Testing

The Top 13 Mistakes in Load Testing Applications

by Mark D. Anderson

A lot more people talk about it than actually do it. People define it differently. No amount of reading can compare to personal experience. Size does matter.

QUICK LOOK

- Common load testing errors and consequences
- Tips on determining the workload, employing virtual "super users," and knowing when to stop testing

We are speaking, of course, of load testing.

This article outlines thirteen common load testing mistakes that I have encountered in my work with clients. The focus of this article is Web *applications*—not Web sites that are just static files and images, but sites that are user interfaces to backend applications (such as a stock trading or credit card authorization system). We also do not discuss "native" client/server systems, though many of the same observations apply.

This list of mistakes is in no particular order (and is certainly not in order of importance).

ONE
Confusing Load Testing with Something Else

Load testing is about verifying the performance of a system under a simulated multi-user workload.

Load testing is not functional testing. In fact, those two are usually far distant from each other in many di-

mensions: the goals are different; the necessary skills are different; the test scripts are usually different (and far fewer, in the case of load testing); the appropriate tool technologies are usually different; and the appropriate stages in the product life cycle are usually different.

Load testing is not about verifying single-user performance. Obviously, end-to-end response time is different when a system is used by just a single user than when used by many—that is why load testing is necessary, after all. But as shown in Figure 1, the slowdown per component can vary considerably as the users are increased. The slowest component in the single-user case may be completely different from the component causing the slowdown when there are multiple users. Furthermore, single-user performance can be dominated by delays in processing which occur on the client machine—those might be of concern, but since there is a client machine for every user, those client-side contributions are irrelevant in scalability.

Load testing is not primarily about finding multi-user errors. When multiple users hit a server concurrently, there may be no performance problem, but the server might do the wrong thing—for example, it might sell multiple users the one remaining inventory item, or it might just crash. Whether you want to do so or not, you'll end up doing some such testing in at least an ad hoc fashion as a side effect of the load testing. But to do "real," full-scale, multi-user testing typically requires an approach more akin to single-user functional testing, including a way to reliably reproduce race condition scenarios.

