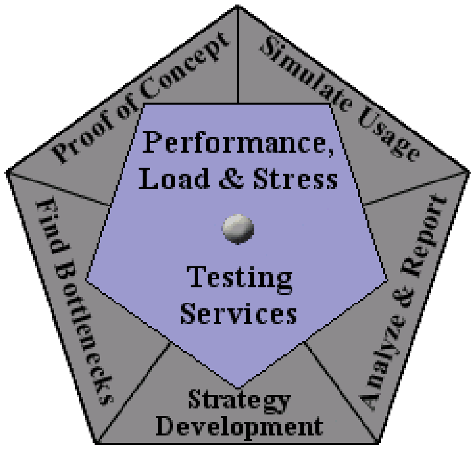
PerfTestPlus

Better Testing... Better Results

User Experience, Not Metrics

by:

R. Scott Barber



Part 5: What should I time and where do I put my timers?

"Even with the response-time market in a continuing stage of development, enterprises should deploy application response-time tools now. The measurements from these tools should be used to understand the response-time characteristics of the most-critical applications, to help establish and meet service-level goals, and to correlate the performance experienced by external users with customer satisfaction." *End-to-End Application Response Times: Market Update* Gartner Inc. Oct 2000

Since this article was published in Gartner, I have been using this quote shamelessly to promote a transition in thought from component-based measurement to User Experience measurements. Though thought transitions don't happen overnight, today more and more managers and stakeholders are asking for User Experience measurements and validation over the more traditional component based performance metrics.

Our previous three articles were collectively about Modeling Real Users. I am assuming that you have read these articles and now understand how to determine and model realistic usage patterns, and agree that a test can only ever be as accurate as its model. Using that as a starting point, we now move into our next group of three articles, which are collectively titled Meaningful Times. This topic will focus on capturing and interpreting meaningful times during a load test. In the third collection of three articles we will discuss how to organize your times and interpretations into easy to understand reports for project stakeholders. This month's edition of the "User Experience, Not Metrics" article series discusses how to use Rational TestStudio to collect accurate and meaningful User Experience measurements through the use of timers. Rational's TestStudio is one of the market leading application response-time tools available today.

If you are new to performance testing, or to this series, it is imperative that you understand the concepts presented in Part 2 of this series where we discussed how to model individual user delays, since in many ways this article is a continuation from that one. The intent of this article is to discuss as well as demonstrate the use of TestStudio based on years of experience. The article is intended for all levels of TestStudio users, but will be most useful to Intermediate tool users and above.

What Should I Time?

One critical part of Performance Testing is collecting meaningful measurements. If the wrong measurements are collected at the wrong time, bottlenecks may be missed, results may be misleading, and measurements may not correlate to actual user-experience. Meaningful times can also aid in the process of tracking down bottlenecks that are detected. Tracking and fixing bottlenecks, often referred to as Performance Tuning, is not explicitly discussed in this series of articles. It is worth mentioning that all of the concepts and methods discussed in this series have been derived through dozens of Performance projects that all included actual tuning of systems.

The entire reason I started giving the talk that lead to this article series was to show people the value of user experience measurements over component based metrics. My argument was, and still is, that it doesn't matter how well each component is performing independently if the overall system performs poorly. Component based testing and tuning has been touted over the years as being synergistic, implying that if each component is tuned to its best possible performance that the system as a whole will perform acceptably. This simply isn't a valid assumption. Until a system is loaded with a realistic user load and observed under this load, over time, one cannot predict what an actual user's experience will be. It is often valid to assume that if each component is tuned to its best possible performance, the system as a whole can't be further tuned without re-architecting at least some part of the system. Even in this case, stakeholders need to know how many users the system can support before performance becomes unacceptable. For these reason, I started my own personal crusade to make the IT world aware of the value of user experience measurements, a.k.a. end-to-end system response times.

Please don't infer from the last paragraph that I oppose component based testing, because I don't. What I oppose is EXCLUSIVE component based testing. Testing components individually is a necessary part of the tuning process. In the next few paragraphs you will see how component based measurements compliment user experience measurements to create a complete picture of system performance as a whole.

There are two popular approaches to performance measurement collection. These are Top-Down and Bottom-Up approaches. We will explore these approaches in detail below. A top-down approach is the right place to start when the actual performance of a system or application is unknown, or when performance is known, but no bottleneck has been determined. A bottom-up approach is appropriate once a specific bottleneck has been identified. How does this relate to User Experience measurements? Simple. The Top-Down approach starts and ends with User Experience measurements, while the Bottom-Up approach starts with component based measurements and ends with User Experience measurements. In our examples, the first thing we need to do is identify IF there are any bottlenecks and if so WHERE they are located. Therefore, we'll start our discussion with the top-down approach.

