# User Experience, Not Metrics

by:
R. Scott Barber


## Part 2: Modeling Individual User Delays

Visitors to your Web site think, read, and type at different speeds, and it's your job to figure out how to model and script those varying speeds as part of your testing process. There are a variety of ways to accomplish this using Rational TestStudio, and you'll learn some of them in this, the second article in the "User Experience, Not Metrics" series. The focus of this series, as explained in Part 1, is on correlating customer satisfaction with performance as experienced by external users. This article and the ones that follow are intended to provide all of the necessary theory about how to model real users for testing purposes as well as demonstrate how to either modify or create scripts that exemplify the theory, based on years of experience with performance testing.

Some industry experts claim that the variance in user delays isn't relevant to actual test results, and that delaying 20 seconds between each user activity provides the same results across a large user community. I would argue that several years ago that might have been true if your site was static HTML with single-server architecture, but it's simply no longer the case with today's multitiered, multifunctional Web sites. "The Science of Web-site Load Testing" by Alberto Savoia explains in a fair amount of detail why the 20-second theory isn't true and provides real-world examples and math to prove it.

This article is intended for all levels of TestStudio users but will be most helpful to Intermediate tool users and above. I present methods for scripting various mathematical distributions here but leave it up to you to determine which one most accurately depicts your particular Web site scenario. It goes without saying that the statistical accuracy of the ultimate user community model is important and unique to every Web site.

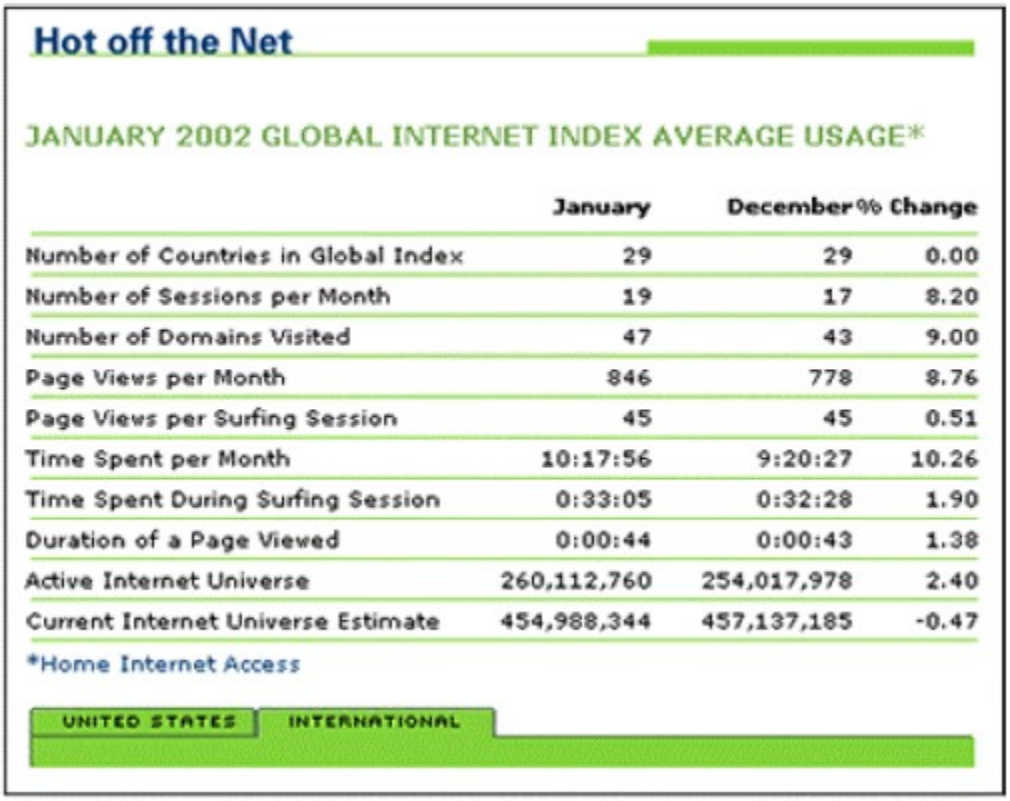
## Determining User Delays

So how long does it take a user to log in, navigate the home page, fill out a form, and so on? Several methods can be used to get some idea of the user delay times associated with user activities on your Web site. The best method, of course, is to use real data collected about your production site. This is rarely possible, though as testing generally occurs before the site is released to production. Because of this, there's sometimes a need to make educated guesses or approximations regarding activity on the site. The four most acceptable methods of determining this are as follows:

- When testing a Web site already in production, you can determine the actual values and distribution by extracting the average and

standard deviation for the viewing (or typing) time from the log file for each page. With this information, the user delay time can easily be determined for each page. Your production site may also have Web traffic monitoring software such as WebTrends or LiveStat that can be used to provide this type of information directly.

