WOPR9 –Experience Report

Richard Leeke

The Assignment

I am currently testing a large J2EE application for an NZ government agency. The solution is a heavily customised package, being delivered in a series of incremental releases. It supports around 1,200 concurrent users as well as significant transaction volumes from interfaced internal legacy systems as well as external systems.

Technologies involved in the solution include Websphere Application Server (30 instances across 15 Solaris hosts), an Oracle database, JMS over MQ, a Stellent document management system, Windows server based document production and PDF conversion systems and interfaces to 3 large Sybase-based legacy systems. In all there are over 30 hosts, including 3 Sun Enterprise servers making up the solution. This infrastructure is duplicated exactly in the performance testing environment, along with 3 Windows servers for load generation.

The Challenges

In the course of the testing we have run into numerous limits or limitations of the testing tools we are using (the old generation Rational performance tester). Some of these are particular limitations of that tool, others seem to be more generic, and would present similar challenges with all of the tools I'm familiar with.

Tool limitations we have encountered include:

- Inability to handle required data correlation for the highly dynamic HTML
- Inability to handle data-dependent conditional navigation in the application
- Exceeding the limit for the size of log files and number of timing points – for long runs attempting to access the results leads to “out of memory” errors
- Extreme difficulty in exercising realistic navigation paths (this almost caused us to miss a critical issue which would have caused daily system outages)
- No inherent support for session abandonment, session churn or multiple sessions per user – all critical to achieve realistic emulation with this application
- Extreme difficulty in handling application or scripting errors without “losing” virtual users from the test run – again significantly distorting the realism of long tests – as load drops off
- Running out of agent memory on long tests due to memory leaks in the tool (the largest leak subsequently proved to be a fault in our solution for the previous issue!)
- HTML constructs not supported by the Rational correlation mechanism
- Inability to reapply test script customisations to re-recorded scripts for each application release
- The tool's reporting was too slow and too inflexible to support diagnosis and test prioritisation during a very aggressive test execution cycle
- The tool's reporting does not support reporting against the (perfectly valid and very sensible) way the performance requirements are expressed

The Sausage Machine

The amount of customisation of recorded scripts required for them to play back, the extent of application change with each release and the frequency of releases meant that re-recording scripts and manually re-applying all required customisations for each release was simply not viable.

The solution we have come up with, affectionately known as "The Sausage Machine", is a testing framework built from a combination of:

- Standard Rational VU scripting components
- Perl scripts to automate the majority of the required script changes
- Excel macros to generate test pacing from the workload model spreadsheet
- Perl scripts to replace the tool's data analysis
- Excel macros to help with timely presentation of test results
- (Under development) a custom graphical tool to assist with problem diagnosis

In short: string and chewing gum - but it does the job.

This experience report focuses largely on the approaches used for addressing generic issues likely to be common to many tools, rather than Rational-specific quirks. In particular, it discusses the techniques used for coping with dynamic HTML, the approach to regeneration of scripts quickly after application change and the approach to reporting.

Outline of the Scripting Process

The process for generating and running a working test script consists of the following steps:

1. Define the navigation path to be used and take screen shots for documentation.
2. Name the transaction with a short mnemonic.
3. Record (an initial attempt at) a set of scripts for the transaction, following a pre-defined script.
4. Run the first phase of the Sausage Machine over the set of scripts, to insert tags at each point of user interaction, which identify each business transaction and transaction step. These tags drive the pacing and timing of the script, as well as providing markers for some of the other automated edits. Tags are inserted whenever the script includes a “think delay” of more than a specified threshold – by default this is 5 seconds.
5. Review the tagged scripts to ensure that all of the tags correspond with the comments entered during recording. If not, either fix up the tags manually or re-record.
6. Enter the details of the transaction and steps into a master spreadsheet which defines the workload model and drives several aspects of Sausage Machine processing. These details include translation of the tags into meaningful names, delay times to allow after each step, the performance criteria that apply to this step and various other classifying details to assist with reporting.
7. Review the scripts for changes required which can not be generated entirely automatically with the Sausage Machine. In some cases this means inserting additional

tags to tell the Sausage Machine what to do, in other cases separate entries in configuration files are required.

8. Run the second pass of the Sausage Machine to make all required changes. There are three main phases:
   - optimisation of data correlation runtime efficiency
   - specific changes to make the scripts play back and to drive reporting
   - insertion of code to control script pacing and record response time details.

