Customers. **Performance**. Loyalty.

WHITEPAPER

Rapid Bottleneck Identification™

A Better Way to Load Test

You're getting ready to launch or upgrade a critical Web application. Quality is crucial, but time is short. How can you make the best use of your time in order to thoroughly test the application?

Rapid Bottleneck Identification — RBI — is a new methodology that allows QA professionals to very quickly uncover Web application limitations and determine what impact those limitations have on the end-user experience. Developed by experts at Empirix, RBI is used as the basis for all of Empirix's hosted load test engagements (a service called e-LoadExpert®), and has been proven to reduce testing cycles dramatically while allowing for more, and more thorough, testing.

Performance testing is part art, part science. In this article, we'll demonstrate how the two come together.

Rapid Bottleneck Identification (RBI) Performance Testing Process

The basis of the Rapid Bottleneck Identification (RBI) performance testing process grew out of the testing experience of Empirix's e-LoadExpert team and from the definition of performance testing that experience engendered.

Talk to any number of QA and Testing specialists about performance testing and you'll get as many definitions of the topic. A Google search produced a series of definitions which could be distilled to the following:

- Performance testing is "testing conducted to evaluate the compliance of a system or component with specified performance requirements."

This is a good starting point for a discussion, but overlooks two central points. The first is, in our experience, every application has at least one bottleneck and secondly, no application we've ever seen has ever scaled to meet its requirements, at least initially. These broad statements come from years of experience for hundreds of clients through thousands of performance tests.

Therefore, if prior to the start of testing we can make a reasonable assumption that bottlenecks will be uncovered and the system or application will not scale to meet its requirements, we need a new definition of performance testing to match our experience.

So, we change our definition to reflect the reality we have encountered:

- Performance testing is "testing conducted to isolate and identify the system and application issues (bottlenecks) that will keep the application from scaling to meet its performance requirements."

This change in definition reflects a philosophical and methodological change that drives everything we do in our hosted testing service, and led to the creation of the RBI methodology for load testing.

Before delving into the specifics of the RBI methodology we must first lay groundwork for the discussion through an examination of bottlenecks (and where they are found) and the distinction between throughput and concurrency testing.

Bottlenecks

What is a bottleneck? A bottleneck is any resource (hardware, software, bandwidth, etc.) that places defining limits on data flow or processing speed. An application is only as efficient as its least efficient element. In Web applications, bottlenecks directly affect performance and scalability by limiting the flow of data or restricting the number of connections/sessions.

As already stated, all systems have at least one bottleneck and most untested systems have more than one bottleneck, but they can only be identified and resolved one at a time.

Locating the Bottlenecks

In Web applications, bottlenecks can be found throughout the system infrastructure and application. Generally speaking, bottlenecks will either be at the system level (allocated bandwidth, firewalls, routers, server hardware, etc.), the Web server level (hit rate, CPU, etc.), application server level (page rate, CPU, memory, etc.), or the database server level (CPU, queuing, connections, etc.), (see Figure 1).

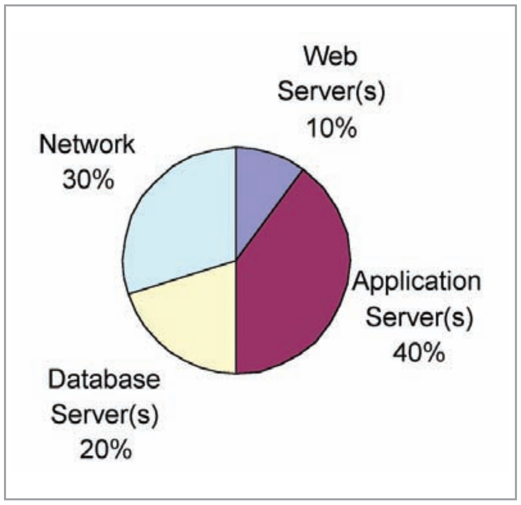

Figure 1: Location of bottlenecks throughout system infrastructure.

In general, the more complex the testing scenario, the more potential bottlenecks can come into play, and the harder it will be to isolate those bottlenecks as more of the tiers are involved.