- In the absence of log files on a production site, or the time and resources for a detailed log analysis, you can leverage some of the metrics and statistics that have already been collected by companies such as Nielsen//NetRatings, Keynote, or MediaMetrix. These statistics provide data on average page-viewing times and user session duration based on an impersonal sample of users and Web sites. Although these numbers are not from your specific Web site, they can work quite well as first approximations. Figure 1 is an example of data from the NetRatings site. Figure 1: Data from NetRatings (average global Web usage, January 2002)



Hot off the Net

JANUARY 2002 GLOBAL INTERNET INDEX AVERAGE USAGE*

| | January | December | % Change |
|---|---|---|---|
| Number of Countries in Global Index | 29 | 29 | 0.00 |
| Number of Sessions per Month | 19 | 17 | 8.20 |
| Number of Domains Visited | 47 | 43 | 9.00 |
| Page Views per Month | 846 | 778 | 8.76 |
| Page Views per Surfing Session | 45 | 45 | 0.51 |
| Time Spent per Month | 10:17:56 | 9:20:27 | 10.26 |
| Time Spent During Surfing Session | 0:33:05 | 0:32:28 | 1.90 |
| Duration of a Page Viewed | 0:00:44 | 0:00:43 | 1.38 |
| Active Internet Universe | 260,112,760 | 254,017,978 | 2.40 |
| Current Internet Universe Estimate | 454,988,344 | 457,137,185 | -0.47 |

*Home Internet Access

UNITED STATES | INTERNATIONAL

**Figure 1: Sample Data from NetRatings**

As you can see, this is very high-level information that may or may not be useful in modeling your actual users. The most relevant piece of information in this chart is "Duration of a Page Viewed" which reports the average amount of time users spend viewing all types of Web pages. The remainder of the statistics are consolidated across a huge user community surfing to any combination of Web sites.

- If you have no log files and don't feel that the metrics from the companies above are representative, you can run simple in-house experiments using employees, customers, clients, friends, or family members to determine, for example, the page-viewing time differences between new and returning users. This method is also known as the "clipboard and stopwatch" method, for obvious reasons. I find this to be a highly effective method of data collection for

Web sites that have never been live, as well as validation of data collected using other methods.

- As a last resort, you can use your intuition, or best guess, to estimate the range for these delays. For realistic load tests, even this last approach is preferable to ignoring the concept of varying user delays. Creating a load test in which every user spends exactly the same amount of time on each page is simply not realistic and will generate misleading results.

## Understanding Delay Ranges and Distributions

You haven't finished laying the groundwork for modeling user delays just because you've now found how long one person spends on your pages, or what the range of time is. You must vary delay times by user, or the server will see a load that looks like the graph in Figure 2.

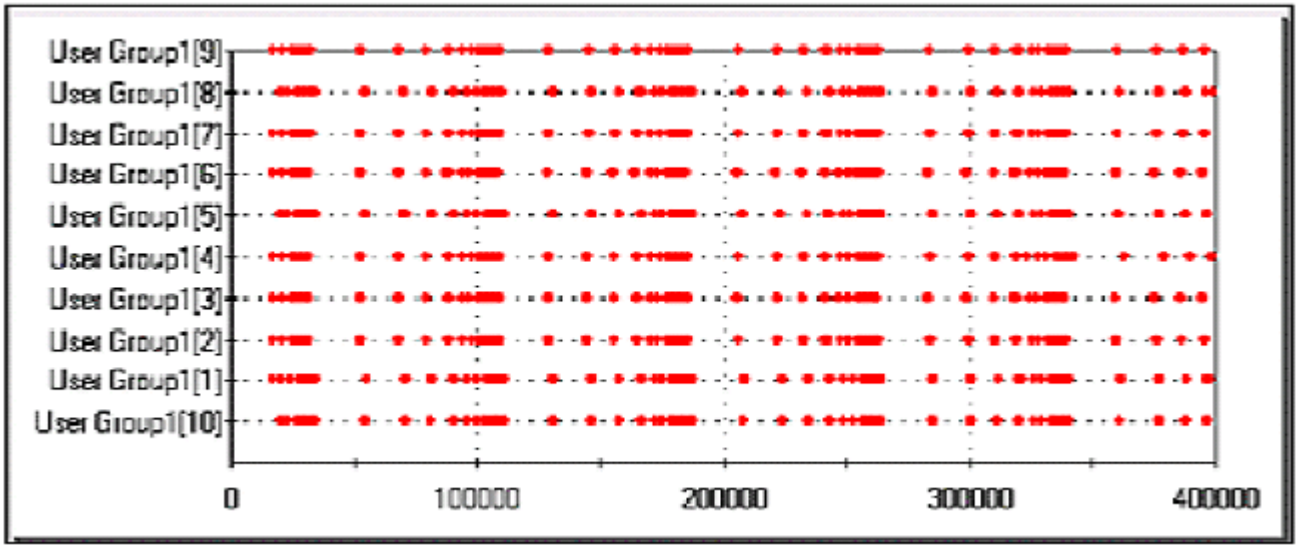



**Figure 2: Response graph showing banding**

In case you aren't familiar with response graphs, each red dot is a user activity (in this case, page requests), the horizontal axis shows time in seconds from the start of the test run, and individual virtual testers are listed on the vertical axis. This particular response graph is an example of "banding" or "striping." Banding should be avoided when doing load testing. The graph above is a good example of a stress test, not a load test. From the server's perspective, this test is the same as putting ten users in a room with ten identical computers with a coach yelling

"On your mark ... get set ... Click 'Home Page' ... wait ... wait ... Click 'Page1.'"

