

Increasing Performance Testing Effectiveness

Techniques for Reducing Test Cycle Time and Increasing Value per Test Cycle

By Richard Leeke, Equinox Ltd

Performance tests invariably seem to be on the critical path, at the compressed “right hand end” of the timeline. All too often they form one of the final steps in a project, when all software and hardware components have finally been integrated. But even with a more mature development process including ongoing performance testing throughout the course of a project or with ongoing “business as usual” performance testing of enhancements, the time pressure to complete the performance testing exercise always seems to be intense.

However, the tools and techniques used to conduct performance tests often do not lend themselves to a rapid turn-around time for testing. Even minor changes in the user interface can completely invalidate recorded test scripts, requiring a time-consuming, labour-intensive and error-prone process of re-recording and reapplying manual changes to the test scripts. Conducting the actual testing also often requires a long series of tests, with different workload parameters, in order to provide the required answers with any confidence.

Too much of the elapsed time of the exercise ends up being spent on the mechanics of the testing; too little on the really high-value work of identifying, diagnosing and resolving issues. But customers aren’t interested in the actual testing – they just want to know that the system is going to work. And in a competitive environment where time to market is crucial, it is easy to see why customers can become frustrated with the time taken.

The following quote from a test strategy document written by one of my colleagues, Derek Mead, sums up the challenges this presents.

A performance testing process risks becoming irrelevant if it is not agile enough to meet the business’ needs for rapid deployment of responsive but stable systems: if the process is not sufficiently responsive to meet the business needs there is a risk that the business will decide to deploy untested rather than wait for test results.

My Bag of Tricks

Over the years I have built up a collection of tools and techniques which help reduce the time taken to customise performance testing scripts and reapply customisations to the scripts as changes to the application under test force the scripts to be re-recorded. Some of the techniques also help make the testing process itself more effective, for example by:

- providing early indications of test results (which may allow a failed run to be cancelled without waiting for the full run to complete),
- allowing the test workload to be adjusted during the course of a test run (effectively compressing multiple test runs into a single longer run),
- assisting with the analysis of test results and diagnosis of issues.

A kitbag of home grown tools is nothing new, of course; we all have them, and tend to take them somewhat for granted. But after a couple of testing engagements recently I found

myself reflecting on just how much time some of these tools and techniques really did save – and how valuable it would be to see equivalent functionality in the commercial tools I use.

In this paper I describe a few of the techniques which I find particularly effective, illustrating their use by way of a case-study of a recent performance testing exercise (which at the time felt like “the performance testing engagement from hell”), during which minimising the time spent on the mechanics of the testing process was absolutely essential.

The Assignment

The assignment was for an overseas office of a large multi-national insurance company. The company was in the process of rolling out a new, browser-based quoting and new business commercial insurance application to its brokers. The brokers accessed the system via a private network. As rival companies’ products were already available via a similar delivery channel, the competitive pressure to complete the rollout was intense.

However, by the time the system had been rolled out to the first 40 or so brokers (out of a total user base of around 2,000), the performance of the system had degenerated to the point where the rollout was placed on hold. The brokers who had tried the system were threatening not to use it and on at least one occasion the system had failed completely.

The application was built using a highly flexible (which is always a warning sign!) pricing and configuration package. The package, which was written in Java, allowed the structure of products and service offerings together with their pricing rules to be defined in a “knowledgebase”. The application’s user interface was dynamically generated, based on knowledgebase settings, by a middle-tier application server. Multiple instances of the application server in a load-balanced configuration provided horizontal scalability and redundancy.

We had previously completed extensive performance testing and tuning exercises for a couple of other implementations of this package (though not in the insurance industry), and were approached by the package vendor to assist.

The objectives of the assignment were agreed as follows:

- Isolate the immediate issues.
- Recommend fixes.
- Estimate the capacity of the existing production infrastructure.

Approach

The performance and stability issues had not been apparent during development and functional testing. Based on our previous experiences with this package, we knew that there could well be subtle, multi-user issues, rather than this being a simple question of capacity.

The approach we recommended was for the client to build a dedicated and isolated test environment in which we would use a performance testing tool to develop a set of test scripts to allow us to attempt to reproduce the symptoms seen in production – reproducing the problem being the essential first step.

We were guaranteed full access to both the client and vendor architects for the system – which we viewed as critical for any chance of success.

