

Part 5: Determining the Root Cause of Script Failures

(This item originally appeared on The Rational Developer Network, an online community for customers of IBM Rational Software. To find out more, including how you can get a free evaluation to the Rational Developer Network, please visit http://www.rational.com/services/rdn/find_out_more.jsp)

In the forums on performance engineering that I participate in and moderate, I get questions like these almost daily:

- *"I recorded my script and it worked just fine for one user, but when I tried two users they both failed. What's wrong with my Web server?"*
- *"My scripts pass but the database doesn't get updated. What's wrong with my database?"*
- *"My scripts work for five users, but when I play back more than ten users my scripts time out. Is my application overloaded already?!"*

In each case, the author assumes that since the script passed in some scenarios, or since the Rational® TestManager software showed a Pass result, the problem must be with the application. While it's true that the application isn't performing as expected and while it may be true that the application is at fault, "appearances often are deceiving," as Aesop (620–560 BC) warned. An application can act in unexpected ways because scripts present it with situations that real users could never create.

Besides, the obvious symptom is very rarely the actual cause of a script failure. Here in the fifth article of the "Beyond Performance Testing" series, we'll take a look at how to analyze script failures with the intent of finding their root cause so we can debug our scripts effectively and then build stable, robust, reusable scripts to test our applications with. We'll explore some common scripting issues that can cause failures (both true failures and false failures) and I'll show you some methods for collecting information about script failures.

So far, this is what we've covered in this series:

Part 1: Introduction

Part 2: A Performance Engineering Strategy

Part 3: How Fast Is Fast Enough?

Part 4: Accounting for User Abandonment

This article is intended for all levels of Rational TestStudio® VU scripters.

Recognizing and Minimizing False Failures (and Passes)

In the Test Log for a VU script, TestManager classifies executed commands as having either passed or failed, but these classifications don't always mean what you might assume they do. Some of the Fail results that show up there aren't actually indicative of problems with the script or application and can be prevented by adjusting HTTP return code or file size parameters. By the same token, some of the Pass results that appear actually mask critical errors. We'll consider how to minimize these false failures and passes first. You should evaluate these areas when developing your scripts as well so you don't waste time troubleshooting in other areas only to find out that you've been misled by the Test Log.

Failures Based on HTTP Return Codes

[HTTP return codes](#) are sent back to the client for each object retrieved from the Web server when you record a script. These are the code numbers you'll see in the `http_header_recv` lines of each block that look like the following:

```
http_header_recv ["script~1.017"] 200; /* OK */
http_header_recv ["script~1.025"] 304; /* Not Modified */
http_header_recv ["script~1.042"] 302; /* Moved Temporarily */
```

By default, when you play back your scripts TestManager expects to get that same code when the same object is requested and will only classify the executed command as having passed if that's the case. This can be either a good thing or a bad thing. It's a good thing in that you won't get a passing result if you're expecting a 200 (OK) and you get a 404 (File Not Found). On the flip side, if you're expecting a 200 (OK) and you get a 304 (Not Modified) — which implies that the request was filled from the cache — you get a failure. In most cases this will be a false failure, because you do want to be able to retrieve objects from the cache.

You can help eliminate some of these false failures by changing the defaults to allow redirects (301, 302, and possibly 303 in place of an expected 200 or 304) and/or cache responses (304 in place of an expected 200, 301, 302, or possibly 303). There are a couple of ways to do this. One way is to modify the `push Http_control` line in each script to include the `HTTP_CACHE_OK` parameter if you want to allow cache responses and the `HTTP_REDIRECT_OK` parameter if you want to allow redirects. That modification has been made to the line below.

```
push Http_control = HTTP_CACHE_OK | HTTP_REDIRECT_OK;
```

Alternatively, through TestManager you can modify the parameters for all of the scripts in a suite at once. (Note that these changes will only affect scripts when they're played back as a part of that suite.) First, open the desired suite and choose Suite > Edit Settings from the menu bar. You'll see a screen like the one in Figure 1.