To see what I mean, take a ruler and hold it vertically on your screen and move it slowly across the graph from left to right. That's what the server sees: no dots, no dots, no dots, lots of dots, no dots. This is a very poor representation of actual user communities. Now look at the response graph in Figure 3.

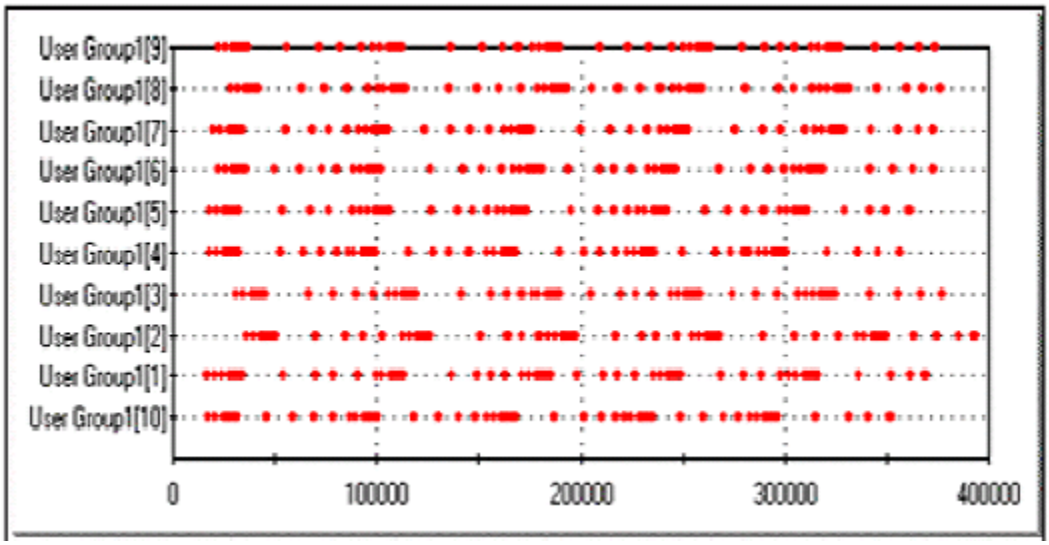



**Figure 3: Response graph without banding**

If you perform the same activity with the ruler, you'll see the dots are more evenly distributed this time. This is a significantly more accurate representation of actual users (although still not perfect). The only difference between these two test runs is that in the latter, bounded random delays instead of static delays were used.

To make the programmatic adjustment that I did between the two examples above, three pieces of information were required. In the first example, I simply used the average delay time as recorded in the script. In the second, I entered the minimum and maximum delay values as well as the distribution. There are numerous mathematical models for these types of distributions. I will focus on the two that are most commonly used: uniform distributions and normal distributions. I've also included a third distribution, known as negative exponential or negexp, for completeness. This distribution isn't used very often but is available. Advanced programmers may use this type of distribution in combination with other mathematical functions to simulate certain user patterns.

A uniform distribution between a minimum and a maximum value is the easiest to model. This distribution model simply selects random numbers that are evenly distributed between the upper and lower bounds. That means that it's no more likely that the number generated will be closer to the middle or the ends of the range. Figure 4 shows a uniform distribution of 1000 values generated between 0 and 25.