Top-Down Approach

Simply put, a top-down approach to measuring performance begins by determining actual user experience and only moves into further analysis where more granular measurements are collected when poor performance is detected. User-experience measurements are easy to conceptualize. These measurements capture the time that passes from when a user takes a particular action to when the user sees the results of that action. This applies for all types of applications, be they web, client server, embedded software, etc. For example, if you were to time how long it took for Part 2 of this series to fully load when you clicked on the link in the beginning of this article that would be a user experience measurement.

Generally, when conducting a performance test, very clear performance goals have been established. The goals should be focused on an acceptable user experience and may look something like this:

> "All static pages on the website will display in 8 seconds or less on a 56.6 kbs modem 95% of the time while 500 users are accessing the site according to the documented user community model."

This is clearly a requirement that implies a top-down approach. It is easy to see that page generation, communications between the client and the server, and client side processing time is what needs to be measured in the case of this requirement. . For a web-based application, this will be the most common starting measurement. These measurements are collected with Rational TestStudio using timers. We will discuss placement of these timers below.

When these timers reveal measurements that do not meet the performance goals, the next level of measurements is needed. Luckily, TestStudio collects this next level of detail automatically. These measurements are known as individual transaction times, identified by command ids. Since we have an entire article dedicated to consolidating and interpreting collected measurements, we aren't going to discuss these times in detail here. We'll look at how TestManager organizes and displays these collected times in the Viewing Times section below.

Bottom-Up Approach

A bottom-up approach would be appropriate if, for example, it was determined that only user transactions requiring a database search yielded unacceptable performance. In a case like this, TestStudio could be used to record and playback database searches directly against the database. This would be done without going through the client computer, the web server, or the application server. This kind of load would not be modeled to match real user patterns and the measurements collected during this kind of test would be limited to the database response time. In this way, the database could be tuned for best possible performance while being searched. The actual measurements collected in this case would be individual transaction measurements. At the conclusion of component tuning, in this case of database searches, user experience measurements must be collected to determine if the overall performance goals have been met.

Occasionally, a system or application will have specific component load or performance requirements. For example, a database may be required to sustain 30 transactions per second without queuing. This often comes into play when an individual component of the system is outsourced. When a component is outsourced, the provider must be held to very specific performance requirements. This is usually done by establishing specific service level agreements, and by periodically testing the system to validate compliance. An example of a performance requirement of a service level agreement for an outsourced database follows:

> "All database transactions will be processed in 2 seconds or less, 95% of the time while the database is receiving 5 or fewer requests per second."

It is well beyond the scope of this article to discuss how to do the kind of testing I just described, or to discuss how to tune our theoretical database. If you find yourself in a position like the one above, where you have pinpointed a bottleneck, and created a repeatable test to exploit that bottleneck, but don't know how to fix it, you can always do what I do… Call the smartest person you know and offer to buy them lunch for some insight!

If these types of measurements are not granular enough to detect or tune a particular bottleneck, you may need to look to another tool, such as a code profiler, to collect what are known as "white box" measurements. TestStudio can only capture hardware statistics and times for transactions that occur over a network of some sort. If the bottleneck is actually inside a piece of code, that is where profilers can be used. Rational has several tools available that may be appropriate for this level of analysis, such as Purify, Quantify, and PureCoverage. These tools are part of the Rational TestStudio package, or are available separately. These are generally considered to be developer tools, but I felt it was appropriate to mention them here for completeness.

Individual Transactions

We have mentioned Individual Transactions in both of the above sections, so I thought we'd discuss what an individual transaction is in a little detail by themselves. Individual transactions are actually a part of every performance testing approach. Whether testing a website, a client server application, or just the database component of a multi-tier application, every communication that occurs between computers is made up of individual transactions. A web page may be generated in a single transaction, but most web pages are actually painted as a result of many (I have seen as many as 46) individual HTTP requests, or transactions. These transactions represent each individual object that must be retrieved from the server to then display to the user. A client server application may handle everything except database transactions on the client side, so there may only be one measurable transaction per user interaction with the system.

This is why when you insert timers into your script while recording, you often see many commands in the script between the start and stop times. If you refer to the script we created during Part 2 you will notice that Page 1 generated 3 individual transactions, or `http_request` commands and Page 2 generated 12 individual transactions. You can see that these individual transactions, when viewed collectively, yield a user experience measurement, but when viewed individually they do not. Individually, they represent just on part of that user experience. It is valuable to see these individual transactions because they can often show you, for example, whether the slow component of a web page is a graphic, or the database search.

Approach Summary

Determining when to start with which approach is sometimes not clear. As an example, let's consider the online bookstore from our previous articles. At some point, if a customer has decided to order, they must start by entering their billing information. Since we don't know how well this function within the application will perform, we would begin with a top-down approach by collecting a user-experience measurement. This measurement is found by capturing the time elapsed between the moment the user clicks the submit button, until the "your transaction has been approved" message appears on the next page. If that user transaction is performing unacceptably, then a bottom-up approach may be taken, with focus on the components that are involved in completing the transaction. In this case, one measurement that may be useful is capturing the time between when the application server sends the update statement to the database and when the database acknowledges the update. This transaction is just one of many components to the complete user transaction. It makes no sense to capture this time unless it can be determined that the database transaction may be the actual bottleneck. So you can see that no matter which approach you start with, you will likely use both during the course of performance testing, but will always end with user experience measurements.