Take the example graphed in Figure 2. The graph shows a test for a standard e-commerce application that bottlenecked at approximately 2000 concurrent users, with response times climbing as additional users were added to the system.

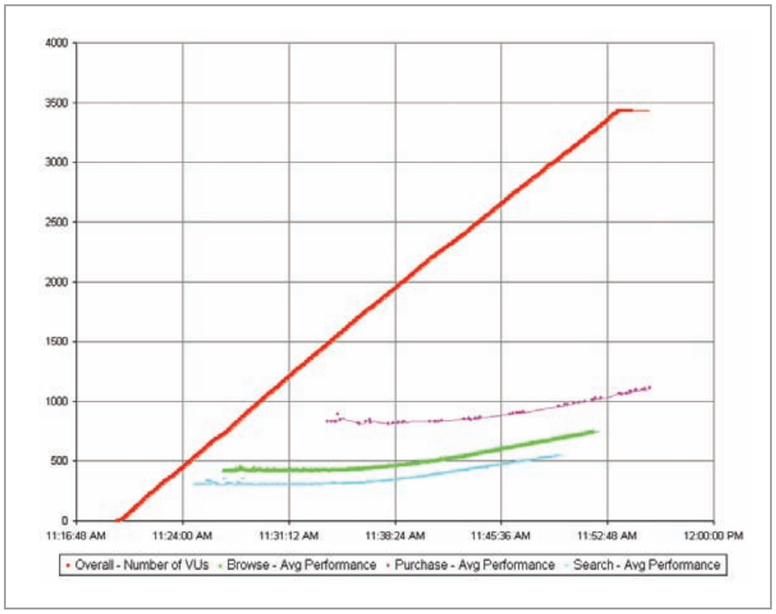

Figure 2: Graph of standard e-commerce application test.

In this case, there were only three transactions being tested, but still these transactions interacted with all tiers of the system and application. As with the usage scenarios of typical e-commerce applications, users were browsing, searching, adding items to cart and completing purchases. The bottlenecks in this test could have been caused by any one of those functions, or perhaps it was a system issue. The bottom line is the more variables in a test, the harder it is to determine what is causing the bottleneck.

So, if problems can be encountered in any tier of the architecture, and the likelihood of finding problems in any one tier is not substantially higher than in any other, is there another place we can look for guidance?

Throughput Versus Concurrency

In Empirix's experience, 80 percent of all system and application issues come down to limitations in throughput capacity as apposed to concurrency, (see, Figure 3).

Throughput. Throughput is the measure of the flow of data that a system can support and is measured in hits per second, pages per second, Mbps (megabits per second).

Concurrency. Concurrency is the measure of independent users that a system can support. Concurrent users in a Web application are defined as users with a connection to a system (and often a session on an application) who are performing some action or sequence of actions at a regular interval.

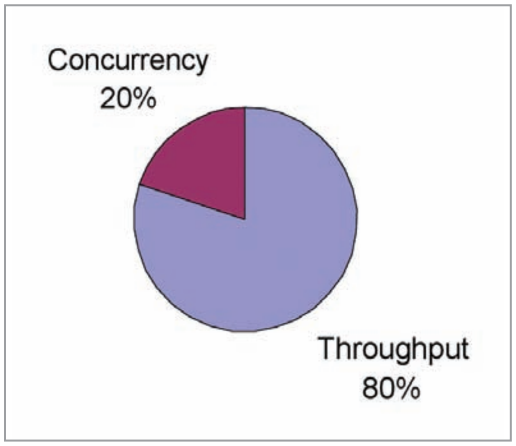

Figure 3: Bottlenecks in Throughput versus Concurrency diagram.

Testing for Throughput

Testing for throughput involves minimizing the number of user connections to a system while maximizing the amount of work that is being performed by those user connections. In this way the system and application under test are pushed to their maximum capacity.

For throughput testing at the system level, basic files should be added to the Web and application servers. Typically a large image is used for bandwidth tests, a small text file or image is used for hit-rate tests and a very simple sample application page (a 'Hello World' page for instance) is used for page-rate testing. Should the system be unable to meet the requirements of the application there is no need to continue testing until the system itself has been improved either through tuning of settings, an increase in hardware capacity or an increase in allocated bandwidth.