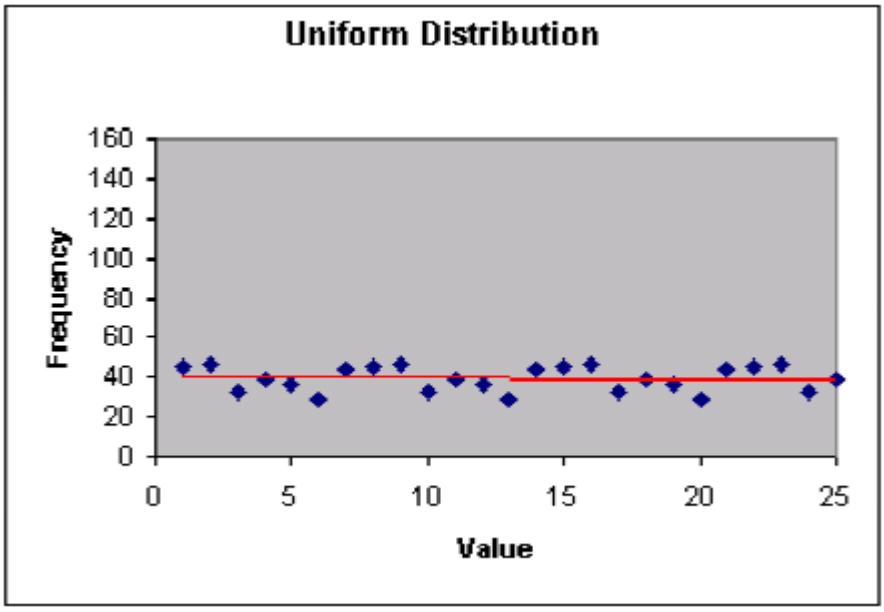



**Figure 4: A uniform distribution between a minimum and a maximum value**

A normal distribution, also known as a bell curve, is more difficult to model but is more accurate in almost all cases. This distribution model selects numbers randomly such that the frequency of selection is weighted toward the center, or average value. Figure 5 shows a normal distribution of 1000 values generated between 0 and 25 (that is, a mean of 12.5 and a standard deviation of 3.2). Normal distributions are generally considered to be the most accurate mathematical model of quantifiable measures of large cross sections of people when actual data is unavailable.

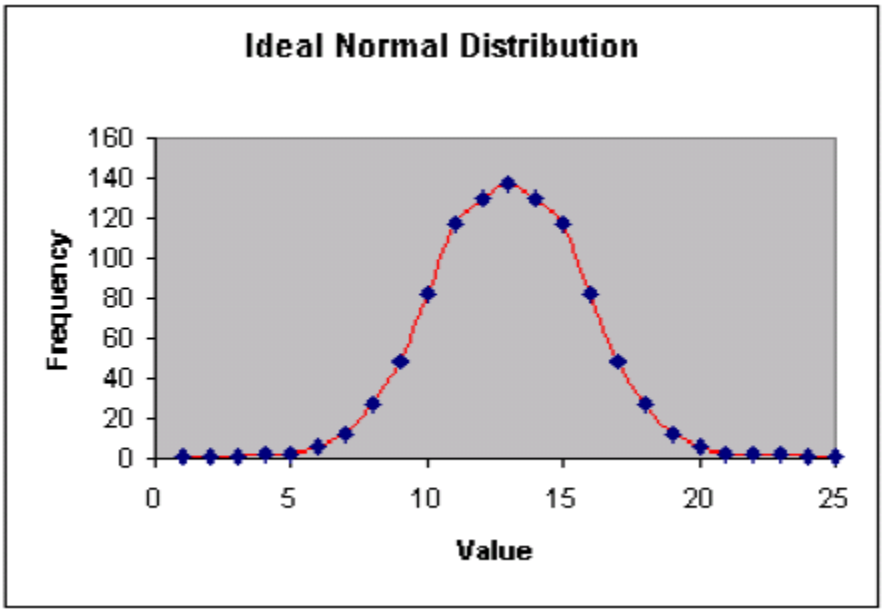



**Figure 5: An ideal normal distribution**

The negexp or negative exponential distribution creates a distribution that looks like the graph in Figure 6. This model skews the frequency of delay times strongly toward one end. I've very rarely found this distribution to be useful, but it's included here for completeness. Figure 6 shows a negexp distribution of 1000 values generated between 0 and 25.

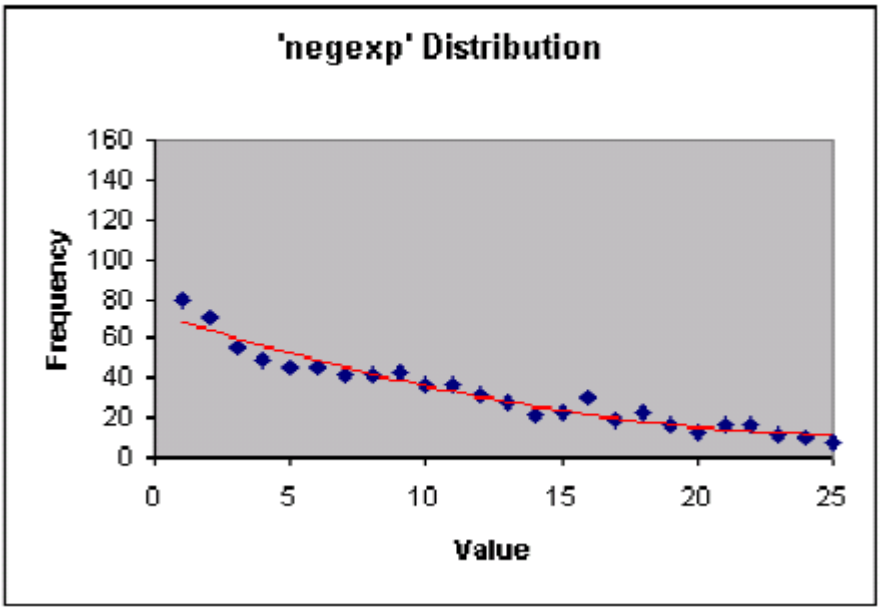



**Figure 6: A negative exponential distribution**

## Generating Time Delays in TestStudio

There are several ways to generate time delays using Rational TestStudio's VuC language. Rational documentation and training materials explain in detail how to accomplish this using the commands and parameters associated with the Think_avg command. Without an in-depth discussion, I'll summarize by saying that this method is valid only if Think_avg command aren't inside or adjacent to timers. I've found that it's programmatically easier to use Think_avg commands exclusively to represent client-side processing, but not for user modeling. Detailed discussions of timers and client-side processing time modeling are planned for future articles in this series.