Where do I put the Timers?

Top down, bottom up, either way we eventually want to collect user experience measurements. In this section we'll explore several methods of placing timers while recording or editing VU scripts. The following sections are written with the assumption that the reader is familiar with inserting blocks and timers into a script while recording. Information about customizing toolbars and inserting blocks and timers into a script is available in the documentation that came with your software.

Blocking User Activity

Before we actually determine where to place our timers we need to determine what user activity goes inside and outside the timers. In most cases this is a fairly simple task. Whenever a user clicks a button or a link that causes communication back to the server, it needs to be captured and timed. On a static website, each timer STARTS immediately before performing an activity that leads to a new page being generated and STOPS as soon as that page is completely painted. This gets a little more complicated when there are active components on the client side of the application. It is often unclear to the tester whether processes (like data validation upon submittal or active menu bars that change based on mouse position) invoke client side or server side processing. If you have access to a developer, they can tell you what invokes server side processing and what doesn't. If you don't have access to a developer, you just have to experiment. Remember that VU scripts don't actually measure client side processing time during playback, but do indirectly collect that time during recording.

If you don't know what does and doesn't invoke server side processing, the easiest way to find out is to block everything with blocks or timers while recording and manually delete timers in your script that have no activity between the start and stop commands.

This may be easier seen with an example. Let's return to www.noblestar.com. On the homepage, we notice that as we hover over the menu selections in the top menu bar, the bottom menu bar changes dynamically based on which main heading is currently highlighted. Initially I thought this was all client side processing, but I later found that it wasn't. When I recorded the following script I found that I was wrong:

1) Start a timer, or block timer, named "Home Page" and launch the browser going directly to noblestar.com

2) Stop the timer as soon as the page is loaded.

3) Start a new timer named "Menus"

4) Move your mouse around the screen, ensuring that you highlight several menu options, but don't click on any

5) Stop the timer

6) Start a new timer called “Page1”

7) Move directly to a menu option or link and click it

8) Stop the timer as soon as the next page is loaded

9) End the recording.

When I did this exercise, I got the attached script. As you can see in this script, there were a lot of requests and receives recorded inside the “Menus” timer. The resulting question is “so where to I put my timers?”

Part of this answer is obvious. The “HomePage” timer was captured correctly. While you could enter timers that block hovering over each menu selection, in this case that would be overkill. In truth the transactions that produce the dynamic menu are very few, very fast, and cached the first time. If we use block timers (discussed in detail below) rather than timers, we can stop the “HomePage” block timer, move the mouse to highlight the link that we want to navigate to next, then start the “Page1” block timer. The key is to ensure that the link you want to click on is visable on the screen before starting the timer, and ensuring that you don’t hover over a menu option that will change what is visable on the screen before clicking the link. This method logically separates the script into two distinct page load times and an identifiable set of individual transaction times that can be analyzed for the menu transactions. See the entire script here or the listing below for a section of the script that shows just the block timers and a small selection of command id’s.

```
/*
      ->-> Session File Information <-<-
*/

#include <VU.h>
{
push Http_control = HTTP_PARTIAL_OK | HTTP_CACHE_OK | HTTP_REDIRECT_OK;
push Timeout_scale = 200; /* Set timeouts to 200% of maximum response time */
push Think_def = "LR";
Min_tmout = 120000;       /* Set minimum Timeout_val to 2 minutes          */
push Timeout_val = Min_tmout;
/* Start_Block "Home Page" */

push Think_avg = 0;

www_noblestar_com_2 = http_request ["Home Pa001"] "www.noblestar.com:80",
      HTTP_CONN_DIRECT,
   "GET / HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "\r\n";
   { string SgenURI_001; }
   SgenURI_001 = _reference_URI; /* Save "Referer:" string */

start_time ["Home Page"] _fc_ts;

set Server_connection = www_noblestar_com_2;

http_header_recv ["Home Pa002"]  200;     /* OK */

http_nrecv ["Home Pa003"]  100 %% ;       /* 1178 bytes */


...

stop_time ["Home Page"]; /* Stop_Block */

set Think_avg = 20119;

set Server_connection = www_noblestar_com_1;

 /* Keep-Alive request over connection www_noblestar_com_1 */
http_request ["Noblest~001"]
   "GET /images/spacer.gif HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Referer: " + SgenURI_003 + "\r\n"
   /* "Referer: http://www.noblestar.com/global/top.html" */
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "If-Modified-Since: Fri, 29 Dec 2000 18:58:43 GMT\r\n"
   "If-None-Match: \"457fe811-1-55-3a4cdee3\"\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "Cookie: NSES40Session=94f%253A3ca25b7e%253A53c78693dbcf7e9d\r\n"
   "\r\n";

http_header_recv ["Noblest~002"]  304;    /* Not Modified */

http_nrecv ["Noblest~003"]  100 %% ;      /* 85 bytes - From Cache */


...

http_request ["Noblest~109"]
   "GET /images/enterprise_on.gif HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Referer: " + SgenURI_003 + "\r\n"
   /* "Referer: http://www.noblestar.com/global/top.html" */
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "If-Modified-Since: Thu, 31 Jan 2002 22:30:19 GMT\r\n"
   "If-None-Match: \"654a4790-1-12e-3c59c57b\"\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "Cookie: NSES40Session=94f%253A3ca25b7e%253A53c78693dbcf7e9d\r\n"
   "\r\n";

http_header_recv ["Noblest~110"]  304;    /* Not Modified */

http_nrecv ["Noblest~111"]  100 %% ;      /* 302 bytes - From Cache */
/* Start_Block "Page1" */

set Think_avg = 9273;

set Server_connection = www_noblestar_com_1;

 /* Keep-Alive request over connection www_noblestar_com_1 */
http_request ["Page1001"]
   "GET /images/custom_off.gif HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Referer: " + SgenURI_003 + "\r\n"
   /* "Referer: http://www.noblestar.com/global/top.html" */
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "If-Modified-Since: Thu, 31 Jan 2002 22:30:18 GMT\r\n"
   "If-None-Match: \"f5065728-1-136-3c59c57a\"\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "Cookie: NSES40Session=94f%253A3ca25b7e%253A53c78693dbcf7e9d\r\n"
   "\r\n";
start_time ["Page1"] _fs_ts;


http_header_recv ["Page1002"]  304; /* Not Modified */

http_nrecv ["Page1003"]  100 %% ;   /* 310 bytes - From Cache */


...

stop_time ["Page1"]; /* Stop_Block */

}
```