Throughput testing of the application involves hitting key pages and user transactions with limited delay between requests to find the page-per-second capacity limit of the various functional components. The pages or transactions with the poorest page throughput are those in need of the most tuning.

Testing for Concurrency

On the system and application level, concurrency is limited by sessions and socket connections. Code flaws can also limit concurrency, as can incorrect server configuration settings.

Concurrency tests are run by ramping a number of users running with realistic page-delay times, ensuring that the ramp up is slow enough to gather useful data throughout the testing. As with throughput testing it is important to test the key pages and user transactions.

Another key difference in throughput and concurrency testing will be discussed later.

How are Throughput and Concurrency Tests Different?

Is the load generated from a 100 Virtual User load test, with one-second think times, equivalent to a 1000 Virtual User load test with 10-second think times? As one can see from the table below, the answer is No.

Table 1: 100 Virtual User Load Test Versus 1,000 Virtual User Load Test

	Throughput	Concurrency
100 Users — 1 Second Delay	$\text{Pages/Second} = \left(100\,VUs \times \frac{1\ page}{VU}\right) \div 1\ \text{Second} = 100$	100 Connections per Session
1,000 Users — 10 Second Delay	$\text{Pages/Second} = \left(1{,}000\,VUs \times \frac{1\ page}{VU}\right) \div 10\ \text{Seconds} = 1{,}000$	1,000 Connections per Session

In terms of throughput the two tests are identical, however in terms of concurrency they are vastly different. The graphs below demonstrate the difference between the two and the importance of testing for both.

In the first case, a throughput test, the application bottlenecked at 50 pages per second (see Figure 4). In the second case however, a concurrency test of the same transactions, the application bottlenecked at 25 pages per second (see Figure 5). The only differences between these two tests were the number of users on the system and the length of time those users stayed on the pages. In the throughput test with fewer users and shorter page view delays the application had more throughput capacity; the second test shows the application was limited in its concurrency. Had the tests only been run for throughput the connection issue would have gone undiscovered until real users were on the system.

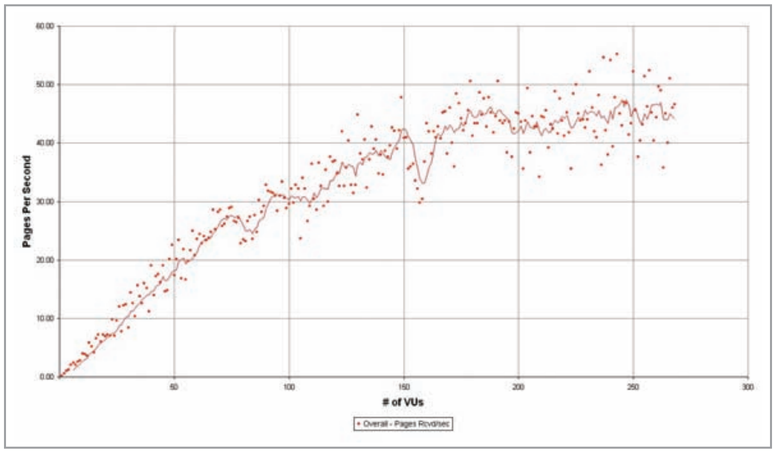

Figure 4: Sample Customer Test 1 — Throughput.

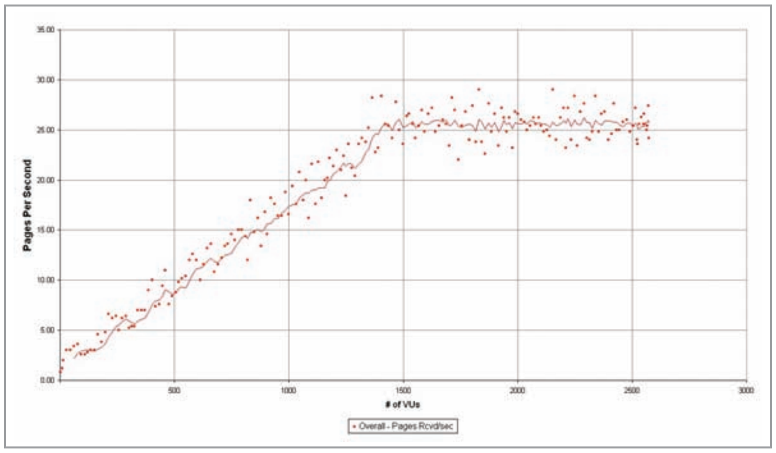

Figure 5: Sample Customer Test 2 — Concurrency.