   The Sausage Machine logs details of what it does, and archives a date-time stamped copy of the before and after versions of each script, to assist with subsequent iterations.
9. Test the script.
10. Repeat steps 3 to 9 until it actually works. In some cases this requires adding extra features to the Sausage Machine.
11. Run the scripts as part of an overall suite, with the workload controlled by one of the files generated from the workload spreadsheet.
12. Export the raw Rational result data and analyse using Perl scripts and Excel macros.

Some Statistics

The overall test suite consists of:

- 18 transaction types, with 1 or more scripts per transaction
- 302 pages (many of which are repeated between transactions, however script re-use is not viable)
- 330,000 lines of test script (including recorded page data)
- 50,000 lines in 20,000 different sections are edited by the Sausage Machine across the scripts for the 18 transactions
- 285 automatically inserted tags
- 180 manually inserted tags
- 630 lines of configuration files
- A total of 810 manually edited lines to generate 50,000 lines of changes
- Roughly 30% of manually edited lines – around 250 lines - were new or modified for the latest application release (plus several new sausage machine features were added)
- Overall the re-scripting process took two to three person weeks (excluding enhancements to the test data set-up process)

The specific techniques used for applying edits to test scripts are naturally specific to Rational - but the same approach could be applied just as well to any tool which exposes a scripting language. The reporting approach is completely generic and can be applied to results from any tool.

One of the key benefits of the approach is that it makes it straightforward to make generic modifications to the way the scripts work after completing initial script development. For example, at various times we have added in custom features to allow profiling of the SQL generated, to profile the rate of business object creation (by type of object) in the application server, to analyse overall heap usage, and the rate of creation of transient and long-lived objects, all broken out by transaction step.

Coping with Dynamic HTML

What's the problem?

There are lots of cases with dynamically generated pages which the Rational tool does not cope with. The (old generation) Rational tool generates scripts based on the pages seen at recording time and only tries to cope with certain classes of change (URL string parameters, form field values, cookies, etc) at run time. Some of the newer tools, including the new Rational one, which interpret the DOM at runtime may cope rather better with some of the classes of problems. However, we failed even to get a single page to play back with the new Rational tool – so never got to put that claim to the test.

Some examples of the types of issues we had to cope with follow. Many of these would be problematic for other tools, too.

- Sets of form fields ("widgets") may or may not occur, depending on data values. (We refer to these as "optional widgets".) Where possible the problem was avoided by selecting homogeneous test data ("vanilla cases"), but the extent of the variance was simply too great to avoid all examples. Attempting to POST back values for fields which were not present could cause application failure.
- The names for sets of form fields (widgets) change at run time depending on data values, or the presence or absence of earlier fields. (We call these "mutating widgets".) Rational assumes that names don't change, so the POSTs name invalid fields.
- Some form field names change at run time simply due to iteration – visit the same page twice with the same data and some field names are different.
- Form field **types** vary in a data dependent way (e.g. a set of *hidden fields* at record time becomes a set of *buttons* at playback time – resulting in POSTing several buttons at the same time – which caused nasty and obscure errors).
- The **name** of a field requiring correlation is found from the **value** of another field (and that field just happens to have a mutating name to keep us on our toes).
- Dynamically generated Javascript for launching additional windows contains embedded URL parameters.
- Javascript dynamically adds a new button to the DOM (so the button is not seen by the script recorder) and then submits the button.
- Etc, etc, etc.

In many cases these issues are either entirely generic, or at least apply to all HTML pages in this particular application. In these cases the approach we have adopted is simply to post-process the scripts to fix-up the issues.

In other cases the required fix-ups require intelligent intervention. We handle these cases in a variety of ways, whilst sticking rigidly to one guiding principle: we don't make direct changes to the test script code, we find some means of encoding the change in such a way as to minimise the effort involved in re-applying the equivalent change to new scripts following re-recording for new release of the application.

Had this approach been fully defined from the outset, we would probably have a single configuration file per script defining the required changes. As it is, the approach grew in a piecemeal fashion, which has left us with an ugly hodgepodge of separate configuration files and specially formatted tags which are manually inserted into the raw scripts as markers for actions required of the sausage machine.

Regenerating Scripts with each new Release

The project lifecycle sees two or three major releases a year. We are engaged to repeat performance testing for each release deemed to present a significant risk. Given our recent experiences, it seems likely that we will be engaged for all major releases – and probably also be retained to provide additional diagnostic and tuning assistance for interim development releases.

