with the tools also grows.
- When the last script is integrated into the workload, the test is executed as a ‘dry run’ to ensure it is completely repeatable and 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 optimised.
- Repeatable test scripts can be run for extended periods as soak tests. 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 co-ordination. As the time approaches to execute the test, team members who will execute the test need to liaise with those who will monitor the test well in advance. The ‘test monitoring’ team members are often working in dispersed locations and need to be kept very well informed if the test is to run smoothly and all results are to be captured correctly. The test monitoring team members 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, pre-process it and make it available to the person who will analyse the data fully and produce the required reports.

Beyond the co-ordination of the various team members, performance test execution tends to follow a standard routine shown below.

1. Preparation of database (restore from tape, if required).
2. Prepare test environment as required and verify its state.
3. Start monitoring processes (network, clients and servers, database).
4. Start the load simulation and observe system monitor(s).
5. If a separate tool is used, when the load is stable, start the application test running tool and response time measurement.
6. Monitor the test closely for the duration of the test.
7. If the test running tools do not stop automatically, terminate the test when the test period ends.
8. Stop monitoring tools and save results.
9. Archive all captured results, and ensure all results data is backed up securely.
10. Produce interim reports; confer with other team members concerning any anomalies.
11. Prepare 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 behaviour, 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 behaviour 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 summarise these measurements and for each measurement taken the following will be reported:

- The 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 percent 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 on 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 simulated - *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 less than planned if the system cannot support the transaction load.

It is common to execute a series of test runs at varying load. Using the results of a series of tests, a graph of response time 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 identification of bottlenecks in a system's architecture.

Which performance test architecture?

There are three options for conducting performance testing. All require automated test tools. The advantages and disadvantages of each are summarised in table 13.1. 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, security infrastructure implemented with production scale Internet connection.

Consideration	***Do It yourself***	***Outsourced***	***Performance Test Portal***
Test tool license	▪ You acquire the licenses for performance test tools	▪ Included in the price	▪ You rent a timeslot on their portal service
Test tool host	▪ You provide	▪ Included in the price	▪ Included in the price
Internet connections	▪ You must organise	▪ Theirs included. You liaise with your own ISP	▪ Theirs included. You liaise with your own ISP
Simplicity	▪ Complicated, you do everything.	▪ Simplest solution – the services provider do it all.	▪ Simple infrastructure/tools solution, but you are responsible for building/running the test
Cost	▪ Potentially very expensive	▪ Lower tool/infrastructure costs, but you pay for the services	▪ Low tool/infrastructure costs
Pros and cons	▪ Could be very expensive. You acquire tools that may rarely be used in the future ▪ You need to organise, through perhaps two ISPs, large network connections (not an issue for intranets) ▪ You are in control. ▪ Complicated – you do everything.	▪ Potentially cheaper, if you would have hired consultants anyway ▪ Simpler, you need one arrangement with your own ISP only. ▪ You can specify the tests, the services company build/execute them, you manage the supplier. ▪ Simplest solution.	▪ Cheapest tool/infrastructure costs, but you still need to buy/acquire skills. ▪ Simplest infrastructure solution, but you must manage/perform the tests. ▪ You are in control. ▪ For majority of sites, a simple solution.

Table 1 Three Performance Test architectures Compared

Practicalities

One serious challenge with E-Business performance testing is the time allowed to plan, prepare, execute and analyse performance tests. How long a period exists between system testing eliminating all but the trivial faults and delivery into production? We normally budget 6-8 weeks to prepare a test on a medium to high complexity environment. How much time will you have?

However, there may be simplifying factors that make testing Web applications easier:

- Firstly, web applications are relatively simple from the point of view of thin clients sending messaged 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 creation of performance test scripts can often be started earlier than traditional client/server applications. Tests using drivers only (and not browsers) may be adequate

Normally, we subject the system under test to load using drivers that simulate real users by sending messages across the network to 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 a small time 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 and the synchronisation between test tool and browser it not always reliable.

If you do decide to ignore the time taken by browsers themselves, be sure to discuss this with your users, technical architect 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 in the project.

Software Quality

In many projects, the time allowed for functional and non-functional testing (including performance testing) is ‘squeezed’. Too little time is allocated overall, and developers often regard the system test period as ‘contingency’. 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 attempt 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 resources on the client, 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 the software errors 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. However, when a single test script is integrated into the complete workload and the full load simulation run, other users of the system will probably be very badly affected by the sudden application of such 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.

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, which usually has to be completed in a very limited timescale. Few project managers have direct experience of the tasks involved in preparing 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.