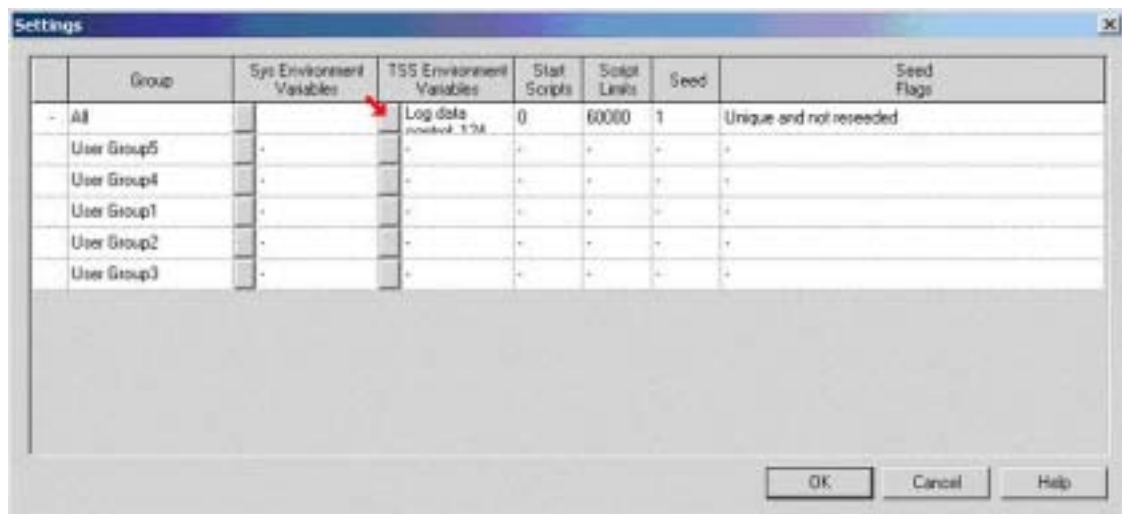


Figure 1: Settings screen for a suite

Then click the TSS Environment Variables button for all user groups (indicated by the red arrow in Figure 1) and click the VU HTTP tab. This brings you to the screen in Figure 2.



Figure 2: VU HTTP tab, TSS Environment Variables dialog box (default)

You can see that the default is to not allow the HTTP Control options. To change that, simply check the desired boxes, as shown in Figure 3.



Figure 3: VU HTTP tab, TSS Environment Variables dialog box (modified for return codes)

False passes based on HTTP return codes can also be a problem during playback. Sometimes, for example, a 200 is both expected and received, but in fact what's been received is the wrong thing. For instance, when you click a link and get a message that says "I'm sorry, the site is down for maintenance, please come back tomorrow," you've received a valid page with a 200 code, but you haven't received the page you were looking for. In general, detecting false passes is a matter of spot checking responses. Pages with embedded error messages are discussed in more detail [below](#).

Failures Based on File Sizes

File sizes are another way that TestManager determines if a request passed or failed during playback. In your scripts you'll see lines like the following:

```
http_nrecv ["script~1.003"] 100 %% ; /* 56304 bytes */
http_nrecv ["script~1.006"] 100 %% ; /* 2103 bytes - From Cache */
http_nrecv ["script~1.012"] 2048; /* 2048/2048 bytes */
```

While there are several variations on how file sizes are evaluated for correctness during playback, they really boil down to either allowing response sizes to vary or not. In the first case, you ensure the script shows 100%%; downloading a file of any size bigger than 0 bytes counts as a Pass. In the second case, you ensure the values in the command match the number of bytes when recorded; any file that's downloaded with a size different from that number of bytes counts as a Fail.

In most cases, allowing response sizes to vary will keep you from getting tons of false failures, since anything dynamic on an HTML page — even if you can't see it on the screen — can cause the file size to change and thus a failure to occur if responses sizes aren't allowed to vary. You may choose to dictate specific sizes for certain graphics or screens if, for example, you know that the correct screen/graphic will always be one size and that the error screen/graphic will always be a different size.

Other than modifying the code line by line, you can set the parameter globally like we did for the HTTP return codes above. For an individual script, you can do this by adding or deleting the `HTTP_PARTIAL_OK` parameter in the `push HTTP_control` line. For an entire suite, you can do this through TestManager by returning to the VU HTTP tab of the TSS Environment Variables dialog box and checking or unchecking the "Allow partial responses" option (see Figure 4). Once again, this is a useful tool to help minimize false passes and fails.



Figure 4: VU HTTP tab, TSS Environment Variables dialog box (modified for sizes)

Passes Returned for Embedded Errors

Embedded errors are those red words on a page that's substituting for the page you actually want, telling you that you failed to get to the page you were looking for. So instead of the Web server returning a 403 or 404 error, it redirects you to a custom error page that tells you why your request didn't work. These pages are generally thought of as user friendly, but they're also performance test scripter *unfriendly*. As mentioned above, TestManager (and virtually all other load-generation tools) expects errors in code, not text.

The bottom line is this: embedded errors are very good at presenting results that, on the surface, appear to be passing results, when in fact your script is just being redirected to the same custom error page over and over again. One indicator that this may be occurring during your test execution is that page response times are much faster than expected and don't seem to slow down even under relatively extreme loads.

Embedded errors really deserve an article all their own, but allow me to at least explain conceptually how to detect and handle them. We'll discuss other methods of detecting this issue later. If you have an application that uses embedded errors rather than HTTP return codes to signify errors, you'll have to write custom functions to read the actual HTML of the response from the `_response` file and look for those error messages. If your custom function finds one of those error messages, it must return an error using the `testcase` command and/or terminate the virtual user with the `user_exit` command. These concepts have been presented previously in different contexts in several "User Experience, Not Metrics" and "Beyond Performance Testing" articles.

Addressing Common Causes of Script Failures

Now we'll look at some common scripting issues that can cause real failures: datapools, navigation, data correlation, and authorization/authentication. I'll suggest ways to approach script failures when any of these issues might be the cause.

Datapools

Datapools, particularly when used for user names and passwords, often cause script failures that are deceiving. Without going into detail about creating or maintaining datapools, I do want to point out these key areas to verify when a script failure occurs:

- Ensure that in the datapool section of the script, the variables you want to be read from the datapool are marked as `INCLUDE`, not `EXCLUDE`. This is a very easy mistake to make.
- Ensure that the data in the datapool is correct.
- Ensure that you have enough data or you specify `DP_WRAP_OFF` if overlapping/duplicate data is a potential issue.