While each major release is undergoing customer acceptance testing there are weekly minor releases into the functional test environments. Often these minor releases would completely trash some or all of our test scripts – so there is a constant tension between the time required to keep the test environment up to date and be testing the latest version and the time required actually to run and analyse tests and diagnose issues. We generally end up compromising on one or two full re-records of all scripts per major release, by freezing (or at least chilling) the performance test environment once the release is reasonably stable, and then only taking selective patches where they are believed to have a performance impact.

This is far from ideal. The customer would like always to be testing the latest release – and certainly to test the release that is finally destined for production, but the project timeline is too aggressive and the scripting complexity too great for that to be viable. We think we are achieving miracles by having the test suite going in two or three weeks after a new release – the customer would like two or three days.

Results Analysis

This project has presented several reporting challenges:

- An aggressive test execution cycle demands rapid analysis of results
- The tool cannot handle the volume of results from extended duration runs
- Performance requirements (really goals) are expressed in a way that is not measurable with "out of the box" reporting

The test execution phase prior to each production release is typically planned to last around 4 weeks – but for the most recent release this was extended (and deployment delayed) by an additional six weeks while performance issues were addressed and additional hardware purchased. On each release tested to date we have encountered numerous "no go" issues. The pressure to diagnose and resolve these issues quickly is of course intense, especially when project delays block other dependent programmes in the organisation. The overall "burn" rate of delays to a project of this size is huge.

During the execution phase, activity is intense. We typically work on a daily cycle of two or three two hour tests (or more, shorter exploratory/diagnostic tests) by day, with a longer test (perhaps eight hours) overnight, usually supplemented with additional tests at the weekend. We have a daily half hour "scrum" style meeting at 9:15 each morning with key management and technical staff from all teams involved in the solution, to review the results of the previous day's testing, perform initial triage of issues and agree priorities and a day plan. These sessions are attended by a mixture of technical and management staff, and are particularly keenly followed when there are roadblock issues. We counted seven technical staff plus seven project managers at one morning scrum recently.

Test results are explored in detail – and with the technical leads for all aspects of the solution present this proves to be an effective forum for issue diagnosis – or at least identifying worthwhile avenues to explore. However, for this to be an effective use of the time of so many key project staff, it is essential that the results are presented in a way that allows rapid drill-down in numerous dimensions.

To have the results from an overnight run in a state to allow analysis by 9:15 the next morning is a significant challenge – especially on a performance tester's sleep budget. Extended duration runs may have millions of timing points (several times larger than the largest Rational's in-built reporting can cope with). Even with the largest result sets Rational **can** manage it might take 10 or fifteen minutes to produce a single view of the data – and there is no fast slice-and dice style reporting. (Or it might take that long to produce an "out of memory" error.)

Over the years we have found various tricks for getting around these limitations whilst sticking with Rational's reporting – but invariably these result in either an even more cumbersome reporting process, or severe dilution of the data, or usually both. One work around we used at one stage on this project also had the effect of causing us to miss a critical issue for a while. This one is worthy of explanation, so is described briefly as an appendix.

However, our productivity and effectiveness during the test execution phase was simply not high enough to meet the client's needs by using the Rational tool's in-built reporting, so (like most performance testers I know) we now do almost all our analysis and reporting outside the tool.

Examples of Results Presentation

A few samples of the ways that we present results data follow. All of these views are available within a few clicks from the spreadsheets generated automatically at the end of each run and are used regularly during the daily review meetings.