As this was a diagnostic and tuning exercise, as opposed to a full performance test, we did not spend long attempting to define a realistic workload model. We simply built scripts to

exercise an end-to-end transaction for a typical, common business scenario (a new policy for a single location, providing cover for three types of risk), and used our judgement and brief client feedback to come up with “reasonable” pacing for the scripts. The general consensus was that the scenario chosen would exercise of the order of 70% of the functionality of the system. If we had been unable to reproduce the issues in this way we would then have extended the test scripts to provide fuller coverage – for example by including further risk types and also back-office functions. As it turned out, this approach turned up plenty of issues to be going on with.

We used Rational Performance Tester for applying the load and a wide variety of tools for monitoring and diagnosis, including:

- Windows performance monitor.
- HPROF (for profiling Java – only viable for single-user tests).
- The Java debugging interface (to capture JVM stack trace samples during multi-user tests to allow simple statistical profiling).
- A database monitoring tool (Zero Impact) to sniff the traffic to the back-end database server.

Challenges

From our previous experience we knew that the architecture of the application under test made it particularly resistant to performance testing. I am a great advocate of ensuring that testability ranks high on the list of system qualities when an application is being designed. This one seemed to have been designed for *un*-testability, if there is such a word.

The basic issue stemmed from the fact that the *names* of many of the objects such as form fields in the application’s HTML were dynamically generated by the middle-tier application server. Any given instance of the application server would continue to use the same name for a form field until it was shut down, but the next time it was started it might use a different name. Likewise, with load balanced servers, the name used for a field would depend which server happened to service the request.

HTML test scripts of course rely on the field names, so whenever something caused field names to change, the test scripts would break. Performance testing scripts tend to have a short-enough shelf-life at the best of times. These ones needed to be kept in the deep freeze.

In our previous dealings with the product we had found workarounds for these issues, but only for as long as nothing changed in the knowledgebase or the application code. Even a simple change like selecting a different logging level on the server was still enough to change the field names, however.

Once the scripts had been broken by a change it was not viable to correct them by hand – there were simply too many fields. Even the scripts for the simple new business transaction which we used for the tuning had over 200 objects for which the Rational tool generated automated data correlation code, and another 200 for which values were taken from a “datapool”. The only option was to re-record the scripts and reapply customisations.

One further issue came to light after we had agreed to undertake the exercise and I had provided the client with a hardware specification for the platform to run the Rational tool. My colleague, who had done the bulk of the previous work on other implementations of this package reviewed the hardware specifications and said that he simply did not believe it was possible to run 50 to 100 virtual testers on a single, high-end Windows desktop, with this application – he felt it would be CPU constrained at 10 to 20 virtual testers. The target of 50

to 100 virtual testers I had agreed with the client had seemed entirely manageable from a single test driver workstation to me, based on experience with other applications – though I had encountered occasional exceptions. In the previous tests with this package, I knew that we had been using a high end Solaris server to drive the virtual testers, but I hadn't realised we had been using nearly all of the capacity of the server. Checking back with the client for the current exercise, they were not going to be able to source additional agent platforms in the time available, so we realised we needed to work out where all of the CPU was going.

We also had very limited time for the exercise, constrained by client pressure due to the critical business impact of the problem, and a maximum window of two weeks which we could fit in between existing client commitments.

When we arrived on site, we found that the test environment was still a bunch of PCs with little more than the operating system installed. The architects who had been “dedicated” to the exercise were also now only available part time, having been reassigned, since the client had decided on a hardware solution to the problem.

When we finally got the entire environment going somewhere in the middle of day three, we recorded a trial test script, only to find that something in the application didn't agree with the Rational test script generator, resulting in a nasty corruption of the generated script - such that it wouldn't even compile. I seem to recall posting rather more questions than usual on the performance testing forums that week.

Naturally there were plenty of other challenges, too. All of which combined to make it essential that the mechanics of the process of applying the load took as little of our time as possible, in order to leave any time at all for the hard part – diagnosing what was going on.

Overview of Tools and Techniques Used

In the course of this assignment we used a wide variety of things from the “bag of tricks”, not to mention adding a couple of new ones. The main ones are outlined below.

Framework for Rational Performance Testing

I have a standard framework I use for projects which use Rational Performance Tester. Although this is very specific to the Rational tool, I have used several of the same ideas successfully with other commercial tools or with specific, proprietary test harnesses. The main features of the framework which we used on this exercise are as follows.