Remember always to check the request or requests before the first identified failure to verify input data. It often takes several requests past the actual script error to generate an error that TestManager recognizes as a failure. If you're unsure how to check what data was actually passed to the application for a particular virtual user, see the sections below under "[Collecting Information About Script Failures](#)."

Navigation

If you're using split scripts of any sort, you can end up with some extremely strange-looking errors that turn out to be the result of requesting pages in the wrong order. This can even happen with entire-path scripts when previous requests don't process properly. If you suspect that navigation could be involved in an error at either the script *or* the application level, I suggest the following approach:

- Ensure that the navigation path your virtual user is following is valid by testing it manually.
- Ensure that the last page that showed a Pass result in TestManager is the correct page and has the correct content (see "[Viewing Returned Pages After the Test Run](#)" for how to do this).
- Ensure that the data being passed in the request for the failing page is both correct and properly formatted (see "[Viewing Native Logs](#)" for how to do this).
- Step through the script one page at a time, evaluating both requests and responses, until you can narrow down the cause of error to one request/response pair (see "[Stepping Through Scripts](#)" for how to do this).

A significant majority of errors appear on the surface to be navigation related. The ones that actually turn out to be navigation related are usually due to either split scripts or the application. Even though issues related to split scripting are fairly common, they're beyond the scope of this article.

Data Correlation

Data correlation is another area that commonly leads to both script and application errors. *Data correlation* is just a fancy term for "I need to grab some data from the last page that changes every time someone accesses that page." [Part 11](#) of the "User Experience, Not Metrics" series discussed data correlation as it relates to authentication and session tracking.

Data correlation issues are often fairly difficult to detect. By default, Robot doesn't correlate any values during script generation. This may or may not be acceptable for your application. If you record a script and it doesn't work, I recommend regenerating that same script with the recording options set to correlate values. First go to Tools > Session Record Options, click the Generator per Protocol tab, and change the "Correlate variables in response" item to "All" (see Figure 5).

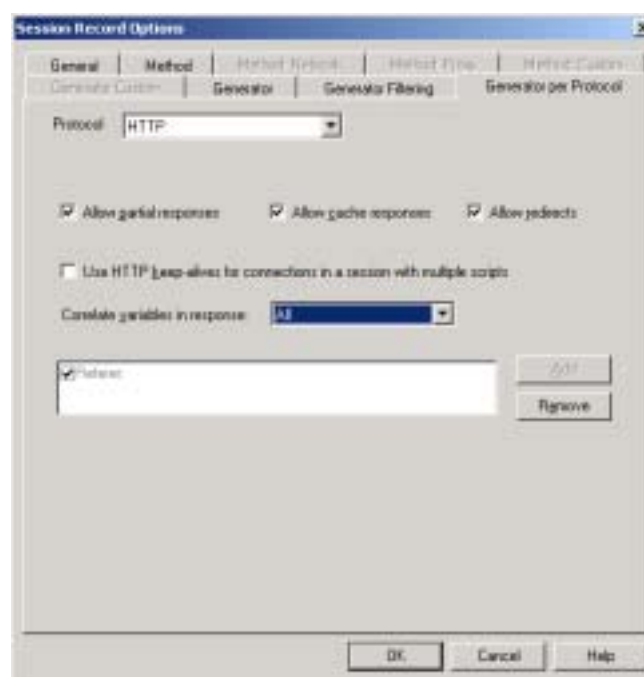


Figure 5: Generator per Protocol tab, Session Record Options dialog box

Then regenerate the script in Robot by going to Tools > Regenerate Test Scripts from Session.

If the script works this way, you have two options. The first is to leave it as is and the second is to evaluate exactly which variables need to be correlated for the script to work properly. The latter is a tedious process no matter how you do it and is beneficial only if (1) conserving small amounts of script overhead is more important than hours of your time, or (2) making your code look "clean" is very important. The one good thing about data correlation issues is that they almost always present themselves the first time the script is run. The script will almost never pass, even with a single virtual user, if you have a data correlation issue.

Authorization/Authentication

Authorization and/or authentication issues are really just special cases of datapool and data correlation issues. All I want to add here is a list of some common indicators that an error may be related to either authorization or authentication:

- The logon page appears, but every subsequent page yields an error for one or more virtual users.
- The HTTP return code on the failing pages is either 401, 403, or any code in the 500s.
- The script seems to be working, but back-end data isn't being updated.
- The script seems to be working, but searches are returning no data when you know the data exists and the request for data is formatted correctly.

In these cases, first make sure that all of your user credentials are valid for the actions you're assigning users to. Next ensure that all session identification information is being correlated properly (see [Part 11](#) of the "User Experience, Not Metrics" series for more information). Finally, get the member of the development team who's responsible for security involved. Security can be done so many different ways that it's simply impossible for me to make further generalizations about how to debug these issues.

Collecting Information About Script Failures

There are probably more ways to collect information about performance script failures than there are performance testers in the world. Needless to say, there's simply no way I can discuss all of them in this article. Besides, it would be terribly dishonest of me to lead you to believe that I know them all — or even most of them! What I can do, however, is discuss some of the common methods at our disposal through TestManager plus a few custom methods that aren't native to TestManager, some of which you may be familiar with if you've used other industry-leading tools.