Over ambition, at least early in the project, is common. Project managers often assume that databases have to be populated with valid data, that every transaction must be incorporated into the load and every response time measured. As usual, the 80/20 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 be measured. Experienced testers would probably assume a 90/10 rule. Inexperienced managers seem to mix up the 90 and the 10.

The skills required to execute automated tests using proprietary tools are not hard to acquire, but as with most software development and testing activities, there are principles that, if adhered to, should allow reasonably competent 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 akin to 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 SUT however, usually need a period to gain familiarity with the system to be tested. Allowance for this should be made as in the early stages of test development; testers will have to grapple with the vagaries of the SUT 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 business rules, for example, reconciliation of financial fields in different tables are not adhered to.

In both cases, the load simulation may not be compromised, but the application may not be able to handle such inconsistencies and fail. In these circumstances, test scripts developed on a small coherent database will no longer work on a prepared production size database. Clearly, it is very helpful for the person preparing the test database to 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. This method will work in most circumstances.

Presenting results to customers and suppliers

On a previous project, we worked as consultants who specified supervised the performance testing. The performance testing took place towards the end of the project, and was going badly. Reported response times were totally acceptable. Our role was to advise both client and supplier, to witness and to review the testing. Fortnightly 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 for the second half of the meeting. The test results were presented to them and quite a lot of pressure applied to impress on them the importance of the current shortcomings. It was something of a fortnightly ritual.

The point to be made here is that 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 then need to defend themselves. Typically, they will suggest the tests are flawed. “The load requirements are excessive”, “the numbers of concurrent users, the transaction rates, the database volumes are all to high”. “The testers aren’t simulating the users correctly”, “real users would never do that”, “this isn’t a good simulation” etc. etc.

Typically, 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 data and be confident that your tests are sound. You must test your tests, and make sure the experts and not yourself do any 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. You have to remember that performance is a major hurdle for the suppliers to get over, and a lot of money may depend on the successful outcome of the performance testing. You have been warned.

Reliability/Resilience Testing

What is availability? Essentially, availability is measured in terms of uptime – the proportion of time that a Web-based service is available for use divided by the total time in a measured period. In mainframe installations, uptime might be as high as 99.99% over a period of one year. That sounds incredibly high, doesn’t it? 99.99% availability amounts to around 9 hours downtime in the year, where a system runs 24 hours per day, 365 days per year. Assuming all systems have do be down for short periods to perform maintenance activities, this is really quite remarkable.

What might you expect of an E-Commerce site? You would wish the shop to be open

24 hours per day, 365 days per year, wouldn't you? The objective of all Web sites is to be continuously available. Availability testing aims to evaluate a site's ability to stay up for a long time. Does that mean tests should span extremely long timescales? This isn't feasible for most projects of course, but we could re-use some or all of the automated Performance Tests that if they are available. If a performance test of the technical infrastructure exists, we would of course use those. But if performance testing of the entire application is done late, we might only have a few days just before 'go live' to conduct these tests.

Failover Testing

Where sites are required to be resilient and/or reliable, 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 backups, software and hardware upgrades and other maintenance activities to be performed (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 behaviour is seen for the first time. Failover testing aims to explore the behavior of the system under selected failure scenarios and normally involves the following:

- Identification of the components that could fail and cause a loss of service.
- An analysis of the failure modes or scenarios that could occur where you need confidence that the recovery measure will work.
- An automated test that can be used to load the system and explore the behavior of the system over an extended period.
- The same automated test that can be used to load the system under test and monitor the behaviour of the system 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.

Fault tree analysis and fault tree diagrams are a logical representation of a system or service and the ways in which it can fail. Figure 13.5 shows the relationship between basic component failure events, intermediate sub-system 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. However, if we had two web servers and facilities to load balance 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 fail simultaneously.