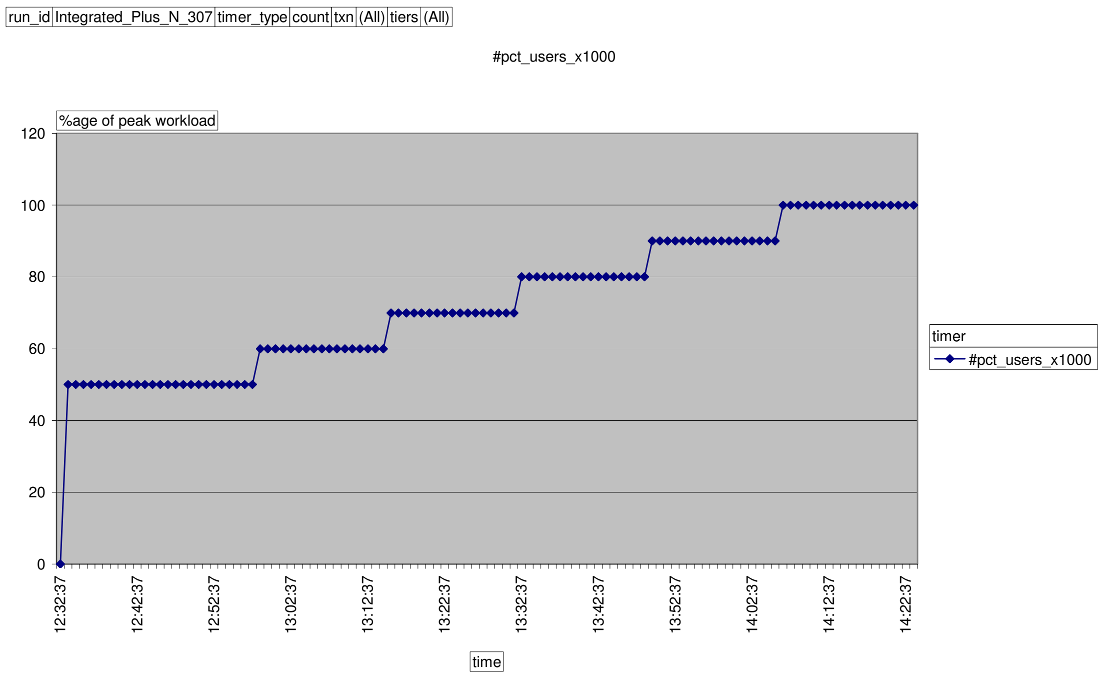
"Traffic Light" View

This view shows a simple "pass/fail" against non functional requirements for every timing point, at each stage of the run. The example shows a sample from the 100% load stage of a test run (1145 active users).