Please note that although I'm going to discuss only information available through TestManager, I'm not advocating ignoring other sources of information. On the contrary, I strongly encourage you to use all sources of information available to you, which may include but not be limited to the following:

- Web server logs
- application server logs
- network or application monitoring software
- database logs and/or queries
- system admins/developers
- system documentation
- RFC documentation

Viewing Native Logs

By default, TestManager captures and saves in log files all of the request and response data transmitted during a script execution. While the logging settings can and should be changed (on the Logging tab of the TSS Environment Variables dialog box) after scripts are fully developed and large loads are being simulated, the default settings are correct for determining the cause of script failures. If you're already familiar with these logs, feel free to jump to the next section.

These log files can be viewed in two different ways. The first way is to right-click a Fail (or a Pass) result in the Test Log in TestManager and then choose Properties (see Figure 6).

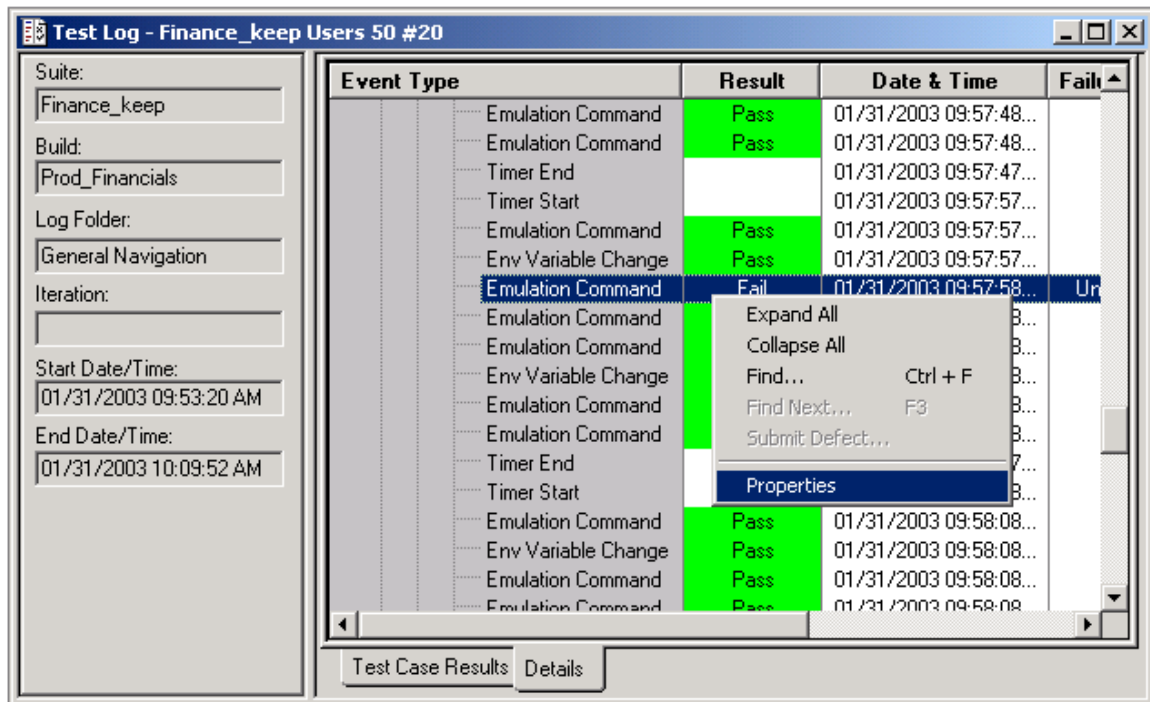


Figure 6: Test Log window in TestManager

Clicking on the Virtual Tester Associated Data tab will show you the view demonstrated in Figure 7.