Listing 1: Portion of Noblestar_block_example script, as recorded

In the example below, we will execute this script and look at how the results are organized.

Timers vs. Block Timers

So far in this article, we have been oscillating between the use of timers and block timers. Fundamentally, we know that these serve the same overall function. They both time the "stuff" that happens between the start and stop commands in the script. There are a few differences that are worth discussing though. If you record a script and start both a timer and a block timer at the same time, and stop them both at the same time, you will end up with script segments that look like the listings below. In general testers will choose one method over the other based on personal preference of how they prefer the labeling of individual transactions.

```
/* Start_Block "Home Page" */

push Think_avg = 0;

www_noblestar_com_2 = http_request ["Home Pa001"] "www.noblestar.com:80",
     HTTP_CONN_DIRECT,
   "GET / HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "\r\n";

   { string SgenURI_001; }
   SgenURI_001 = _reference_URI; /* Save "Referer:" string */

start_time ["Home Page"] _fc_ts;

set Server_connection = www_noblestar_com_2;

http_header_recv ["Home Pa002"]  200;     /* OK */

http_nrecv ["Home Pa003"]  100 %% ;       /* 1178 bytes */

set Think_avg = 1602;


…

stop_time ["Home Page"]; /* Stop_Block */
```

Listing 2: Portion of Noblestar_block_example script, as recorded

```
start_time ["Home Page"];

push Think_avg = 0;

www_noblestar_com_2 = http_request ["Noblest~001"] "www.noblestar.com:80",
     HTTP_CONN_DIRECT,
   "GET / HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "\r\n";

   { string SgenURI_001; }
   SgenURI_001 = _reference_URI; /* Save "Referer:" string */
```

```
set Server_connection = www_noblestar_com_2;

http_header_recv ["Noblest~002"]  200;    /* OK */

http_nrecv ["Noblest~003"]  100 %% ;      /* 1178 bytes */

set Think_avg = 11582;

…


stop_time ["Home Page"];
```

Listing 3: Portion of Noblestar_menu_example script, as recorded

If you look closely at these two script segments, you will notice there are two differences, but are otherwise identical. You can see that the two start times seem to begin in different places, and that the start time command in Listing 3 has no extension, while the start block command in Listing 2 has an extension of _fs_ts. The simple explanation is that these are actually two different ways of scripting exactly the same thing.