Testing the RBI Way

Traditionally, performance testers have focused on concurrent users as the key metric for application scalability. However, if 80 percent of the application and system level issues are found in throughput tests, then a new approach is needed. The Empirix RBI testing methodology represents a shift away from a focus on concurrent users, though as the previous example shows, concurrency is still an important part of testing.

Starting with our basic assumptions, the RBI method comes about naturally.

1. All Web applications have bottlenecks.
2. These bottlenecks can only be uncovered one a time.
3. Focus should be placed on where the bottlenecks are most likely to be found — in the throughput.

At the highest level the RBI methodology can be summed up as follows:

- RBI testing isolates bottlenecks quickly by starting with the simplest possible tests, working to build in complexity, so that when an issue is uncovered all of the simpler causes have been ruled out. It's fast because we focus on the bottlenecks first, and where they are most likely to be found — in the throughput.

RBI Testing Is

Fast

Focusing on Throughput Testing Take Less Time (Minutes vs. Hours)

Focusing initially on throughput testing saves time, but how much time? Take an example of a system expected to handle 5,000 concurrent users, with users spending an average of 45 seconds on each page. If the application has a bottleneck that will limit its scalability to 25 pages per second, a concurrency test would have found this bottleneck at approximately 1125 users (25 pages per second x 45 seconds per page).

In the interest of not biasing the data, a typical concurrency test ramp up should proceed slowly. Empirix recommends ramping one user every five seconds. In this example then, the bottleneck would have been encountered 5625 seconds or 94 minutes into the test (1125 users x 5 seconds). However, to validate the bottleneck, the test would have to continue beyond that point to prove that the throughput was not climbing as the users were added.

A throughput test could have found this problem in less than 60 seconds.

Simple

Much Testing Can be Done Without Even Looking at the Application

Very often performance testing begins with overly complex scenarios exercising too many components, making it easy for bottlenecks to hide. The RBI methodology begins with system-level testing which can be carried out before the application is even deployed.

Modular & Iterative

Focus on One Piece of the Puzzle Until it Works, then Proceed

Continuing the idea of simplicity, the RBI methodology involves testing the simplest possible test case, and then building in complexity. If the simplest test case works, then the next level of complexity fails, it is easy to determine where the bottleneck is — in that newly added complexity.

Has 'Knowledge' Built in

When Problems Occur Each Previous Step has Already Been Ruled Out

When testing with the RBI methodology, the knowledge needed for identifying the location of the bottleneck is built in. When a bottleneck is uncovered, the modular and iterative nature of the methodology means that all of the previously tested components have been ruled out. For instance, if hitting the homepage shows no bottlenecks, but hitting the homepage plus executing a search shows very poor performance than the cause of the bottleneck is in the search functionality.

By uncovering bottlenecks in this tiered approach you can isolate issues in components of which you have limited knowledge.

Based on Years of Testing Experience

Empirix's e-LoadExpert consultants have years of performance testing experience on all types of platforms. The RBI methodology is a direct result of that experience, and a desire to learn something from every test run. While every performance test run will teach you something, the quality of that information is a direct reflection of how well the test is designed. Using this methodology is the easiest way to leverage the uniquely broad experience of our e-LoadExpert team.

Putting it All Together

Bottleneck Knowledge + Testing Methodology = A Highly Scalable Web Application

Once again, the RBI Methodology involves stepping into the system one level at a time, starting with the simplest possible test case and building in complexity. If a test passes then testing moves on; if not, attempts should be made to fix the problem followed by retesting.

If a step fails a requirement there is no sense in proceeding because adding in the next layer of complexity can only make performance worse. Fix the problem first, and then move on.