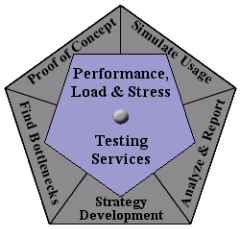
To script user delays that aren't included inside timers, the VuC delay command is used. Following are discussions of using the delay command to create static, uniform, negexp, and normal distributions of user delay times.

If it's determined that some activity takes all users exactly 8 seconds to perform (rare, but possible), the command will be:

delay(8000);

The number in parentheses, 8000, is the static delay value in milliseconds. The syntax for this command is delay(value); A static delay should only be used if a very good case can be made that it's an accurate model.

If you determine that it takes your users between 6 and 12 seconds to perform an activity on your Web




site and that a uniform distribution is the most accurate representation of actual users, then the command will be:

delay(uniform(6000, 12000));

In this example, 6000 is the minimum value and 12000 is the maximum value in milliseconds. The syntax for this command is delay(uniform(min_value, max_value)).

If you determine that it takes your users between 9 and 15 seconds to perform an activity on the Web site, but most users take closer to 9 seconds than 15 while none take less than 9 seconds to perform that activity, a negative exponential distribution will be most accurate. An example would be a site where a 9-second movie plays as an introduction and the Click Here to Enter button doesn't appear until the movie has completed playing. This command is:

delay(negexp(9000, 15000));

In this example, 9000 is the minimum value and 15000 is the maximum value in milliseconds. The syntax for this command is delay(negexp(min_value, max_value)).

There is a no VuC function to create a normal distribution using the delay command. To overcome this, we created the following function to be used in combination with the delay command to generate a normal distribution. The normdist function below should be placed into each script immediately following the #include command.

```
int func normdist(min, max, stdev) /* specifies input values
for normdist function */

/* min: Minimum value; max: Maximum value;
stdev: degree of deviation */
int min, max, stdev;
{

/* declare range, iterate and result as integers -VuC
does not support floating point math */
int range, iterate, result;

/* range of possible values is the difference between the
max and min values */
range = max -min;

/* this number of iterations ensures the proper shape of
the resulting curve */
iterate = range / stdev;

/* integers are not automatically initialized to 0
upon declaration */
result = 0;

/* compensation for integer vs. floating point math */
stdev += 1;

for (c = iterate; c != 0; c--) /* loop through iterations */
result += (uniform (1, 100) * stdev) / 100;
/* calculate and tally result */
```

```
return result + min; /* send final result back */
}
```

This function, when executed 1000 times with a minimum value of 0 milliseconds, a maximum value of 25000 milliseconds, and a standard deviation of 3200 milliseconds, generates the normal distribution in Figure 7. Note that these are the same parameters used to generate the ideal normal curve, just expressed in milliseconds rather than in seconds. As you can see, this graph is nearly identical to the ideal normal distribution graph shown in Figure 5.