Let me give you a slightly more thorough explanation of a very complex topic: The start_time command supports 6 extensions. These extensions specify the point during a transmission where the timing begins. They are:

_fc_ts which stands for First Connect Timestamp

_fs_ts which stands for First Send Timestamp

_fr_ts which stands for First Receive Timestamp

_lc_ts which stands for Last Connect Timestamp

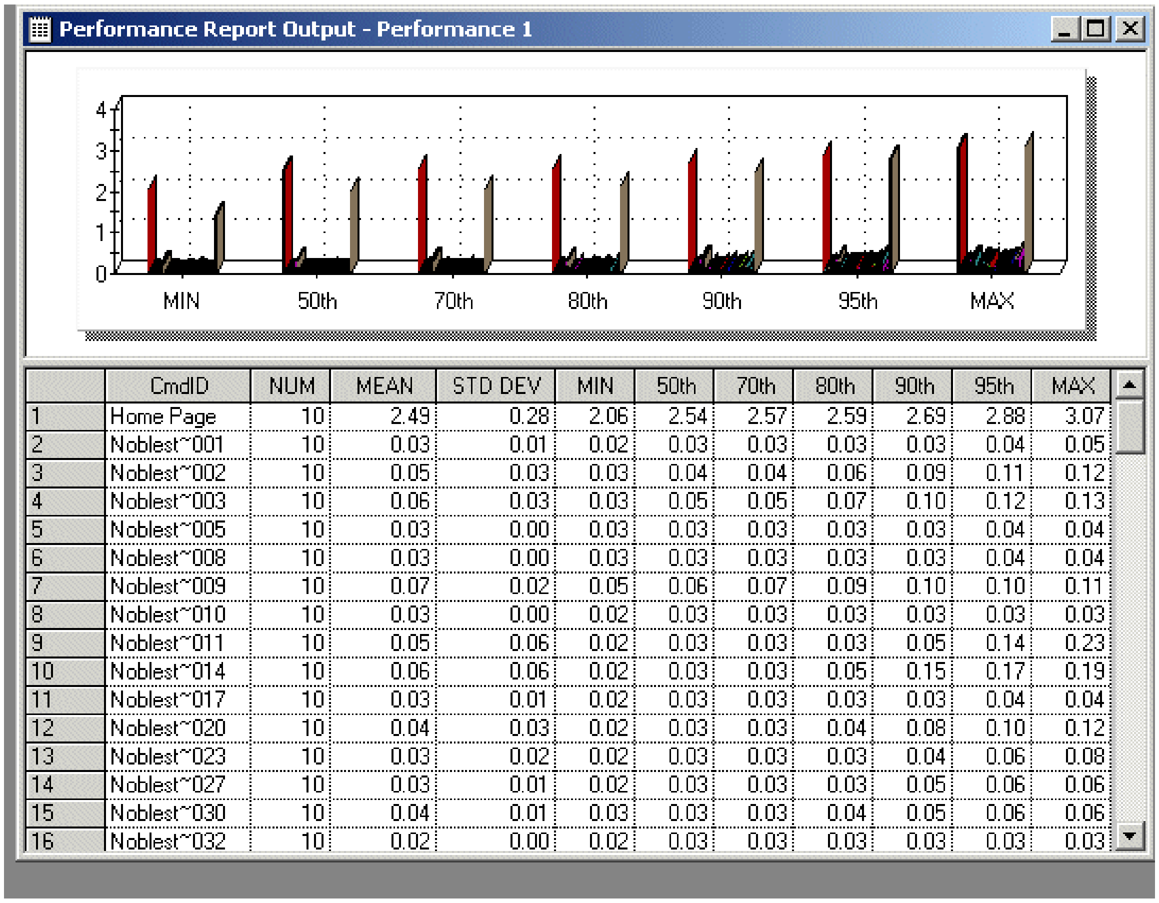
_ls_ts which stands for Last Send Timestamp

_lr_ts which stands for Last Receive Timestamp

When no extension is specified, the script defaults to _fs_ts and, thus, when the two scripts above are executed, they both time the same thing. I debated including a detailed explanation of the six extensions and their various uses, but quickly realized that it would be too ambitious of an undertaking as an add-on to this article.

The second difference we will notice between the two script segments is that Listing 2 shows command ids inside the timer as [“Home Pa001"], and Listing 3 show them as ["Noblest~001"] When a timer is used, the Command ID label is always the first 8 characters of the name of the script followed by a sequential number (or the first 7 followed by a ~ if the name of the script is longer than 8 characters). When a block is used, the Command ID labels inside the block are the first 8 characters of the name of the timer block, rather than the name of the script, followed by the sequence number. This may not seem significant at first, but next month when we start interpreting times, we will see how much easier it is to track down bottlenecks when we can see which timer each individual transaction time belongs to.

