



Heavy load

Gordon McKeown, Technical Director at Facilita, with some cautionary tales on common serious load testing pitfalls

Tester B has recently taken over from Tester A. Phase 2 of the testing is about to start and the number of virtual (simulated) web users is being increased from 500 to 3,000. Worried (unnecessarily) that the load generation hardware might not cope, Tester B examines the scripts for the testing tool. After each and every page download Tester A has inserted a check on the contents of the page returned. Noting that the testing tool automatically checks the HTTP return code, Tester B decides that these checks are an unnecessary overhead and removes them. The first 3,000 user run reports some poor response times. The development team are notified and implement some changes. The test is repeated. The results are astounding! All transactions were completed rapidly. The good news is e-mailed to interested parties. A puzzled reply is received. The back-end mainframe service had in fact been shut down for maintenance during the test. The test is re-run with detailed logging switched on. Every request had returned with a 200 (OK) HTTP code and a page containing the message "Error! The service is not available."

What lessons are to be learned from this story? Several spring to mind. First there are the classic software development sayings, "If it ain't broke..." and, "Do not engage in premature optimisation". Most importantly: load testers must thoroughly understand the target application and technology.

Defeated before you start?

"Of course we must load test! These modern tools are very productive so a week or so should be enough to get the scripts working. Allow a couple of days to set up the test system and a week to run the tests. We should add some contingency time. The go-live date is the 28th so we'll schedule the tests to start on the 8th..."

If reasoning like this is used it's a very safe bet that the go-live date will slip. Load testing, like many virtuous activities, is better carried out late than never. However, it is invariably better carried out earlier than later. The key point is to assume that problems will be identified by load testing and that they will need to be rectified. Once changes are made the tests

must be repeated. Major issues have been identified with almost every significant system that we have been involved in testing. Any realistic plan must presume re-working, fine-tuning or making changes to the hardware specification as a result of testing.

Who?

Assembling the right people to both carry out and support the testing is vital. A cohesive multi-disciplinary team that covers the end-to-end technology is the ideal. Not every member will be full-time, but they should be engaged and aware of developments. Remember that behind that innocent web site there may be layers of complex technology including multi-tier client server, databases, J2EE, .NET, transaction processing and mainframes. All experienced testers will remember the frustrations and delays that occur when the CICS system or back-end database hits a problem and there is no pre-arranged specialist support. The same applies to identifying bottlenecks and tuning: the key specialists need to be involved with the testing.

An indicator of possible failure is the absence of the system architect/chief designer from the load testing process. When testing a complex system it is often hard to diagnose problems and bottlenecks. The architect, assuming the name means anything, or other technical guru who understands the system as a whole, has a vital role; ideally they should be crawling over all aspects of the testing. Often they have moved on, had their contract terminated as they are expensive and after all "the system has been completed", or never existed in the first place. If you are charged with testing a complex enterprise system in the absence of such a person then prepare for storms ahead.

What of the testers themselves? It's a common complaint that some managers believe testing can be largely de-skilled. Of course nobody will admit to this attitude in so many words. To some extent the fallacy of de-skilling is encouraged by the hype of (some) tool vendors. Intelligence, experience and other attributes such as tenacity cannot be substituted. Knowledge of the internals of a system is unnecessary when performing functional testing at the user interface; sometimes it can even be a liability. The same does not

follow for load testing. The tests themselves are a "black box", in that a load testing tool generates network traffic, or makes remote calls to the system under test. However system configuration, trouble shooting and the analysis of the test results all require an understanding of the internal components and implementation technologies.

Tools can help

It is sometimes a mistake to purchase a tool that can only test web sites because that is how the user interface is being implemented. Early testing of components may be appropriate before the user interface has been created so a tool that can test eg EJBs, .NET remoting or databases directly, in addition to web testing, would be required.

Tool support for multiple technologies proved extremely useful for one of our customers. The system to be tested was a modern web-based interface to a mainframe-based legacy system. Disaster struck the testing schedule when it transpired that the test mainframe service was not available. A "mainframe simulator" was hurriedly deployed. This consisted of another instance of the load test tool with scripts that played the role of the missing mainframe by receiving and servicing incoming connection requests and providing valid responses to client messages. Not only was tool support for network level data transfer required, but also the unusual capability of handling incoming connection requests; most load testing tools only simulate clients which typically initiate connections.