TWO

Confusing the Web Server with the Web Application

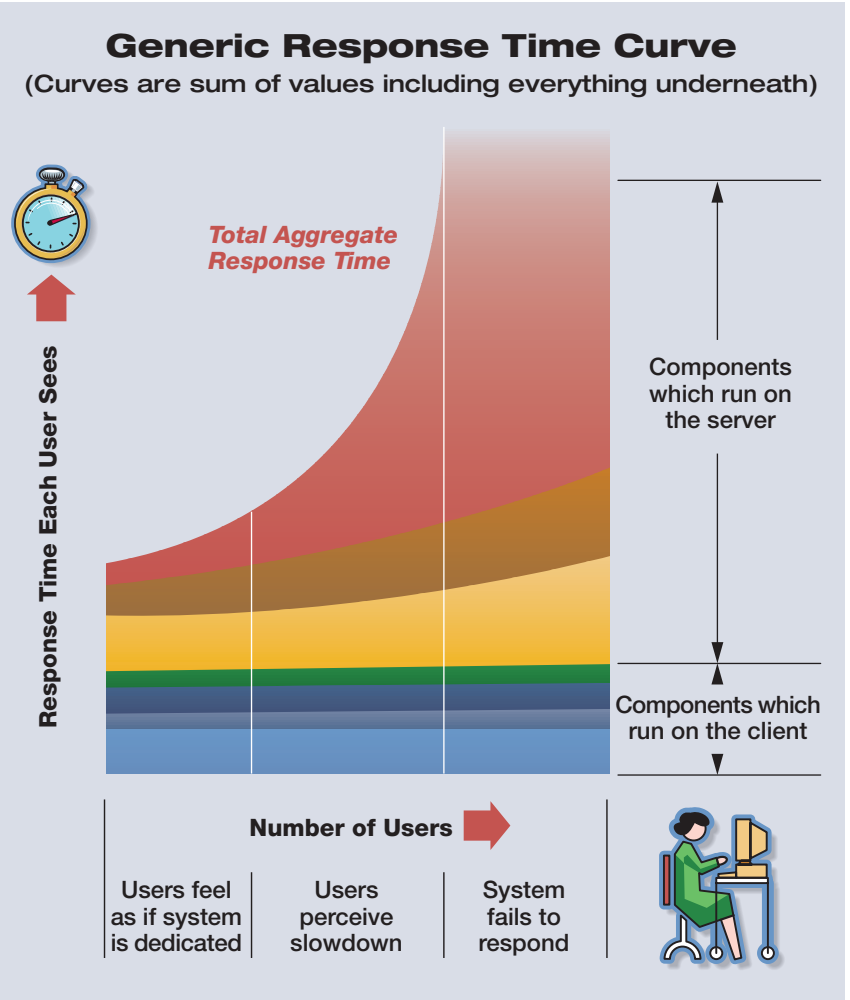

FIGURE 1 This figure shows an idealized plot of the response time of a system as the number of users is increased. When there are only a small number of users, they each feel as if they have dedicated use of the system. Beyond a certain level, the system begins to slow down because of contention over some shared resource. Then, eventually the system just breaks, and no response comes back at all.

CHART BY ANNIE BISSETT

We started this article saying that we want to test a Web *application*, not a Web server. The Web server receives http requests and quickly translates those into other kinds of requests to downstream applications. The amount of time spent in the Web server itself should be negligible: well under a millisecond per request. All the time is really taken up by the downstream application(s) behind it.

You shouldn't completely bypass the Web server in your testing (in fact most systems aren't designed to have a clean layer underneath that would make that possible anyway). And you do need to make sure the Web server layer can handle the requisite number of concurrent connections; this number is increased if the size of responses is high, the clients have low-bandwidth network connections, the client browsers implement http "keep-alive" but not "pipelining," or the backend is slow.

But in the end, you are testing the application behind the curtain, not the curtain. This means that while load testing a Web application, you usually needn't worry about such things as emulating http protocol variation, sending http requests for static images or pages, or doing anything else that is absorbed by the Web server and not seen downstream. You still might want to test those other aspects; just separate that exercise from your application load testing. (Note that you particularly want to verify the Web server layer if it is running https,

since connection establishment is so expensive in that case—and someone has probably done something dim like configured the server to encrypt all the gifs along with the text.)

You also shouldn't get distracted by Web server benchmarking tools such as MindCraft's WebStone, SPEC's specWeb96, or ZDNet's WebBench. These are for testing Web servers, not testing Web applications. They come with a supplied set of standardized static files to put behind your Web server for testing purposes. Such a load is suitable only for a static Web site—not for a typical "weblication," in which there may be no static html files on a disk at all, and all the html sent back is dynamically generated by a backend server.

The load from a Web benchmark has nothing to do with your application. It is possible to alter the URLs used by the tool, in order to direct them toward your real application (particularly with WebStone, since it is open source). But there are other tools that were actually built for application load testing, some of which are discussed later in this article.

THREE
Starting Late

Load testing can and should be done long before a system has a stable or complete user interface. One reason that people often schedule load testing as a final step in a test or development plan is the confusion linking load testing with functional testing. Your load testing tools are only reliant on there being a server which can handle http requests for the most important workflows. There needn't be a nice browser user interface, or even the kind of interface most Web sites have.

Perhaps another reason that people schedule load testing to happen last is that conventional wisdom says that one should get an application working first, and *then* do performance tuning. That advice is dubious even when building single-user applications, but it is likely to result in a disaster if applied to developing a multi-user application. Often the problems discovered by load testing can reveal deep architectural problems, which might be addressed if discovered early enough.

Of course, you also want to perform a load test as a final pre-launch qualification, for either an initial release or any major new revision.

You'll also want to do some load tests for capacity planning purposes (comparing different hardware deployment alternatives) mid-way through the project.