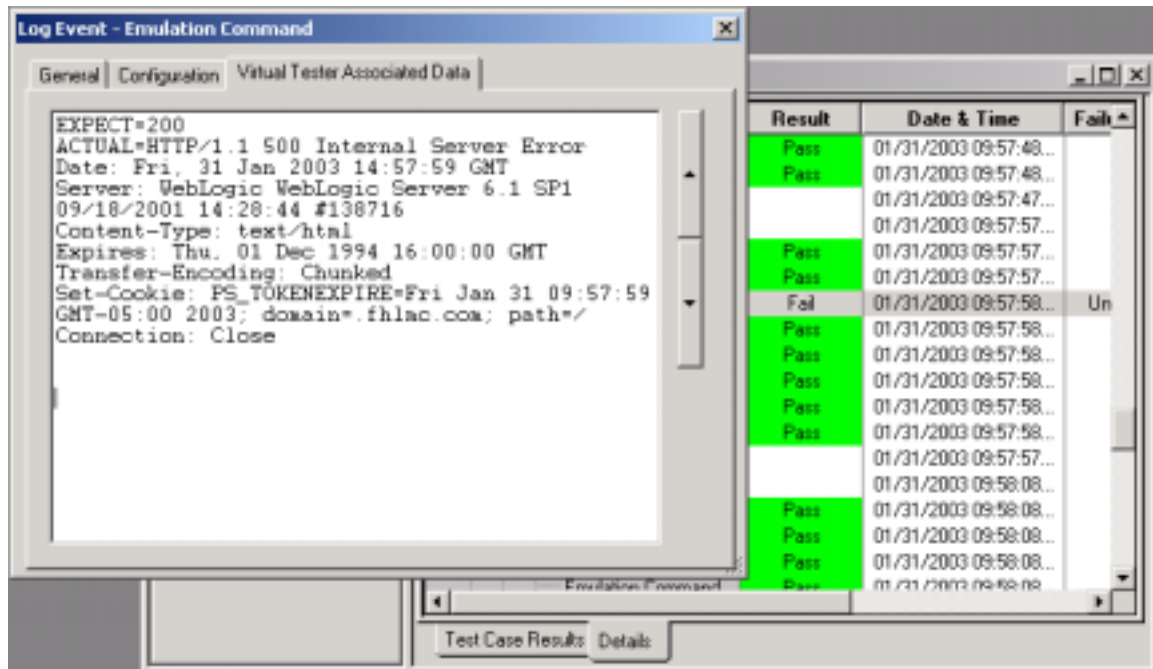


Figure 7: Virtual Tester Associated Data tab of Log Event window

As you can see in this case, the emulation command failed because TestManager was expecting an HTTP return code of 200 but received a return code of 500 instead. Viewing the logs this way allows you to easily navigate between requests and responses associated with a specific virtual user. Clicking the General tab gives you all of the information to identify that virtual user and the specific command ID for the request or response you're viewing. All of the information that's viewable through this interface in TestManager is stored in flat files in the project repository, which brings us to the other way to view these logs.

The second way to view native logs is to open the files that hold the data presented in the TestManager interface directly. Those files are located in the following directory:

```
[Drive]:\[RepositoryName]\TestDatastore\TMS_Builds\[BuildName]\[SubBuild]\[TestRun]\perfddata\
```

Three types of log files may be included in that directory:

- d00# — file(s) containing most of the data viewable in TestManager, one for each VU
- e00# — file(s) containing script execution errors, one for each VU
- o00# — file(s) containing custom output (discussed in the next section), one for each VU

(Note that unless you've edited your scripts to provide custom output, you won't find any o00# files in the directory.) To view any of these files directly, simply right-click, select "View With" (or "Open With" in XP), and select your favorite text editor (Notepad, Wordpad, and MS Word all work fine, though sometimes the files are too big for Notepad's buffer). While you won't find any new information in these files, it does present the information in a format that lends itself to searching, copying and pasting, and such. I use both viewing methods, depending on what I'm looking for.

Creating Custom Output

Creating custom output is a powerful yet simple way to evaluate what's really happening in a script. The easiest way to demonstrate custom output is with an example.

Let's say you have a script that logs a user on to your site, navigates content for a while, and logs off. Your script works just fine for one user, but at a hundred users you realize that 6% of your users are failing to log on. You suspect that six of the user names in your datapool are invalid, but you don't know which ones. You start scrolling through the test logs to find out which users got the logon error, then start reading through the request data to see which user names they have. This is a rather tedious process, as you can imagine.

The alternative is to create custom output. Immediately following the `stop_time` command for your logon page (arbitrary decision), place the following line of code:

```
printf("Virtual User # "+ itoa(_uid) +" was username "+datapool_value(DP1, 'username'));
```

Then compile the script and run it again. Upon completion, you'll find 100 o00# files in the `./perfddata/` directory. Each one of these files will contain text like this:

```
Virtual User # 17 was username Jamet
```

A quick glance at the User Start (User Group... line that you had to expand to see the individual command failures in the Test Log will show you which virtual user numbers were associated with the six failures. Luckily enough, you'll notice that the o00# file for virtual user #17 is o017. So all you have to do is open the o00# file that matches the virtual user number associated with the failed log on, and you've got the user name that you can now test manually in order to see if it was the cause of the error. In our example, the user name was supposed to be Janet, not Jamet. Though this is a simple example, I'm sure you can immediately see how this kind of custom output can be extremely useful in debugging scripts — especially scripts with custom code, variables, and match statements like we've discussed in previous articles.

Viewing Returned Pages After the Test Run

One of the most frequent questions I get while demonstrating or teaching VU scripting in Robot is "How do I see the pages to know if they came back correctly?" In fact, I remember asking that question myself not too awfully many years back. While the technique I'm about to share with you for viewing HTML pages returned during a performance test run after that test has been completed is a little tedious, it's quite simple and I use it all the time while debugging scripts. I suggest that you follow along by doing this exercise with me.

1. Launch Notepad and Internet Explorer, then return to TestManager.
2. Identify the page you want to view in the Test Log (generally by clicking a Timer Start or Timer End to view the timer name and verify it's the page you want).
3. Within that timer, there will be several Emulation Command and Env Variable Change entries. Click each one and look at the Virtual Tester Associated Data tab in the resulting dialog box to find the emulation command containing the main HTML of the page, normally starting with <head> (see Figure 8). This is generally (though not necessarily) the second emulation command after the Timer Start entry. The first emulation command after the Timer Start entry is usually the request and the second is usually the response containing the main HTML.