range	timer	txn	count	tph	response_95	response_98	response_99
12) 1145 Users	_CDoc_a_Icon	__CDoc	569	2276	0.3	0.5	0.56
12) 1145 Users	_CDoc_b_Search	__CDoc	572	2288	0.88	1.14	1.39
12) 1145 Users	_CDoc_c_List	__CDoc	574	2296	3.02	3.69	3.83
12) 1145 Users	_CDoc_d_Docs	__CDoc	575	2300	3.84	4.81	6.13
12) 1145 Users	_CDoc_e_AddDoc	__CDoc	576	2304	0.69	0.91	1.55
12) 1145 Users	_CDoc_f_ExpandDocType	__CDoc	577	2308	0.28	0.39	0.58
12) 1145 Users	_CDoc_g_ExpandCode	__CDoc	579	2316	0.23	0.36	0.53
12) 1145 Users	_CDoc_h_SelectClass	__CDoc	581	2324	8.16	11.91	13.5
12) 1145 Users	_CDoc_i_FilterDocType	__CDoc	580	2320	0.17	0.31	0.45
12) 1145 Users	_CDoc_j_DocTypeSrchOK	__CDoc	578	2312	2.08	2.88	3.33
12) 1145 Users	_CDoc_k_RemoveTo	__CDoc	577	2308	0.31	0.53	0.66
12) 1145 Users	_CDoc_l_AddTo	__CDoc	576	2304	0.47	0.63	0.72
12) 1145 Users	_CDoc_m_AddCC	__CDoc	577	2308	0.5	0.64	0.88
12) 1145 Users	_CDoc_n_AddDocContinue	__CDoc	581	2324	14.55	17.45	20.63
12) 1145 Users	_CDoc_o_SaveDoc	__CDoc	581	2324	0.13	0.19	0.27
12) 1145 Users	_CDoc_p_DocPropsOK	__CDoc	582	2328	33.16	51.91	58.8
12) 1145 Users	_CaseDetails_Message	N/A	1239	4956	15.08	19.22	19.73
12) 1145 Users	_DeptClaims_a_Icon	__DeptClaims	183	732	14.53	16.66	17.83
12) 1145 Users	_DeptClaims_b_Tab	__DeptClaims	183	732	10.73	16	18.83
12) 1145 Users	_DeptWork_a_Icon	__DeptWork	389	1556	4.88	7.23	10.23
12) 1145 Users	_DeptWork_b_DeptWork	__DeptWork	389	1556	1.27	1.88	2.16
12) 1145 Users	_ERN_a_NextTask	__ERegNonWk	240	960	8.52	9.83	10
12) 1145 Users	_ERN_b_Select	__ERegNonWk	240	960	0.38	0.56	0.64
12) 1145 Users	_ERN_c_DoTask	__ERegNonWk	239	956	8.66	9.56	10.27
12) 1145 Users	_ERN_d_ACC45	__ERegNonWk	239	956	0.61	0.73	0.86
12) 1145 Users	_ERN_e_Name	__ERegNonWk	240	960	0.77	1.14	1.41
12) 1145 Users	_ERN_e_Name.0-20	__ERegNonWk	240	960	0.77	1.14	1.41
12) 1145 Users	_ERN_e_Name.1-10	__ERegNonWk	240	960	0.77	1.14	1.41
12) 1145 Users	_ERN_f_Create	__ERegNonWk	239	956	3.08	3.58	3.73
12) 1145 Users	_ERN_g_RegOK	__ERegNonWk	236	944	55.34	72.7	79.44
12) 1145 Users	_ERN_h_CoverStatus	__ERegNonWk	235	940	0.3	0.5	0.59
12) 1145 Users	_ERN_i_Branch	__ERegNonWk	235	940	0.22	0.36	0.39
12) 1145 Users	_ERN_j_Finish	__ERegNonWk	236	944	7.17	10.33	44.17
12) 1145 Users	_ERegNonWkNavIn_a_Icon	__ERegNonWk	1	4	0.67	0.67	0.67
12) 1145 Users	_EmpSrch_a_Icon	__EmpSrch	21	84	0.16	0.16	0.16
12) 1145 Users	_EmpSrch_b_EmpTab	__EmpSrch	21	84	0.23	0.23	0.23
12) 1145 Users	_EmpSrch_c_Search	__EmpSrch	21	84	28.95	28.95	28.95
12) 1145 Users	_EmpSrch_c_Search.1-10	__EmpSrch	14	56	28.95	28.95	28.95
12) 1145 Users	_EmpSrch_c_Search.11-100	__EmpSrch	6	24	1.05	1.05	1.05
12) 1145 Users	_EmpSrch_d_Choose	__EmpSrch	20	80	29.86	29.86	29.86
12) 1145 Users	_EmpSrch_e_ClaimsTab	__EmpSrch	20	80	0.38	0.38	0.38
12) 1145 Users	_Logoff_a_OK	__Logoff	11	44	0.27	0.27	0.27
12) 1145 Users	_Logon_a_URL	__Logon	93	372	3.06	3.45	3.97
12) 1145 Users	_ManRegWk_a_ACC45	__ManRegWk	40	160	0.53	0.56	0.56
12) 1145 Users	_ManRegWk_b_Name	__ManRegWk	40	160	0.97	1.13	1.13
12) 1145 Users	_ManRegWk_b_Name.0-20	__ManRegWk	40	160	0.97	1.13	1.13
12) 1145 Users	_ManRegWk_b_Name.1-10	__ManRegWk	40	160	0.97	1.13	1.13
12) 1145 Users	_ManRegWk_c_Create	__ManRegWk	40	160	2.67	3.06	3.06
12) 1145 Users	_ManRegWk_d_Occupation	__ManRegWk	40	160	0.48	0.64	0.64
12) 1145 Users	_ManRegWk_e_OccSearch	__ManRegWk	40	160	0.69	2.12	2.12
12) 1145 Users	_ManRegWk_e_OccSearch.1-10	__ManRegWk	28	112	0.39	0.39	0.39
12) 1145 Users	_ManRegWk_e_OccSearch.101+	__ManRegWk	2	8	2.12	2.12	2.12
12) 1145 Users	_ManRegWk_e_OccSearch.11-100	__ManRegWk	10	40	0.38	0.38	0.38
12) 1145 Users	_ManRegWk_f_OccOK	__ManRegWk	39	156	1.19	1.36	1.36