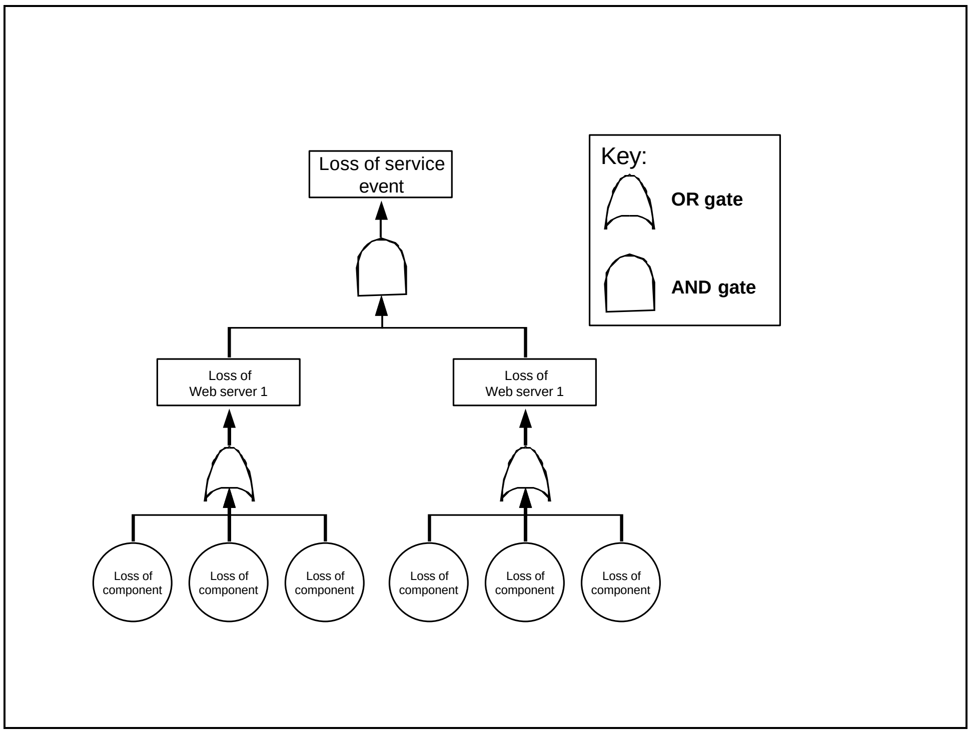

Figure 13.5 Failure Event Heirarchy

FTA documents the Boolean relationships between lower level events to those above them and a complete fault tree documents the permutations of failures that will cause failure, but also should still allow a service to be provided. Figure 13.6 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.

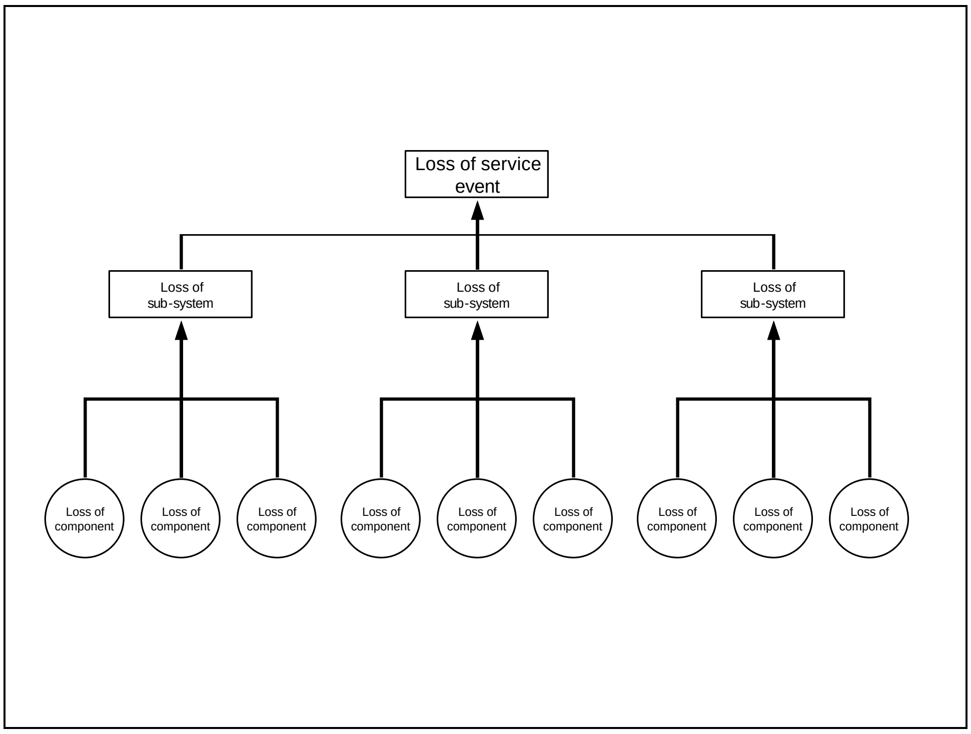

Figure 13.6 Example Fault tree for a Web service.

FTA is most helpful for the technical architect who is designing the mechanisms for coping with failures. Fault tree diagrams are sometimes complicated with many Boolean symbols representing the dependencies of services on their sub-systems and the dependencies of sub-systems or their 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 by discussing these with the technical experts, testing follows a straightforward, bottom-up approach. With an automated test running, the tester explores a range of failure modes. Early tests focus on individual component failures (e.g. you offline a disk, power down a server, break a network connection etc.) 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:

- 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 the concern over the system 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 maybe 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 to simulate three main scenarios:

- Extreme loads that subject specific components or resources in the technical architecture to large numbers of concurrent demands.
- Extended periods of normal (or extreme) loads.