- A spreadsheet for expressing aspects of the workload model in a format that can be easily reviewed by users. This deals with both the low level pacing (think times for each user interaction) and higher level pacing (the rate of user/customer transactions).
- A Perl script for applying transaction pacing and timing from the spreadsheet into the correct points in the recorded test scripts. This makes re-recording test scripts after application changes much less onerous.
- Support for “start to start” or “end to start” delays (which are useful for modelling different types of workloads).
- Randomisation of “think time” delays.
- Randomisation of “transaction arrival rate” pacing.
- The ability to scale transaction pacing dynamically during a test run.
- The ability to vary the number of active virtual users dynamically during a test run.
- Interactive display of individual page response times during a test run.
- The ability to monitor average end-user response times, page hit rate and transaction throughput interactively during a test run.

- The ability to automate the “ramp-up” of user numbers and/or vary the pacing model dynamically, to allow a single test run to measure multiple workloads.
- Logging of all dynamically calculated or controlled data for subsequent analysis.
- The ability to generate multiple timer names per user interaction. This is very useful for differentiating between users with different emulated line speeds, or other different characteristics, performing the same transaction.
- Visualisation of the HTML pages seen at recording time and playback time (useful for quickly diagnosing playback issues – since the Rational tool does not directly allow the pages returned to be viewed).

The Answer to that Test Platform CPU Capacity Issue

The weekend before arriving on site, I got to the bottom of the issue which was causing such high CPU utilisation on the test driver platform with this application. It has to do with the mechanism used by the Rational tool for automatically correlating variables in the test script at runtime. The high number of HTML variables per page in this application meant that parsing the returned pages at run-time for each variable rapidly became CPU constrained.

So I made use of the time on the overseas flight by writing a Perl script to change the Rational test scripts to do it more efficiently. (It takes a long time to fly anywhere from New Zealand.)

Java Profiling

Working out where all the time is going is never easy – especially when the behaviour only manifests itself under load. We used two techniques for profiling the JVMs that were running the various components of the system.

The Java JDK includes tools which allow a full trace of code execution. However, the overhead of comprehensive profiling is such that it simply isn’t viable for diagnosing issues which only manifest themselves under load. We used this approach to gain a basic understanding of the behaviour of the system and investigate some issues which were not load dependent.

A technique I have used in other environments is to capture stack traces periodically to generate a statistical profile of application behaviour. We couldn’t find any readily available statistical profiling tools for Java, so during the course of the exercise, my colleague wrote a simple piece of Javascript to do this, using the Java Debugging Interface (JDI). Another one for the bag of tricks.

Database Monitoring

We used a third party database monitoring tool to capture full details of all SQL sent to the back-end server during the tests. The tool we used, Zero Impact from SQL Power Tools, works by sniffing the network traffic to the database server and extracting details of SQL statements sent, together with timings and data volumes for the corresponding responses. Analysing these logs provides a very effective means of prioritising database tuning efforts – by focussing on those SQL statements contributing the most overall load on the server.

We initially used this to monitor the database traffic generated on our test database server by our test scripts, but the technique proved so effective that we also persuaded the client to allow us to sniff the traffic against the production server. The fact that the tool is demonstrably non-intrusive (it can be used from a read-only switch port spanning the database server port), meant that we were able to persuade the normally paranoid network administrators to permit a laptop into the production data centre.

Automating the Manual Script Changes

All of the performance testing tools I have used require at least some manual editing of recorded test scripts before being able to run a meaningful test. In this case, the changes we needed to make to a newly recorded set of scripts were as follows:

- Inserting logic to apply the required pacing and meaningful transaction timers.
- Optimising the test scripts to reduce the CPU overhead.
- Inserting specific validation code to detect certain classes of stress-related application errors.
- Fixing-up the corruptions in the test scripts due to the Rational script generation bug.

The first three were done automatically with Perl scripts. The manual fix-up required 5 or 10 minutes of editing. By the end of the exercise I had the process for recording and customising the test script (which included 35 timed user interactions) down to about 25 minutes.

Pacing and Timing the Tests

The technique I use for applying pacing and timing logic to the test scripts is essentially very simple (based on some code my colleague Derek Mead wrote during our very first HTTP performance test - although over the years I've managed to complicate it a bit!). Some of the capabilities are essentially equivalent to Rational's "Transactor" mechanism (which wasn't around when we started doing this).

All of the interesting stuff happens at the points of user interaction. Once a response is received from the server, we want the test script to end a named timer for the previous operation, delay for a specified (but randomised) think time interval and then start a timer for the next transaction, before proceeding. A call to a single routine takes care of all of that, following each interaction with the server.