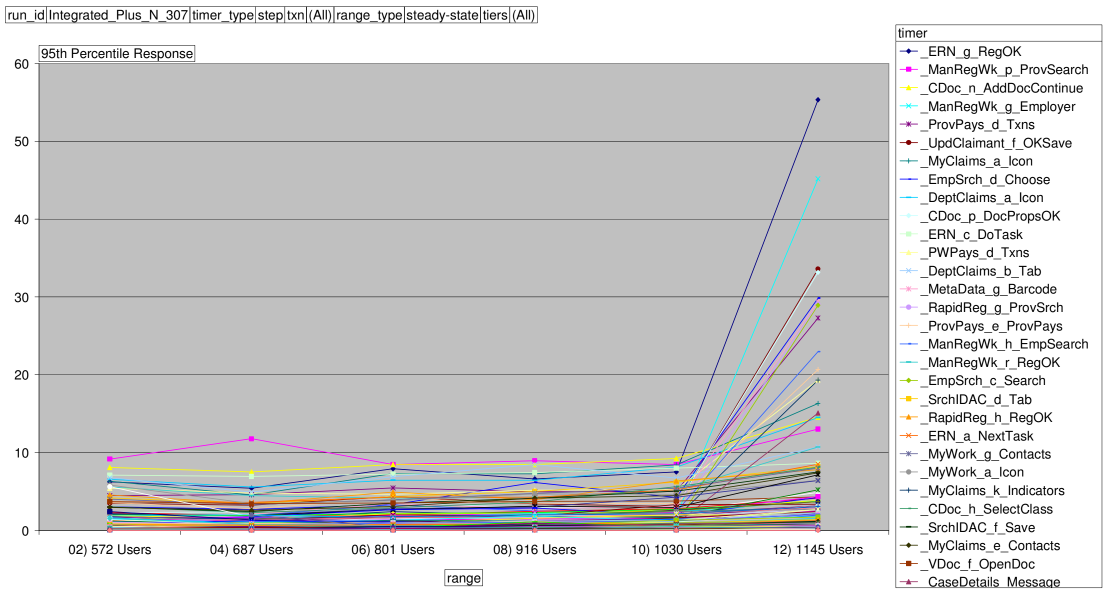
Workload

The Rational tool allows user-defined timing points to be recorded, but not other types of metrics – so we log useful metrics as "pseudo-timers" to allow reporting on the same timescale as performance measures. This chart simply shows the workload as a percentage of the total number of users involved in the run.



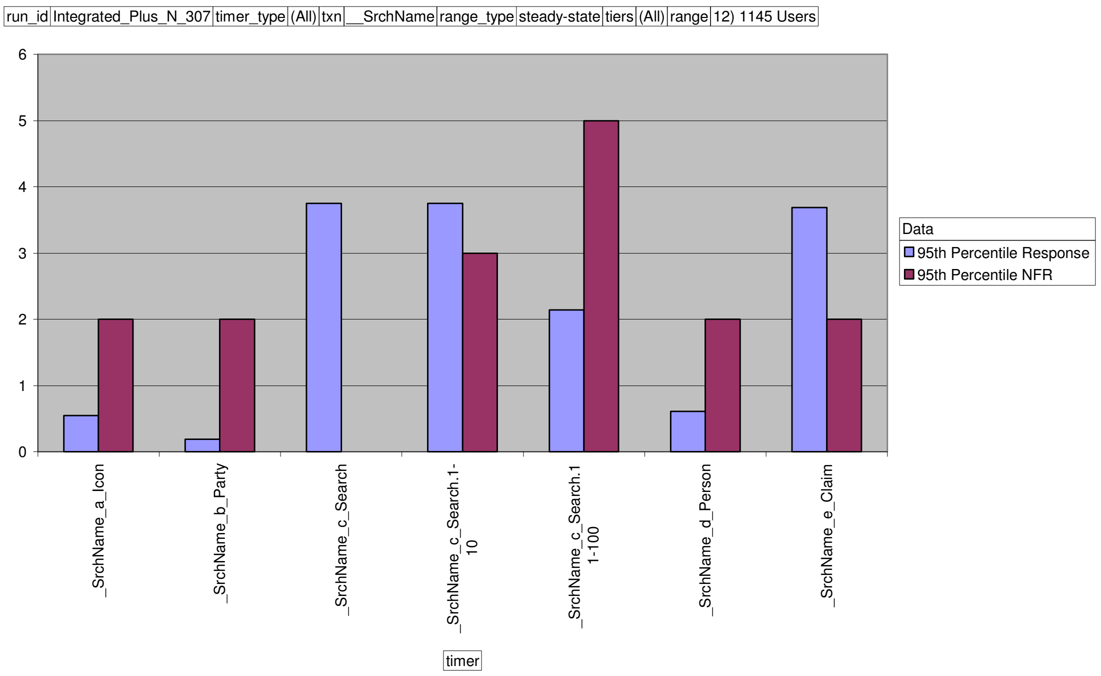
The Degradation Curve

Probably the view of results which we use the most is the degradation curve. In this case, the chart shows 95th percentile response times against load for each of the periods of steady-state load.