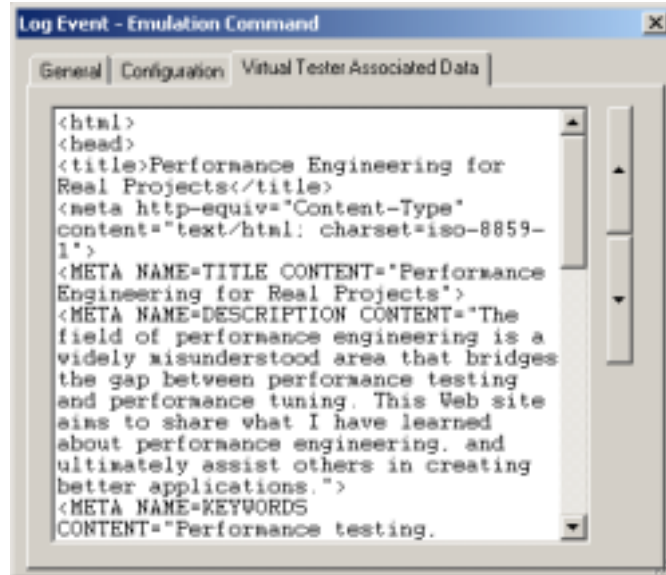


Figure 8: Virtual Tester Associated Data tab for an HTML page

4. Click somewhere inside the display box containing the HTML, then press CTRL+A to select all of the text and CTRL+C to copy it.
5. Navigate to Notepad and click the text area. If there's text there, press CTRL+A to select the text (you can omit this step if this is a new file), then press CTRL+V to paste the text from the Test Log into Notepad.
6. Press CTRL+S to save the file. When you do this, be sure to save it with a .htm or .html extension. I usually just name the file tmp.htm and save it to the root directory of my C:\ drive.
7. Navigate to Internet Explorer. On the menu bar, choose File > Open. Browse to the file you just saved and click OK. The Web page (minus graphics) will appear in your Web browser.

Figure 9 shows part of a PeopleSoft 8.4 data entry screen that was viewed in this fashion.

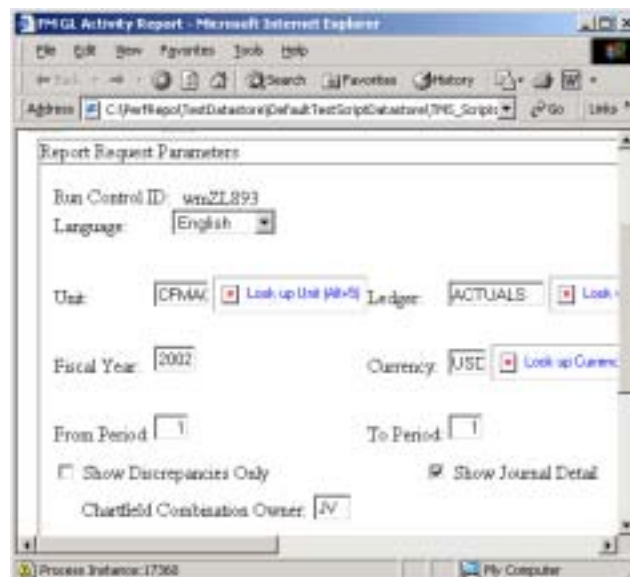


Figure 9: Web page generated from the Virtual Tester Associated Data

Would you rather try to find the bad value in this script, or in a browser window?

If you want to get fancy, you can use the `output_html` procedure in Listing 2 to actually save the HTML files independently while the script is running. The procedure creates files named by the user ID followed by the command ID of the request that generated the response and saves the contents of the `_response` (if it's in HTML) into that file. First you'll want to create a new directory called `pagefiles` in the `TMS_Scripts` directory. Here's the resulting path:

```
[Drive]:\[Repository]\TestDatastore\DefaultTestScriptDatastore\TMS_Scripts\pagefiles
```

Then you'll put this procedure either immediately following the `#include<VU.h>` line or in an included header file.

```
#include<VU.h>
proc output_html()
{
  if ((match('html', _response)) || (match('HTML', _response))) {
    z = open ("pagefiles\\"+_cmd_id+"_" + itoa(_uid) + ".html", "w");
    fprintf (z, "%s", _response);
    close(z);
  }
}
```

Listing 2: The `output_html` procedure

In your script, you'll want to call the `output_html` procedure after every `http_nrecv` command.

```
http_nrecv ["perftes~003"] 100 %% ; /* Internally Generated */
output_html();
```

Now after you run your script, you can navigate to the `./pageview/` directory and double-click any file to view the page (or frame) in a browser window.

Please note that neither the procedure nor the Test Log assembles the frames into complete pages or includes graphics files in the pages you'll be viewing. Don't waste any time wondering if this is an error.