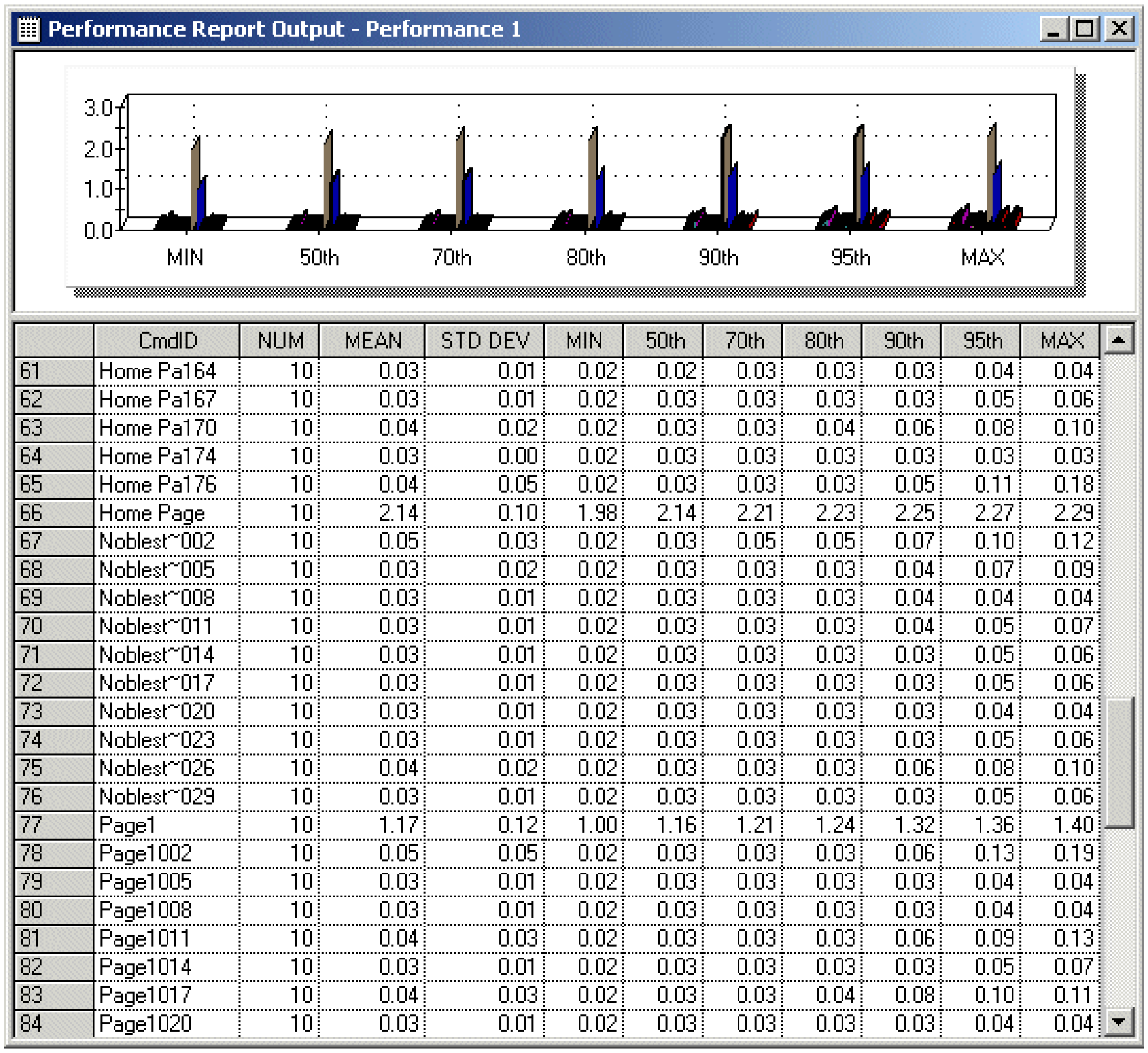
When I execute the Noblestar_menu_example script, for 1 user over 10 iterations, I get the following results in TestManager:



	CmdID	NUM	MEAN	STD DEV	MIN	50th	70th	80th	90th	95th	MAX
1	Home Page	10	2.49	0.28	2.06	2.54	2.57	2.59	2.69	2.88	3.07
2	Noblest~001	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.05
3	Noblest~002	10	0.05	0.03	0.03	0.04	0.04	0.06	0.09	0.11	0.12
4	Noblest~003	10	0.06	0.03	0.03	0.05	0.05	0.07	0.10	0.12	0.13
5	Noblest~005	10	0.03	0.00	0.03	0.03	0.03	0.03	0.03	0.04	0.04
6	Noblest~008	10	0.03	0.00	0.03	0.03	0.03	0.03	0.03	0.04	0.04
7	Noblest~009	10	0.07	0.02	0.05	0.06	0.07	0.09	0.10	0.10	0.11
8	Noblest~010	10	0.03	0.00	0.02	0.03	0.03	0.03	0.03	0.03	0.03
9	Noblest~011	10	0.05	0.06	0.02	0.03	0.03	0.03	0.05	0.14	0.23
10	Noblest~014	10	0.06	0.06	0.02	0.03	0.03	0.05	0.15	0.17	0.19
11	Noblest~017	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.04
12	Noblest~020	10	0.04	0.03	0.02	0.03	0.03	0.04	0.08	0.10	0.12
13	Noblest~023	10	0.03	0.02	0.02	0.03	0.03	0.03	0.04	0.06	0.08
14	Noblest~027	10	0.03	0.01	0.02	0.03	0.03	0.03	0.05	0.06	0.06
15	Noblest~030	10	0.04	0.01	0.03	0.03	0.03	0.04	0.05	0.06	0.06
16	Noblest~032	10	0.02	0.00	0.02	0.03	0.03	0.03	0.03	0.03	0.03