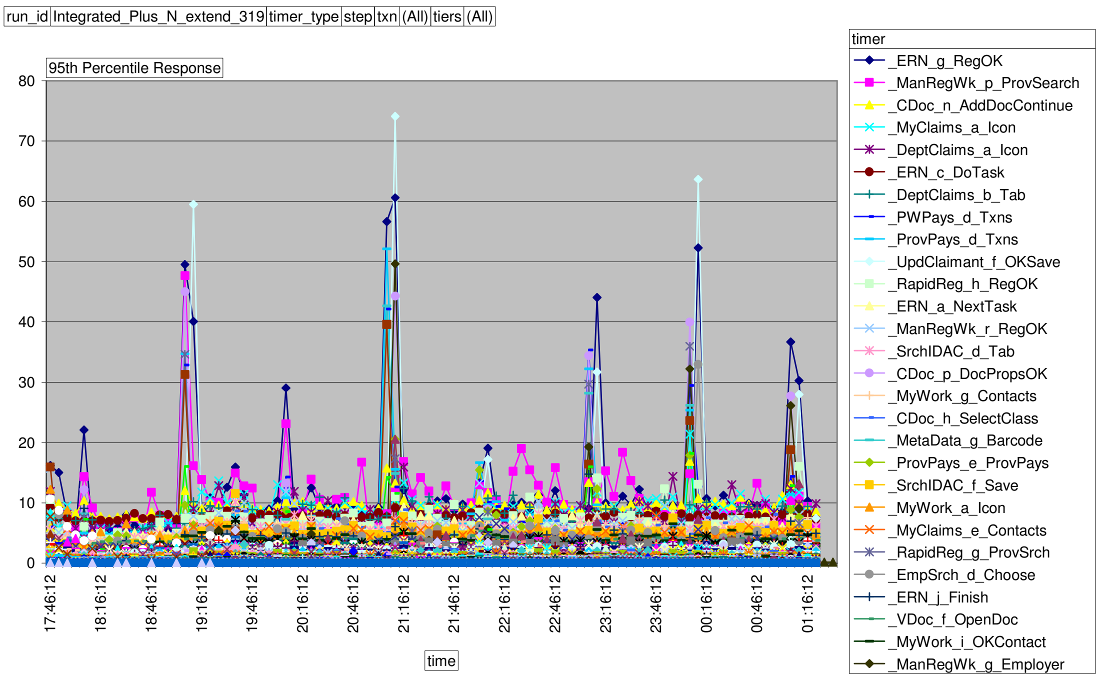
Response vs NFR per Transaction

This chart shows the detail of response times (blue) versus non-functional requirements (red) for all steps making up a particular transaction. Note that for step c) the requirement varies according to the number of rows returned.