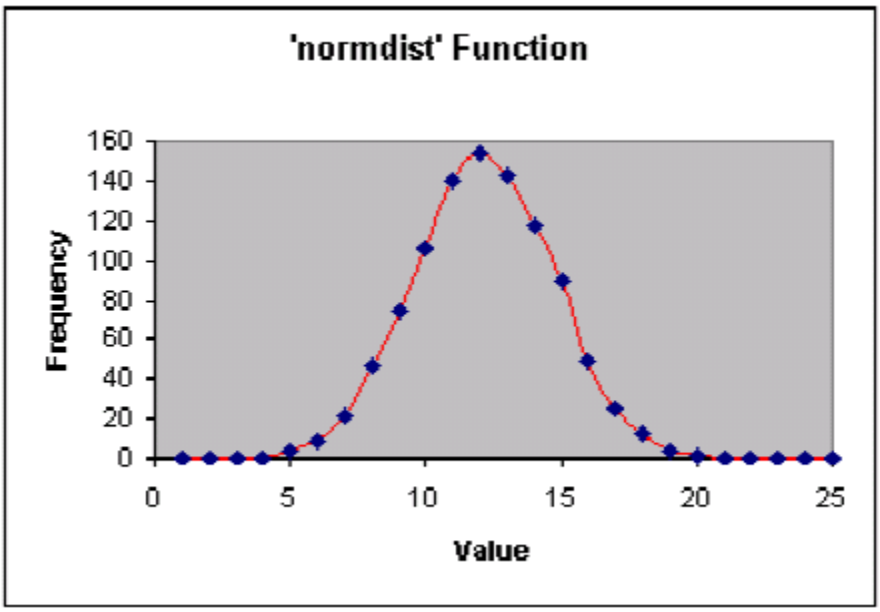



**Figure 7: Normal distribution generated with the normdist function**

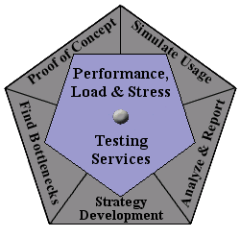
The normal distribution will be the most often used with the delay function. To model a user delay distributed normally between 10 and 35 seconds with a standard deviation of 3.2 seconds (as in the example above, only shifted to the right by 10 seconds), use this command:

delay(normdist(10000, 25000, 3200));

In this command, 10000 is the minimum value, 25000 is the maximum value, and 3200 is the standard deviation, all in milliseconds. The syntax for this command is delay(normdist(min_value, max_value, std_deviation)). Remember, this command will generate errors if the code for the normdist function isn't included in the script.

## Now You Try It

To demonstrate the ease and usefulness of these concepts, I suggest you follow along with the exercises below. I've developed these exercises against the noblestar.com production site. I've assumed that you already know how to record and play back a VU script and how to insert timers while recording.




## Determine User Delays and Distributions Exercise

Pick your favorite mostly static Web site (using a site that will change during the course of this exercise will simply frustrate you). Determine a short navigation path to follow. For example, on noblestar.com go to the Home Page, click About Us, then click Essentials, and finally click Heritage. First jot down on a note pad what you think the distribution and delay times would be for each of these pages. Now, go find some co-workers, give them the URL and the instructions on what pages to check out, and then time how long they stay on those pages. See how close they come to the times and distributions you chose. Then compare your guesses to the actual graphs and discussions of the delays and distributions for these pages from the log files.

## Model User Delays and Distributions in VuC Exercise

I've used all four methods of determining user delays and distributions at various times when testing this site. In all cases I come up with varying times that follow a (mostly) normal distribution curve. For the purpose of this example, we'll assume that isn't always the case so you can see how to generate delays using both the built-in C functions and the new normdist function that I introduced previously in the article.

Allow me to reiterate, the delays and distributions used in this example aren't good representations of actual site traffic for this particular site. I did this to create an exercise to demonstrate all of the topics discussed in this article.

First record a simple VU script against a static Web site. (I recorded against noblestar.com.) While you're recording, bracket each page load within a timer. (You could also use a timer block, but the unmodified code will be slightly different. I'll discuss timer blocks in more detail in Part 5.) Record three pages being loaded, ["Home Page"], ["Page1"], and ["Page2"]. See my original script with no modifications.

Once you've recorded this script, play it back with one virtual tester to ensure it plays back correctly before you make any modifications. Once you ensure that the script plays back properly, open the script for modification in Robot and find the first stop_time command. This section of your script should look something like this:

```
http_header_recv ["RDN_on_~233"] 304; /* Not Modified */

http_nrecv ["RDN_on_~234"] 100 %% ; /* 238 bytes -From Cache */
stop_time ["Home Page"];

start_time ["Page1"];
set Think_avg = 12342;

/* Keep-Alive request over connection www_noblestar_com */
http_request ["RDN_on_~235"]
```

This section of code stops the timer for the amount of time that it takes the Home Page to load, starts the timer for the amount of time it takes Page1 to load, then waits a little more than 12 seconds to select the link to start the download of Page1. As you can see, you do not want that twelve-second delay

included in your timer. For this example, you also don't want a static 12-second delay, but a uniformly distributed delay with a minimum of 6 seconds and a maximum of 18 seconds as the delay range. To do this, simply delete or comment out the set Think_avg command on the line following the start_time command and add delay(uniform(6000,18000)) between the stop_time and start_time commands. This section of code should now look like this:

```
http_header_recv ["RDN_on_~233"] 304; /* Not Modified */

http_nrecv ["RDN_on_~234"] 100 %% ; /* 238 bytes -From Cache */
stop_time ["Home Page"];

delay(uniform(6000,18000)); /* added to replace Think_avg below */

start_time ["Page1"];
/* set Think_avg = 12342; -replaced by delay above*/

/* Keep-Alive request over connection www_noblestar_com */
http_request ["RDN_on_~235"]
```

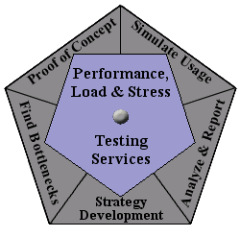
This section of code will now measure the actual time it takes to load the Home Page and Page1 as well as wait for the user to read the Home Page and select the link to navigate to Page1. The wait, in this case, will be a randomly selected, uniformly distributed amount of time between 6 and 18 seconds.