When focusing our reliability on specific components, we are looking to stress that component by subjecting it to an unreasonably large number of requests to perform its designed function. For example, if a single (non-redundant) 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). However, by testing it in isolation, we can 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 to 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 are tests that subject a system to a load over an extended period of perhaps 24, 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 one 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). But we are particularly interested in the system's ability to withstand continuous running of as wide a variety of test transactions to find if there are any obscure memory leaks, locking or race conditions. As for performance testing, monitoring the resources being 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 increasing resources being used up over time and not being replaced. Where memory is allocated to systems and not returned, this 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 detect 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 it to be monitored, upgraded, backed up and fixed quickly when things go wrong. Post-Deployment Monitoring is covered in chapter 17. The procedures that web site managers use to perform upgrades, backups, releases and restorations from failures are critical to providing a reliable service so need testing, particularly if the site is will undergo rapid change after deployment.

The management procedures fall into five broad categories:

- System shutdown and start-up procedures.
- Server, network and software infrastructure and application code installation and upgrades.
- Server, network and software infrastructure configuration and security changes.
- Normal system backups.
- 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.

Substantially, 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:

- 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, as far as possible be run in as realistic 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 defines the system management procedures to be performed and 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.

(Performance Testing) Tools for Service Testing

As can be seen in the table below, there are a large number of tools available for performance testing. Most of the proprietary tools run on the Windows platform and a smaller number run under Unix. These tools all have GUI front-ends, sophisticated analysis and graphing capabilities. They tend to be easier to use.

Of the Free tools, the vast majority run under Unix and many are written in Perl. They tend to be less sophisticated lack powerful scripting languages. 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 you have a technical or programmer background. However, they are free 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 a good reason for choosing a free tool. However, even if you have a budget to spend, 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 experience with, and then select a proprietary tool with more confidence because you have more experience of using a tool.

Proprietary Tools	**URL**
Astra LoadTest	www-svca.mercuryinteractive.com
Atesto Internet Loadmodeller	www.atesto.com
CapCal	www.capcal.com
e-Load, Bean-Test	www.empirix.com
eValid	www.soft.com
forecastweb	www.facilita.co.uk
OpenSTA	www.opensta.org
Portent	www.loadtesting.com
PureLoad	www.pureload.com
QAload, EcoTOOLS, File-AID/CS	www.compuware.com
silkperformer	www.segue.com
SiteLoad	www.rational.com
SiteTools Loader, SiteTools Monitor	www.softlight.com
Test Perspective	www.keynote.com
Web Performance Trainer	www.webperformanceinc.com
Web Server Stress Tool	www.paessler.com/tools/
WebART	www.oclc.org/webart
WebAvalanche, WebReflector	www.cawnetworks.com
WebLOAD	www.radview.com
Web Polygraph	www.web-polygraph.org
WebSizr	www.technovations.com
WebSpray	www.redhillnetworks.com
Free Tools	**URL**
ApacheBench	www.cpan.org/modules/by-module/HTTPD/
Deluge	deluge.sourceforge.net
DieselTest	sourceforge.net/projects/dieseltest/
Hammerhead 2	hammerhead.sourceforge.net
http_load	www.acme.com/software/http_load/
InetMonitor	www.microsoft.com/siteserver/site/DeployAdmin/ InetMonitor.htm
Load	www.pushtotest.com
OpenLoad	openload.sourceforge.net
Siege	www.joedog.org
SSL Stress Tool	sslclient.sourceforge.net
Torture	stein.cshl.org/~lstein/torture/
Velometer	www.velometer.com
Wcat	www.microsoft.com/downloads/
Web Application Stress (WAS) Tool	webtool.rte.microsoft.com
WTest	members.attcanada.ca/~bibhasb/

References

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

1990.

[2] Menascé D. A., Almeida V. A. F., Scaling for E-Business, Prentice Hall PTR, NJ, 2000.

[3] Gerrard P., *Client/Server Performance Testing*, www.evolutif.co.uk 1995.

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

[5] Nguyen H. Q., *Testing Applications on the Web*, Wiley, NY, 2001

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

[7] Kolish T., Doyle T., Gain Econfidence, Segue Software, MA, 1999.

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

[9] Leveson N. G., *Safeware*, Addison Welsey, MA, 1995.