The day of reckoning

You discuss and agree a test plan that allows a day for checking the test environment by running through the test scripts at a low intensity (say five virtual users). A series of tests simulating up to 1,000 users are then run to check that performance targets will be met on the end hardware. One week in and you are still running tests at the five user level, senior management is restless and a crisis is at hand. This scenario occurs in one form or another when there has been inadequate testing prior to the final load testing phase; frequent iterative load testing should avoid it. Unfortunately, too frequently, the final load testing phase is where the project "chickens come home to

Helping you complete your software development puzzle

Software Configuration Management

Load Testing

Defect Tracking

Regression Testing

Industry leading software at an affordable price



T +44 (0) 1344 297600 W www.contemporary.co.uk E info@contemporary.co.uk

roost". In these circumstances the first challenge is political. The load testing team have to convey the reality of the situation to management. What has been labelled "load testing for capacity planning" has turned into "multi-user functional testing" plus "integration testing". Failure to get the message across may result in the load testing team and (the load testing tools) getting unfairly blamed.

What exactly are we supposed to be testing?

Creating the test configuration is itself a task that is full of pitfalls. Pre-deployment testing may be possible on the precise hardware that will be used 'live', but there still remains the issue of the network. In other cases, where the new application is sharing resources with an existing one or where testing is targeted at a new version of an existing application then different test hardware is often used. This is because conducting load testing on a live system may not be practical or advisable. Clearly, an exact replica of the live system is the ideal; any deviations must be carefully analysed.

Although access to the live system may be possible "out of hours" the process of load testing is risky in that it may involve numerous system changes. It is also desirable that the load testing team have complete control over the target configuration during the process. In most circumstances testing therefore should be conducted using dedicated hardware. Should it also be isolated? Again the issue of control and repeatability points to using an isolated network. The clincher in many cases is the issue of security as load testing tools and systems do pose a potential threat if connected to the corporate network. However, what if the corporate network itself is legitimately part of the testing target? All this underlines the importance of agreeing the scope of the tests precisely.

"A. Customer, 123 High St., Anytown..."

Teams that avoid the obvious pitfalls and generally follow good practices sometimes trip up when it comes to test data. There are two closely related issues: varying data input to the system under test and the set up or population

of any database used by the server side of the application. Clearly these must be synchronised as, for instance, the customer identifier input to open an existing account must match a record on the database.

The first problem is identifying which data needs to be varied in the scripts, as opposed to using literal values, ie those logged if record/replay is used. This process is sometimes called *parameterisation* which is an ugly word but we do need to give it a name. There are two motivations for parameterisation: because the application requires it and in order to exercise the SUT properly. The test tool may well help by automating certain aspects, but unfortunately human judgment is still required. Within the general category of application-required, variable data are such things as session identifiers, temporary keys and timestamps. These relate to the maintenance of what is often called "conversation state" between the client and the server and avoiding breaking business rules (eg a user can only be logged on to the system once at any time). With a mixture of run-time intelligence, for example cookie handling for web testing, plus script generation techniques, the load testing tool can handle a lot of this automatically. In any case problems in this area are usually obvious: the test fails! More insidious is inadequate discretionary parameterisation. As a simple example imagine testing an on-line shop. If no parameterisation is carried out then each virtual user would browse the same items, details of which would be cached so the response times would be misleading. Buying the same items would also be misleading, possibly with over long responses due to excessive database locking.

The performance of many systems is highly dependent on the test data chosen, both to populate the test database and as input data. The data should closely correspond to reality. Even something apparently innocuous and convenient like giving test customers names of the form "Customer0000" to "Customer9999" may bias the results depending on how database tables are indexed.

Ideally the test database should resemble the live database as closely as possible. Having the ability to restore a database to a known state is a good thing as it allows repeated tests that are directly comparable.

If a test database needs to be created then do not stint on getting both the size and "shape" (data characteristics) right. In some cases it is convenient to use the load testing tool to populate the database by driving the application. On occasion this technique has to be used because there are very complex business rules within the application and it is simply too hard to work out the SQL statements to create valid entries!

Having too few records in a test database is the most common mistake. This can play havoc with the measured response times. You would expect them to be too short as the data will always be cached in RAM and sometimes this is indeed the case. Sometimes the opposite occurs. Tests of an application that accessed a relational database via EJBs were carried out with a small test database. The response times were appalling. Further analysis revealed that excessive locking was occurring. The database had been configured for page locking rather than row locking and the probability of simultaneous hits on a page were very high due to the small amount of data. This unforeseen problem was in fact serendipitous. The actual database was due to start small and to grow so it was decided to start with row locking and to switch to page locking later.

Do not despair!

Load testing is subject to the law of diminishing returns. This is both good and bad. It means that even a little bit of testing is extremely useful. Gross errors and issues are often exposed in the first hour. A potential pitfall is to test too much. I have to say that I have never seen this occur in practice. The opposite fault, too little testing, is common. Get load testing embedded into your process, get the right people and the right tools, and give them enough time.

PT