During script recording, I insert comments of a standard format after each interaction, identifying the name of each transaction step. A spreadsheet with the required timer name and average delay for each transaction step generates the required calls to the timing routine and a Perl script puts the calls into the code.

Decoupling the definition of the pacing from the test script itself helps to minimise the effort involved in re-recording scripts following application change. This was particularly important for this exercise, given the degree of churn of test scripts we experienced.

Having now got all of the pacing and timing logic neatly encapsulated like this, various other things become possible, mostly by making use of the Rational "shared variable" mechanism.

- Think time delays can be scaled by multiplying by a shared variable which can be set manually (or by a controlling test script) during a test run.
- Individual response times can be displayed for each transaction step as the test proceeds.
- An overall rolling average response time and transaction throughput can be calculated and displayed via shared variables.
- The number of virtual users actively participating in the test can be varied by setting a shared variable (to be checked during each call to the pacing routine).

Various other ideas are on the "to do" list to be included. For example, Scott Barber's approach to "accounting for abandonment" could be incorporated very easily, as could validating individual response times against per-transaction response time criteria. Even simple "goal based" tests should be possible (e.g. "find the maximum number of users for

which the average response time remains below X”) – although I suspect the practicalities might prove a bit challenging in this case.

The framework supports two types of timing delays, and also differentiates between two “levels” of timers.

Delays can be “start to start” (which simulates a situation where the arrival rate of new requests is independent of the system response) or “end to start” (for cases where the user’s next action is dependent on receipt of the previous response).

Timers can be “transaction level” or “step level”. A transaction level timer, which can span multiple user interactions, represents a business transaction. The pacing of this level of timer, which may be “start to start” or “end to start” depending on the situation being modelled, controls the overall transaction rate of the test. A step level timer spans a single interaction with the system. Although the framework supports both, these are generally “end to start”.

Examples of Use

A few samples of the results from the exercise I have been describing illustrate the way that this all works.

Workload Model

An extract from the spreadsheet defining the pacing for the test is shown below.

Txn ID	Timer Name	Level	Delay	Type	Command
Stagger	Stagger Start	Step	0	end-start	Stagger wt("Stagger Start", 0, 0)
Txn Login	Login	Txn	0	end-start	Txn Login tt("Login", 0, 0)
Start	Login Page	Step	5	end-start	Start wt("Login Page", 5, 0)
Login	Submit Login	Step	5	end-start	Login wt("Submit Login", 5, 0)
end	end	Step	0	end-start	end wt("end", 0, 0)
Txn NB	New Business	Txn	0	end-start	Txn NB tt("New Business", 0, 0)
NB	Select New Business	Step	20	end-start	NB wt("Select New Business", 20, 0)
end	end	Step	0	end-start	end wt("end", 0, 0)
Txn <App>	<App> Transaction	Txn	0	end-start	Txn <App> tt("<App> Transaction", 0, 0)
<Appname>	Launch <Appname>	Step	5	end-start	<Appname> wt("Launch <Appname>", 5, 0)
AC	Select Account	Step	5	end-start	AC wt("Select Account", 5, 0)
Suburb	Select Suburb	Step	5	end-start	Suburb wt("Select Suburb", 5, 0)
GUQ	Underwriting Questions	Step	10	end-start	GUQ wt("Underwriting Questions", 10, 0)
...					

The *Txn ID* column lists the tags inserted as comments into the test script. The *Timer Name*, *Level*, *Delay* and *Type* columns define the details for each call to the pacing routines, and the *Command* column provides the actual procedure call to be inserted by the Perl script.

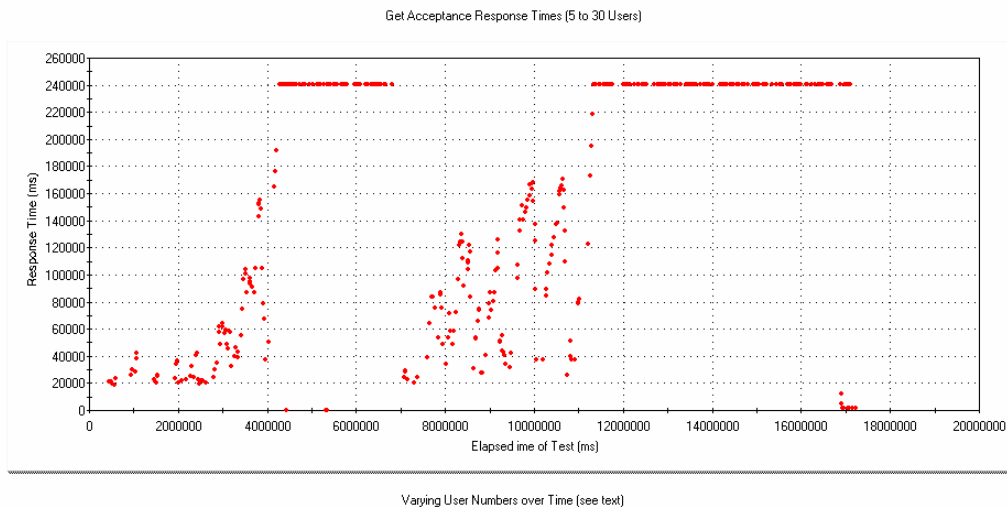
(The *Stagger Start* step is another example of things that the framework handles automatically, by the way.)