Figure 1 – Noblestar_menu_example Results

When I execute the Noblestar_block_example script, for 1 user over 10 iterations, I get the following results in TestManager:



	CmdID	NUM	MEAN	STD DEV	MIN	50th	70th	80th	90th	95th	MAX
61	Home Pa164	10	0.03	0.01	0.02	0.02	0.03	0.03	0.03	0.04	0.04
62	Home Pa167	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.06
63	Home Pa170	10	0.04	0.02	0.02	0.03	0.03	0.04	0.06	0.08	0.10
64	Home Pa174	10	0.03	0.00	0.02	0.03	0.03	0.03	0.03	0.03	0.03
65	Home Pa176	10	0.04	0.05	0.02	0.03	0.03	0.03	0.05	0.11	0.18
66	Home Page	10	2.14	0.10	1.98	2.14	2.21	2.23	2.25	2.27	2.29
67	Noblest~002	10	0.05	0.03	0.02	0.03	0.05	0.05	0.07	0.10	0.12
68	Noblest~005	10	0.03	0.02	0.02	0.03	0.03	0.03	0.04	0.07	0.09
69	Noblest~008	10	0.03	0.01	0.02	0.03	0.03	0.03	0.04	0.04	0.04
70	Noblest~011	10	0.03	0.01	0.02	0.03	0.03	0.03	0.04	0.05	0.07
71	Noblest~014	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.06
72	Noblest~017	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.06
73	Noblest~020	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.04
74	Noblest~023	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.06
75	Noblest~026	10	0.04	0.02	0.02	0.03	0.03	0.03	0.06	0.08	0.10
76	Noblest~029	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.06
77	Page1	10	1.17	0.12	1.00	1.16	1.21	1.24	1.32	1.36	1.40
78	Page1002	10	0.05	0.05	0.02	0.03	0.03	0.03	0.06	0.13	0.19
79	Page1005	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.04
80	Page1008	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.04
81	Page1011	10	0.04	0.03	0.02	0.03	0.03	0.03	0.06	0.09	0.13
82	Page1014	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.05	0.07
83	Page1017	10	0.04	0.03	0.02	0.03	0.03	0.04	0.08	0.10	0.11
84	Page1020	10	0.03	0.01	0.02	0.03	0.03	0.03	0.03	0.04	0.04

Figure 2 – Noblestar_block_example Results

At first glance the timers look identical. Looking closer we see that the individual command ids in Figure 2 are easily identifiable as to which timer they reside in . Conversly in Figure 1, the only way to associate a command id to a specific timer is to return to the script, find the command in question, then search the script to see what timer it comes immediately after.

Nesting Timers

For the sake of completeness, I felt the need to add this short section to address nested timers. From the very first time I learned about using Rational Tools for Performance Testing, I was told to never nest timers. Through a significant amount of research and testing I have found an exception to the "don't nest timers" rule that I'd like to share with you. First, let me describe what I mean by nested timers. If, for some reason, you were to start timer A, then start timer B, then stop timer A, then stop timer B, that would be a nested timer. Generally, there is no need to try to nest timers if you are measuring page load times for web pages. However, I have found that it sometimes adds value to put timers around individual frames in web pages as well as around the entire page. Now, I admit, to do that, you have to be able to manually determine what command ids are part of which frame on your own, and that isn't the point I am trying to make. The actual point is, if you have a need to put timers inside of timers, the order of the starts and stops makes a difference. For timers in timers to return the correct values it is important to organize them so that one timer is completely enclosed within the other, and not staggered like in the example above. For example, below you see timers completely enclosed within other timers, which would yield correct results:

```
start_time  ["Outer"];
        start_time ["Inner1"];
        stop_time ["Inner1"];
        start_time ["Inner2"];
        stop_time ["Inner2"];
stop_time ["Outer"];
```

This example shows timers with staggered start and stop times that will yield incorrect results.

```
start_time  ["Outer"];
        start_time ["Inner1"];
        start_time ["Inner2"];
stop_time ["Outer"];
        stop_time ["Inner1"];
        stop_time ["Inner2"];
```

I would love to be more specific about why the results are inaccurate, but I have yet to completely figure out the pattern. If you want to test this yourself, create a script with the timers listed above, and put varying delays between each timer command and compare the results with the known delay times in TestManager.