Now find the stop_time command for Page1. Your original code will look something like this:

```
http_header_recv ["RDN_on_~242"] 200; /* OK */

http_nrecv ["RDN_on_~243"] 100 %% ; /* 9997 bytes */
stop_time ["Page1"];

start_time ["Page2"];
set Think_avg = 8536;

set Server_connection = www_noblestar_com_1;

/* Keep-Alive request over connection www_noblestar_com_1 */
http_request ["RDN_on_~244"]
```

This section of code stops the timer for the amount of time it takes for Page1 to load, starts the timer for the amount of time it takes Page2 to load, then waits a little more than 8 seconds to select the link to start the download of Page2. Again, you don't want that 8-second delay included in your timer. For this example, you also don't want a static 8-second delay, but a normally distributed delay with a minimum value of 6 seconds, a maximum value of 14 seconds, and a standard deviation of 2 seconds. If you don't have a way to calculate the actual standard deviation of the delay range, a reasonably accurate estimate for standard deviation is 25% of, or .25 times, the desired range (maximum value minus minimum value) of the delay. To do this first, copy and paste the normdist function into your script immediately below the #include command to look like this:

```
#include

int func normdist(min, max, stdev)
```




```
int min, max, stdev; // min: Minimum value; max: Maximum value;
stdev: degree of deviation allowed
{

int range, iterate, result;

range = max ? min;
iterate = range / stdev;
result = 0;
stdev += 1;

for (c = iterate; c != 0; c--)
result += (uniform (1, 100) * stdev) / 100;

return result + min;
}

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

Then simply delete, or comment out, the set Think_avg command on the line following the start_time command and add delay(normdist(6000, 14000, 2000)) between the stop_time and start_time commands. This section of code should now look like this:

```
http_header_recv ["RDN_on_~242"] 200; /* OK */

http_nrecv ["RDN_on_~243"] 100 %% ; /* 9997 bytes */
stop_time ["Page1"];

delay(normdist(6000, 14000, 2000)); /* added to replace Think_avg below */

start_time ["Page2"];
/* set Think_avg = 8536; -replaced by delay above*/

set Server_connection = www_noblestar_com_1;

/* Keep-Alive request over connection www_noblestar_com_1 */
http_request ["RDN_on_~244"]
```

Again, remember that this code will generate errors unless the normdist function is included in your script. This section of code now measures the actual time it takes to load Page1 and Page2. The script also waits a randomly selected, normally distributed amount of time (between six and 14 seconds) for the user to read Page1 and select the link to navigate to Page2. You can examine the modified script in its entirety if you want.

## For Advanced Users: Double-Hump Normal Distribution

To apply this distribution, you should be experienced in using Rational TestStudio for performance engineering and have at least a fair understanding of the C programming language. Many of the Web sites I've tested have pages that have what I call a double-hump normal distribution across a production

user community. Think of a typical text-heavy Web page. The first time you go to that page, you'll probably want to read the text, but the next time you may simply click through that page on the way to a page deeper in the site. (Of course, that means the site navigation is not modeled properly, but that isn't the topic for this article; just because it shouldn't happen doesn't mean it isn't common.) When you look at the user delay times for a page like that, you come up with a graph like the one in Figure 8.

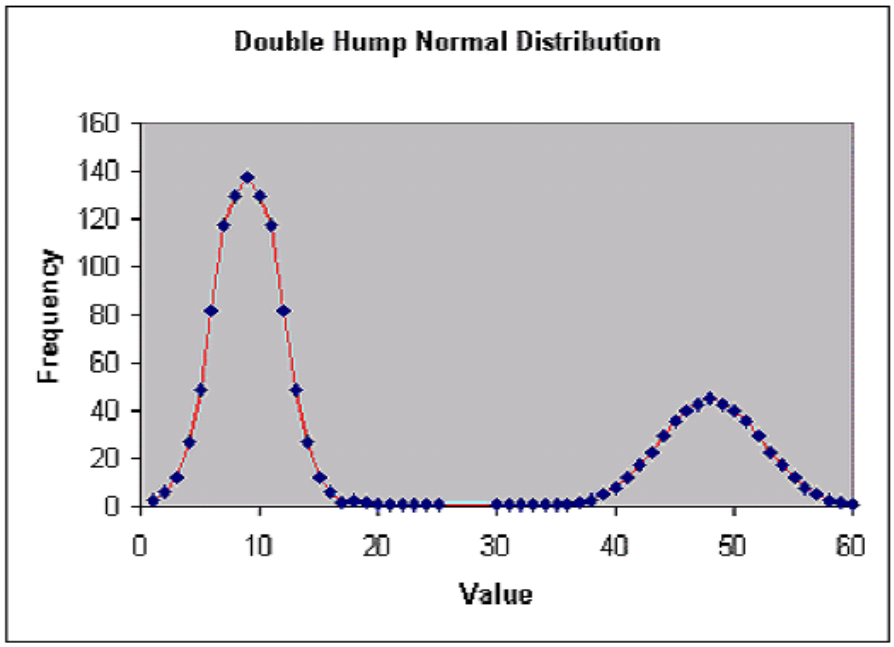



**Figure 8: A double-hump normal distribution representing a text-heavy Web page**

In this model, 60% of the users viewing this page spend about 8 seconds on the page scanning for the next link to click and the other 40% of the users actually read the whole page, which takes about 45 seconds. You can see that both humps are normal distributions with different minimum, maximum, and standard deviation values. The way to model this in your script is to generate a random number between 1 and 10 and then create conditional logic to execute the delay for the leftmost hump when the number is between 1 and 6 and the rightmost hump when the number is between 7 and 10.