Viewing Pages During the Test Run

A couple of months ago I mentioned to my office buddy, Chris Walters, that someone on the RDN forums had asked whether you can view TestStudio VU scripts while they're running, like you can using Mercury LoadRunner. That's one feature that's been on the wish list of virtually everyone I know who uses TestStudio for performance testing. Well, lo and behold, Chris soon sent me an e-mail with some files and a list of instructions (with no explanation of what it was about). I followed the instructions, and there I was watching my script run real-time in a browser! The instructions went something like this:

1. Create a directory called `perflive` in the `TMS_Scripts` directory.

```
[Drive]:\[Repository]\TestDatastore\DefaultTestScriptDatastore\TMS_Scripts\perflive
```

2. Copy these files into it: `options.html`, `perfpagel.html`, `perfview.html`.

3. Copy the `view` procedure in Listing 3 into your script (or header file, as we've done previously).

```
#include<VU.h>
proc view() {
  if ((match('html', _response)) || (match('HTML', _response))) {
    string filename;
    sprintf(&filename, "perflive\\perfpagel.html", _uid);
    file = open(filename, "w");
    fprintf (file, "%s", _response);
    close (file);
  }
}
```

Listing 3: The `view` procedure

4. Call `view` immediately following every `http_nrecv` in your script.

```
http_nrecv ["perftees~003"] 100 %% ; /* Internally Generated */
view();
```

5. Open `perfview.html` in Internet Explorer.

6. Run your script.

7. Watch the Web browser.

The final page I saw looked something like Figure 10.

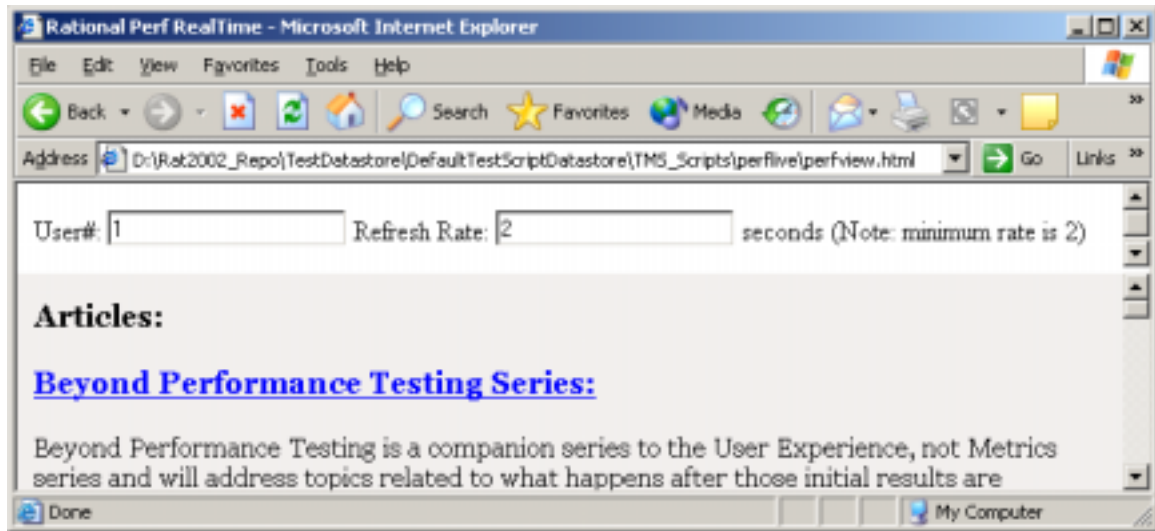


Figure 10: Sample view of `perfview.html`

It's pretty self-explanatory, really. Now all you have to do is decide which user you want to monitor, type in the user number, and watch this user go.

This is obviously very helpful in debugging scripts as well as visualizing errors in the application. If you use this in conjunction with the Test Log and/or the `output_html` procedure, you can both see what's happening as it happens and review the code or view the HTML in a browser at your leisure after the completion of the test execution.

I did add one fancy thing to Chris's code after using it for a while. I wanted to be able to make the `view` and/or the `output_html` calls optional and easy to enable or disable before any test execution for any script. Immediately following the common declarations in the script, I added two integers that I use as flags, `view_flag` and `output_flag`. See Listing 4.

```
push Timeout_val = Min_tmout;
push Think_avg = 0;
```

```
int view_flag = 1; /*1 for view during script execution, 0 for do not view */
int output_flag = 1; /*1 for output during script execution, 0 for do not save
output files */
```

Listing 4: Flag declarations

Then I changed all the `view` and `output_html` procedure calls to be part of `if` statements.

```
if (view_flag == 1) view();
if (output_flag == 1) output_html();
```

This way all I have to do is change the value of the flags (from 1 to 0 or vice versa) to either enable or disable the desired functionality based on my needs for a particular test run.

Stepping Through Scripts

There's one last debugging feature I'd like to present to complete our discussion — stepping through scripts. If you're familiar with GUI scripting in Robot, you're probably very used to stepping through scripts line by line as a debugging technique. VU scripting has that same functionality, but it's rarely used because you can't really see what the response from the emulation command was to know if you've just stepped into (or right past) the error.

This functionality has new value when used in combination with our `view` procedure. To step through a script with the `view` procedure enabled, simply set up your script for view and execute it with a `perfview.html` open in a browser as we did earlier in the article. As soon as TestManager finishes initializing your script, choose Monitor > Suspend Test Run from the menu bar. See Figure 11.