Dynamically Varying the Workload

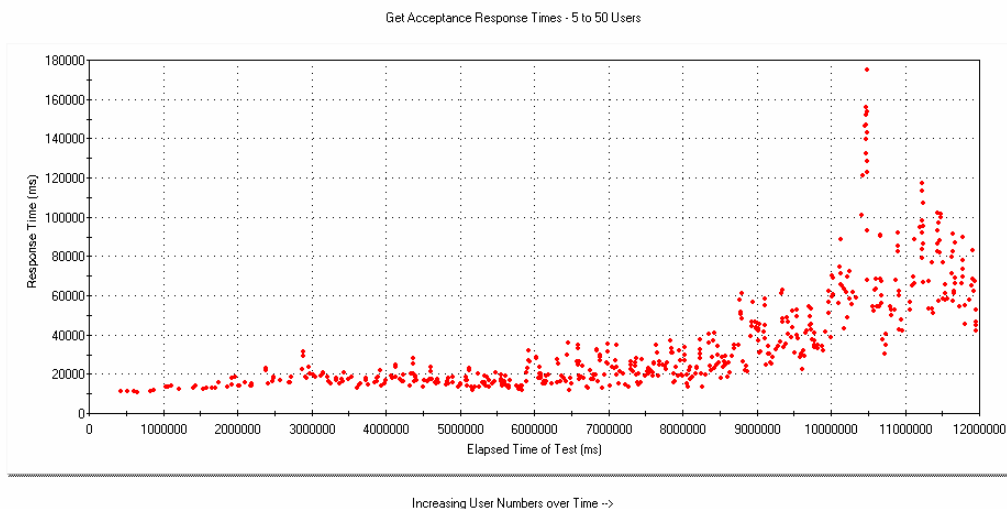
The first scatter chart below shows response times for the most intensive step in the process (acceptance of the transaction) over the course of an exploratory run lasting a few hours, as the number of active users was varied. This was from a test run at the start of the exercise.

A little over an hour into the test I increased the number of active users to 30, at which point response times started to exceed the default 4 minute timeout period set in the test tool and all transactions began to fail (first horizontal line on the chart).

So I reduced the number of active users to five for a while, to see if the server recovered. Once it had, and response times had returned to a “normal” level (around 20 seconds for this step), I increased the load again, only this time more gradually. At about two and a half hours into the run I set the number of active users to 20. Although response times peaked at around three minutes a couple of times this seemed to be just about a sustainable level, so I left some diagnostic information being captured and went to lunch. But it wasn’t stable, and by the time I got back everything was failing once again.