Manipulating Think_avg's

To date, we haven't discussed Think_avgs in our script in great detail. If you look at the set think_avg times in Listings 2 and 3, you will notice that in Listing 2 there is a 1.6 second think_avg and in Listing 3 there is an 11.6 second think_avg – in both cases, they are inside the timer. Also in both cases, it should be obvious that there was not 1, or 11 seconds of client processing time involved in the Home Page transaction. In Part 2 we discussed substituting delays for think_avgs to simulate user think times. Once you have done that, it is important to search through the rest of your script and adjust the remaining think_avgs to ensure they reflect client side processing time, not user think time inside the timers.

Impact on User Delay Times

In Part 2 of this series we discussed how to insert delay times when using timers, not block timers. In the case of block timers, all of the same theories apply, but the delays need to be place immediately after the previous stop timer command, rather than immediately before the start timer command. See the listing below.

```
/* Start_Block "Home Page" */

push Think_avg = 0;

www_noblestar_com_2 = http_request ["Home Pa001"] "www.noblestar.com:80",
      HTTP_CONN_DIRECT,
   "GET / HTTP/1.1\r\n"
   "Accept: */*\r\n"
   "Accept-Language: en-us\r\n"
   "Accept-Encoding: gzip, deflate\r\n"
   "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
   "Host: www.noblestar.com\r\n"
   "Connection: Keep-Alive\r\n"
   "\r\n";

   { string SgenURI_001; }
   SgenURI_001 = _reference_URI; /* Save "Referer:" string */

start_time ["Home Page"] _fc_ts;

set Server_connection = www_noblestar_com_2;

http_header_recv ["Home Pa002"]  200;     /* OK */

http_nrecv ["Home Pa003"]  100 %% ;       /* 1178 bytes */

// set Think_avg = 1602;


…

stop_time ["Home Page"]; /* Stop_Block */

delay(uniform(2000,8000));
```

Listing 4: Portion of Noblestar_block_example script, modified

Now You Try It

Rather than writing separate exercises to demonstrate the use of timer blocks and timer, I recommend that you follow the steps I have outlined in this article to replicate the scripts I created for this article. You are also welcome to download the scripts linked to this article and play them back yourself (although I request that you play them back with 10 users or fewer so my webmaster doesn't have a heartattack).

If you have been using TestStudio for some time, you likely already have a preference between block timers and timers, but I suggest that you take the time to experiment with the use of each. I had been a "timer-guy" for many years before writing this article, but while I was creating the examples I used, I found that block timers would have been quite valuable on some of my recent projects.

If any of you should decide to do extensive research on nesting timers, I request that you share your findings with me. As I mentioned, this has been an area that I have been researching for some time.

Summing It Up

While I took the opportunity in this article to introduce some concepts tangential to the central theme, the main point should still be clear "To collect and analyze user experience measurements, you must place your timers correctly". Improperly placed timers will yield misleading or inaccurate results while correctly placed timers will show you exactly what the current performance of a system or application is, and help you determine what, if anything, needs to be tuned for improved performance.

References

1) *End-to-End Application Response Times: Market Update* Gartner Inc. Oct 2000

Acknowledgments

- The original version of this article was written on commission for IBM Rational and can be found on the IBM DeveloperWorks web site

About the Author

Scott Barber is the CTO of PerfTestPlus (www.PerfTestPlus.com) and Co-Founder of the Workshop on Performance and Reliability (WOPR – www.performance-workshop.org). Scott's particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies, testing embedded systems, testing biometric identification and security systems, group facilitation and authoring instructional or educational materials. In recognition of his standing as a thought leading performance tester, Scott was invited to be a monthly columnist for Software Test and Performance Magazine in addition to his regular contributions to this and other top software testing print and on-line publications, is regularly invited to participate in industry advancing professional workshops and to present at a wide variety of software development and testing venues. His presentations are well received by industry and academic conferences, college classes, local user groups and individual corporations. Scott is active in his personal mission of improving the state of performance testing across the industry by collaborating with other industry authors, thought leaders and expert practitioners as well as volunteering his time to establish and grow industry organizations. His tireless dedication to the advancement of software testing in general and specifically performance testing is often referred to as a hobby in addition to a job due to the enjoyment he gains from his efforts.

About PerfTestPlus

PerfTestPlus was founded on the concept of making software testing industry expertise and thought-leadership available to organizations, large and small, who want to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our founders are dedicated to delivering expert level software-testing-related services in a manner that is both ethical and cost-effective. PerfTestPlus enables individual experts to deliver expert-level services to clients who value true expertise. Rather than trying to find individuals to fit some pre-determined expertise or service offering, PerfTestPlus builds its services around the expertise of its employees. What this means to you is that when you hire an analyst, trainer, mentor or consultant through PerfTestPlus, what you get is someone who is passionate about what you have hired them to do, someone who considers that task to be their specialty, someone who is willing to stake their personal reputation on the quality of their work - not just the reputation of a distant and "faceless" company.