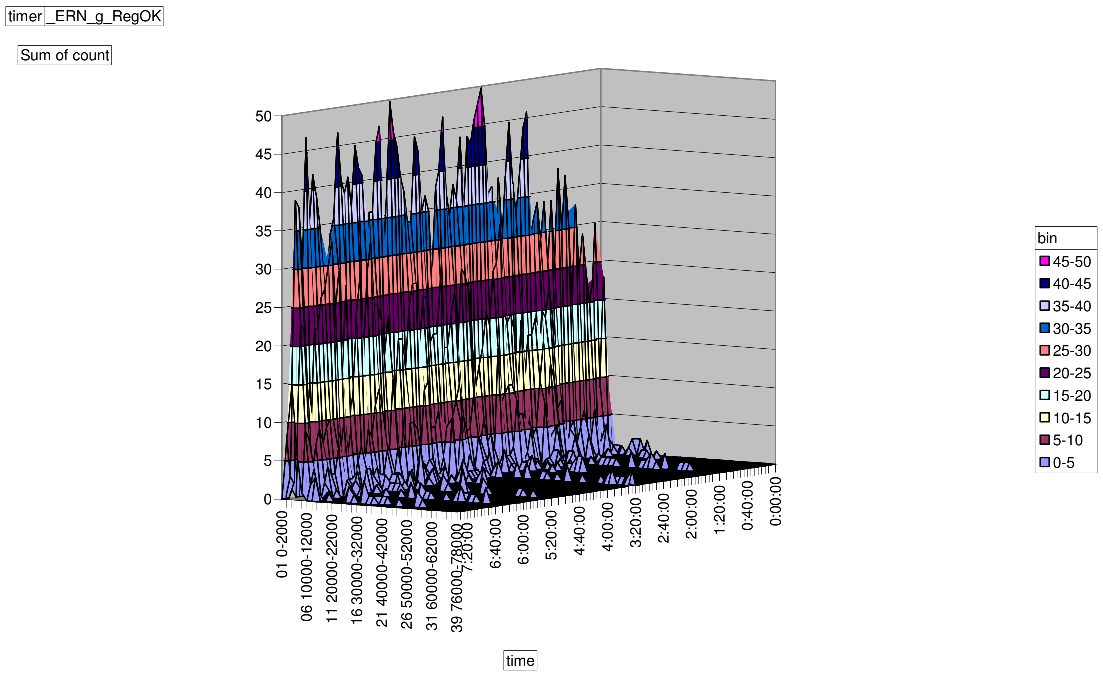
Response Broken Down by Time Buckets

This chart shows the 95th percentile response times in 5 minute buckets over the course of an 8 hour run. Clearly several transaction steps are blocked for extended periods on an hourly basis. By examining the mix of transaction types exhibiting the spikes, some clues as to which tiers of the application may be involved can be determined. We have all transaction steps classified by tier, so the chart can be filtered by tier, looking for patterns.



Distribution of Response Times

A 3-D surface chart can be an effective means of visualising the distribution of response times for a single transaction step. In this case the chart shows the details of the hourly spikes from the previous chart, for one particular step. This chart tends to be more effective as an interactive diagnostic aid than as a snapshot like this, however.



We are also just starting to experiment with a custom-written tool for displaying and exploring a scatter chart showing all timing points. The challenge is to make it fast enough to be useful, and to provide intelligent drill-down reporting.

Appendix - Critical Issue we Almost Missed

As mentioned above, one of the workarounds we used to get around tool limitations almost caused us to miss a critical issue shortly before the system first went live.

We were experiencing two significant problems when we attempted to run extended duration tests. Firstly the volume of results data was exceeding the limits Rational could report on (or even export) for a single run. About three hours at peak load was the most we could manage, yet we wanted to run for ten to twelve hours non-stop. Secondly, there was a significant memory leak when executing scripts, leading to the load generators running out of memory after around four hours.

We only discovered these in the final phases of testing before initial go-live – we simply hadn't been in the position to attempt long test runs earlier. We had a few sampling techniques to reduce the volume of results data (which was not ideal but capable of delivering worthwhile results). However the memory leak was a worry. We had no idea whether this was caused by a fault in our heavily customised scripts or a bug in the underlying Rational engine. We therefore could not predict with any certainty how long (if ever) it was likely to take to resolve.

We decided that we didn't have time to discover how long this particular piece of string was, so dreamt up a hair-brained scheme to get around the problem. We decided to break a ten hour run into five, two-hour chunks. The trick was that for the test to be valid we needed to resume the existing user sessions at the start of each new "chunk". Simply abandoning or logging off the existing sessions and logging on again would have given completely different patterns of memory usage in the application servers.

It actually proved relatively simple to save session state (predominantly current session cookie values) to files at the end of each chunk and then resume the sessions in the next chunk of the run – driving the successive chunks by running Rational repeatedly from a wrapper perl script. We were reasonably confident that we were modelling session memory usage accurately, and couldn't see any reason why the brief pauses should distort anything.

Wrong. In fact what happened was that the five minute pauses with no activity gave the JVM's garbage collector a chance to come-up for air (in some way that we never really understood), which had the effect of delaying the onset of an issue causing the JVM to switch to a far more aggressive GC mode – at which time the system became effectively unusable. Luckily we were nervous enough about the possible effects of the pauses that we tried a longer duration single run with much reduced logging, which happened to show up the issue before the memory leak in the VU processes became a problem.

We subsequently identified the major cause of the memory leak, and took to writing our own timing data out to flat-files, rather than relying on the Rational repository – so we can now run long runs non-stop.