```
stop_time ["Page1"];

y=uniform(1,10); /* randomly select a number
between 1 and 10 */

if (y<=6) /* if 1 through 6 */
{
delay(normdist(0,16000,2000));   /* Delay based on left side hump */
}

else /* if 7 through 10 */
{
delay(normdist(34000,64000,6000)); /* Delay based on right side hump */
}

start_time ["Page2"];
```

This code, executed 1000 times, generated actual values that populated the chart in Figure 9.

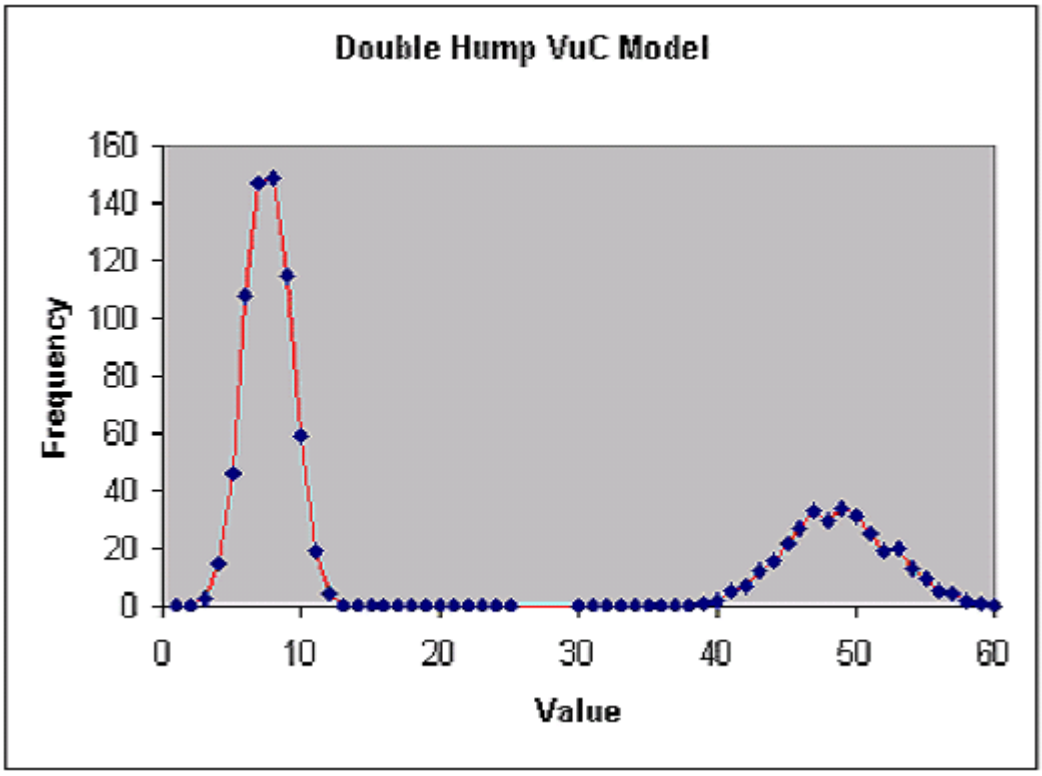



**Figure 9: The double-hump normal distribution generated by our script**

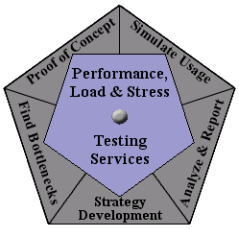
As you can see, using the same theories, it would be easy to vary the values and distribution types using this code, as well as changing the proportion of users who fall into each hump. By putting these principles together in varying combinations, it's possible to model, to a reasonable level of accuracy, almost any user delay distribution. For example, one might add an additional uniform distribution to simulate 15% of the user community varying between 10 seconds and 2 minutes to model outliers. The possibilities are endless.

## Summing It Up

The key point to take away from this article is simple: the more accurately users are modeled, the more reliable performance test results will be. The first component of accurate user modeling is modeling user delays. This article discussed both how to determine user delay times and how to use Rational TestStudio to script that information into your virtual user scripts. The next two articles in this series will discuss how to determine and model individual users' Web site usage patterns and then how to take those individual use models and consolidate them into a single user community model.

## References

- "The Science of Web-site Load Testing" by Alberto Savoia (Keynote Systems, Inc., 2000)




## 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.