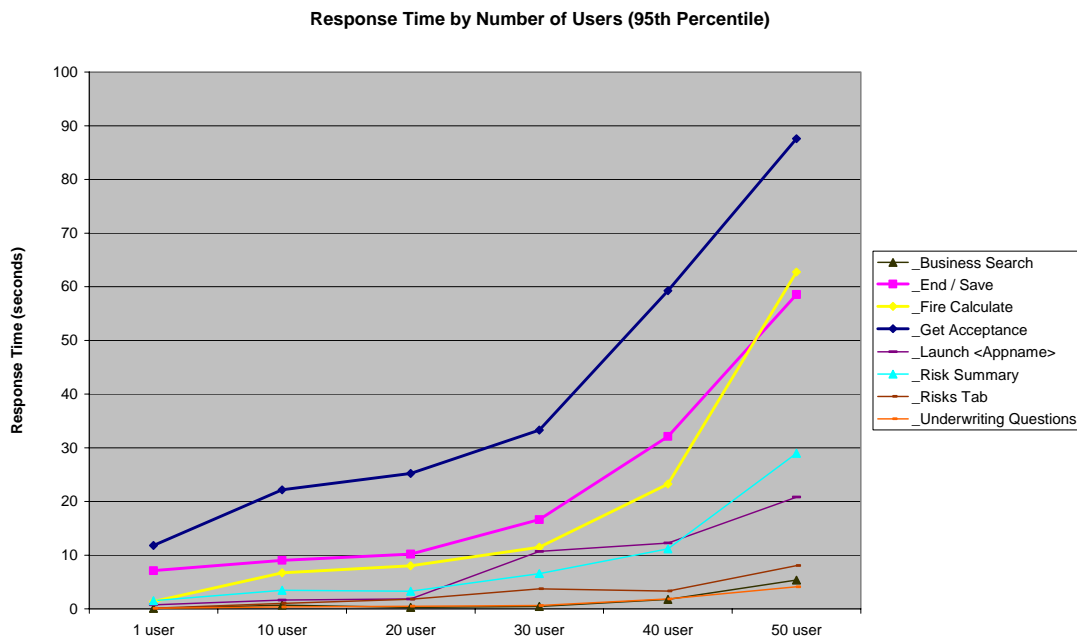
The next scatter chart shows the same timing point measured during a run done on the final morning of the exercise, once the “low hanging fruit” amongst the issues that we had identified had been addressed. In this case, the test suite was set up to increase the number of active users from 5 to 50 at 20 minute intervals during the course of the run. (I’d often intended to automate the change of workload like this, but had never got around to it. This time it was the only way to get the data to plot a “degradation curve” before my flight left, as I was due to be out at the production data centre recording production database traffic, so I didn’t have the chance to run a series of tests.)



Producing a Degradation Curve with a Single Test Run

The graph below shows the degradation curve for a few transactions, derived from the results obtained during the unattended run above. (Thanks to one of Scott Barber's articles for the idea for how to present this!)

Clearly the system was still showing signs of stress at somewhere between 30 and 40 concurrent users on our test hardware (a single CPU desktop), but it was performing an awful lot better than it was when we started.



Outcome of the Exercise

For the record, the major issues identified during the course of this exercise included:

- The most significant issue was that a component responsible for invoking requests on another server was serialised (due to a Java object being defined as static). There was only a single instance of this component in the architecture, so the hardware solution that was in train (adding more application servers) would not have helped – total throughput was constrained by this one component.
- An index was missing on a 45,000 row table that was queried (table scanned) 27 times in the course of one transaction. This had the largest impact on single-user performance, but had not been noticed because individual queries were of the order of one to two seconds. This really demonstrated the value of being able to monitor all database traffic unobtrusively.
- Java garbage collection was invoked explicitly and excessively by the application. This had been introduced following an earlier out of memory exception due to a memory leak which had since been corrected, in the mistaken belief that minimising the memory footprint of the JVM was a good thing.
- The application server was using a JVM version optimised for workstation rather than server usage.
- There was an excessive level of logging enabled in the application, generating several GB per day of logs in production. Reducing this gave a significant saving in response times.

- Various application optimisations to eliminate redundant steps and unnecessary polling were identified.

Overall, the changes resulted in a reduction in single-user response times of the order of 60%, but more importantly the system was now scalable. By the time we had finished, the application server was CPU constrained, which is how it should be.

In Conclusion

Over the course of two weeks (albeit a rather sleep-deprived two weeks) we executed more than 160 test runs, allowing us to identify a number of significant issues across various layers of the architecture. Numerous application and configuration changes were made during the exercise – both to help with diagnosis and to resolve identified issues. Many of those changes required the test scripts to be re-recorded from scratch.

In this paper I have explained the techniques I use to minimise the time spent on the mechanics of the testing. In this case these were absolutely essential to allow us any time for diagnosing and resolving issues.

Decoupling the definition of the workload model from the actual test scripts and automating the customisation of the scripts minimises the test cycle time following application change. Monitoring response times and varying the workload during the course of a test can substantially reduce the number of test cycles required – particularly for “exploratory”, diagnostic tests.

Fewer and faster test cycles mean more effective testing.

About the Author

Richard Leeke is one of the founding partners of Equinox Ltd (www.equinox.co.nz), a software architecture consulting company based in Wellington, New Zealand. Richard specialises in the areas of performance testing and the diagnosis and resolution of application, database, and infrastructure related performance issues. He’s a fairly regular contributor (of both questions and answers) to various performance testing related forums and can be contacted at richard.leeke@equinox.co.nz.