RBI Testing For Common System Level Bottlenecks

Any performance testing engagement should begin with an assessment of the basic network infrastructure supporting the application. If this basic system cannot support the anticipated user load on a system than even infinitely scalable application code will bottleneck. Basic system level tests should be run to validate bandwidth, hit rate and connections. Additionally, simple test application pages should be exercised as well, 'hello world' type pages for instance.

The Application

After validating that the system infrastructure meets the most basic needs of the end users, its time to turn to the application itself. Once again, start with the simplest possible test case and work in complexity.

If testing has come this far without uncovering system-level issues (or those issues have been resolved), any remaining problems are caused by the application itself. For example, if a test application page achieved 100 pages per second, and the homepage bottlenecks at only 10 pages per second, the problem is in the overhead required to display the homepage.

At this point, the test application page test provides two valuable pieces of information. First, the system itself is not the bottleneck, leaving only the code on that homepage as the culprit, and secondly, it teaches us how much tuning the homepage could improve performance. The difference between the performance of the test application page and any particular page under test is the maximum that performance could be increased by tuning.

Since any real-world application page is likely going to require more processing power than a 'hello world' test page, it is reasonable to expect some drop off in performance. However, the greater that drop off, the greater the need for (and potential gain from) tuning. If the drop off between the test page and an actual application page is not substantial, and the performance is still insufficient to meet the needs of the application user base, then more hardware capacity is needed (more machines or more powerful machines).

Up to this point no mention has been made of page-response times. While response times are a key metric of overall performance it is important to remember that response times will be the same for one user as they are for 1,000 or 100,000 users unless a bottleneck is encountered. So in this methodology, response times are only useful as an indicator that a bottleneck has been reached or as failure criteria, with poorly performing pages (either errors or high response times) being those most in need of code optimization.

RBI Testing For Application Bottlenecks

As with the system level testing, the RBI methodology begins application testing with the simplest possible test case, and then builds in complexity. In a typical e-commerce application that would involve testing the homepage first, then adding in pages and business functions until complete real world transactions are being tested first individually and then in complex scenario usage patterns.

As steps are added, degradation in response times or page throughput will be caused by the newly added step, making it easier to isolate what code needs to be looked at.

Once each of the business functions and transactions has been tested and optimized (as necessary) the transactions can be put together into complete scenario concurrency tests. These concurrency tests must be designed with focus on two components. One, the concurrency test must accurately reflect what real users do on the site (for example 50 percent just browse, 35 percent search, 10 percent register and login, five percent add to cart and make purchases), and two, the virtual users must execute the steps of those transaction using the same pacing that real world visitors do.

This data can be gathered from a Web logging tool, with a focus on session length, pages viewed in a session to determine the user pacing, and percentages of pages hit to determine what business functions real users are exercising.

Once the test has been designed from this real-world data (or educated assumptions on site usage for a newly deploying application) the test must be executed in a way that valuable information is gained at various user load levels. If the site is expected to handle 1,000 concurrent users it is important to ramp slowly enough (perhaps adding one user only every few seconds) so it will be easy to determine overall performance at any point of the load test.

Summary

We've demonstrated how the RBI methodology can be successfully applied for load testing Web applications. Much of this process can and should be automated using an automated testing tool or a managed service. The choice of tool vs. service should be based on the frequency of testing, the number of virtual users required, and the level of experience of your in-house staff. It's essentially a rent vs. buy decision. Many organizations purchase automated tools for their everyday testing needs, and then supplement them with a managed service for stress tests or other infrequent tests. This allows them to build in-house expertise without overspending on extra resources that they only need a few times a year. RBI can have a dramatic impact on the productivity of your testing team, and on the quality of your applications.

The e-LoadExpert team is led by Senior Consultant, Colin Mason. Colin is one of the experts and developers of the RBI method. He can be reached at cmason@empirix.com.

Corporate HQ | 20 Crosby Drive | Bedford, MA 01730 | USA | tel: +1 781.266.3200 | fax: +1 781.266.3201 | www.empirix.com
European HQ | Building A, Trinity Court, Wokingham Road | Bracknell, Berkshire RG42 1PL | United Kingdom | tel: +44 (0)1344 668080 | fax: +44 (0)1344 668172

empirix Customers. Performance. Loyalty.