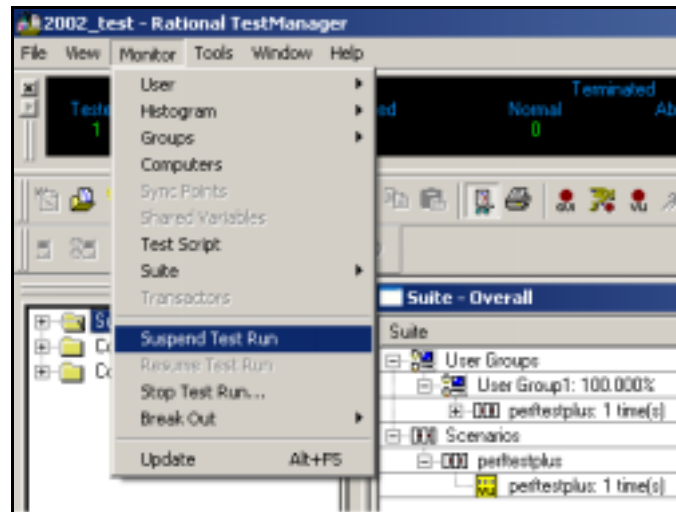


Figure 11: Monitor menu options in TestManager

Then choose Test Script from the Monitor menu to display the Test Script View. From there you can resume running your script and suspend it again after resuming or step through the script using either the single-step or multi-step option. See Figure 12.

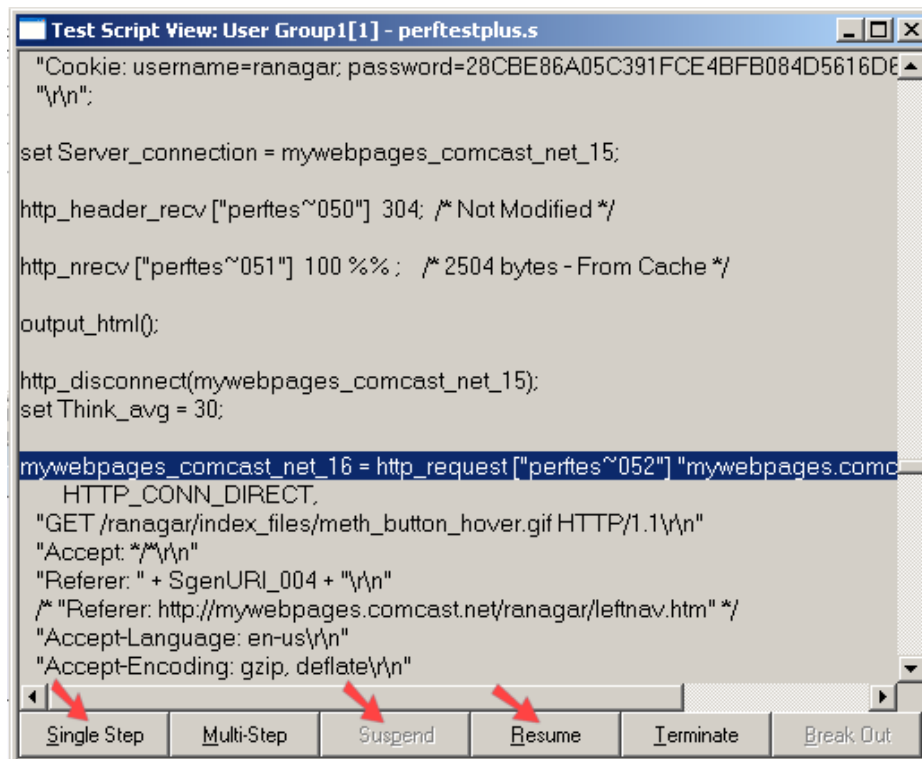


Figure 12: Test Script View in TestManager

For our purposes, we want to click Single Step (which causes the script to execute the next emulation command, then wait for our input), then check our browser with `perfview.html`. As we continue to click and check, we see that now we have complete control over when the script takes the next step. We can see the emulation command that's being executed and the returned HTML all at the same time.

Once again, I'm sure I don't have to spell out how useful this combination is in determining whether an error is due to the script or the application, and if it's due to the script, how easy this makes it to narrow down the possible causes of that script error.

Summing It Up

Historically, it's often been difficult to determine the actual root cause of a script failure from the obvious symptoms. Thinking about some of the common causes for failure can help when trying to track down that root cause. Coupling that thought process with the added ability to view the relevant Web pages in a browser both in real time and subsequent to a test run makes the process significantly easier.

Acknowledgments

Thanks go to Chris Walters for developing and providing the code for the `view` function and assisting with other technical aspects of this article.

About the Author

Scott Barber is a senior consultant for [Noblestar](#) and a member of Noblestar's specialty testing lab, NobleLabs. NobleLabs provides a variety of services in such areas as performance engineering, security engineering, and embedded device interoperability testing. With a background in network architecture, systems design, database design and administration, programming, and management, Scott has become a recognized thought leader in the field of performance engineering. Before joining Noblestar, he was a company commander in the United States Army and a government contractor in the transportation industry.

Scott is an active participant in the [Rational Developer Network public forums](#) and a moderator for the performance testing and Rational TestStudio related forums on [QAForums.com](#). [Scott's Web site](#) complements this series. Please visit it to find more detail on some topics and view slides from various presentations he's given recently. You can address questions/comments to him on either forum or contact him directly via [e-mail](#).