As they say: Test early, test often.

FOUR
Thinking It Can Be Done by One Person

Regardless of whether you *should* load test, it still might not be possible for you. You and your organization might lack:

- The skills necessary—if the people saddled with load testing can't program, and can't run an OS performance monitor, you've got a problem.
- Developer involvement—when the system croaks, only the developer responsible (or the database administrator, in the case of a database server) will be able to debug it, let alone be able to fix it.
- Organizational commitment to time, labor, and hardware—while load testing tools don't have to cost a lot, it still takes time and discipline to perform a load test that will mean anything.

You'll need to know who is responsible for every part of the system that's downstream from your load driver. They'll need to supply you with monitoring tools, or they'll have to be there in person while you are doing the load test.

FIVE
Picking the Wrong Tool

The state of the art in load testing tools leaves much to be desired. To choose among the tools available, it is necessary to locate your load testing requirements along these two dimensions:

- **Complexity:** How many different user workflows should you test? How many separate pages/steps are there in each workflow? (I use the term "workflow" here to mean what others might mean by "scenario," "use case," or "transaction.")

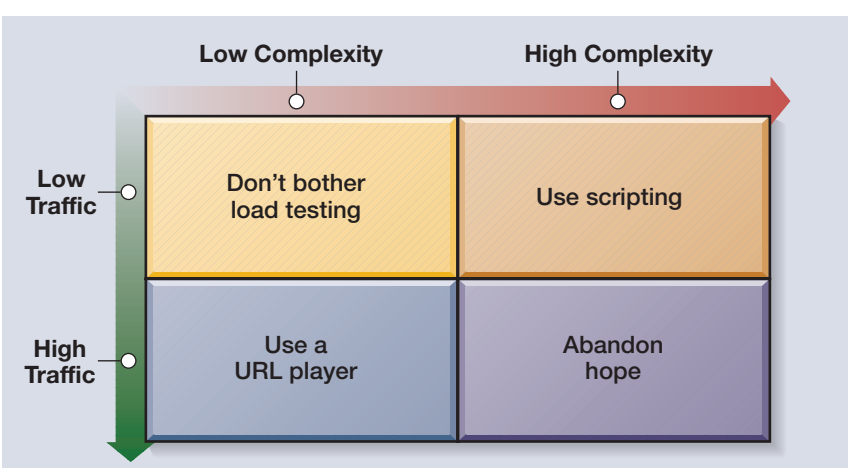

FIGURE 2 A tool selection strategy grid.

CREDIT CREDIT CREDIT

CHART BY ANNIE BISSETT

- **Traffic:** What is the total rate of requests?

Your strategy in tool selection should be something like what's shown in Figure 2 (previous page).

High Complexity, Low Traffic

An example of this case would be an online purchasing system—the request rate is not that high (you probably won't be getting a purchase every second!), yet the workflow is rather complex (checking credit card authorization, etc.).

If you are in this quadrant, then you can perform your load testing by scripting. First, hand-write your workflows in your favorite scripting language, presumably *Python* or *Perl*. Those languages both come with powerful libraries that can emulate *anything* that a browser can do (that is why they are popular for writing Web spidering and link-checking utilities). They can take care of all the various technical issues in browser emulation, such as frames, cookies, http redirects, and so on.

It is probably worthwhile to clarify that these scripts are not driving a browser. The *Perl* (or *Python*, or whatever) libraries are http clients themselves (often called "user agents"). It makes no difference to the server *what* is sending it http requests; and using an http library is considerably simpler, more efficient, and more robust than sending mouse and keyboard clicks to a browser. The application may have *JavaScript* behaviors and other client-side impedimenta, which should undergo some functional testing, but as long as the server doesn't see them, they have no role in load testing.

After your scripts are written, the next step is to hook them up to a parallel-testing engine such as:

- **http://stein.cshl.org/~lstein/torture**
- **http://perl.apache.org/guide/performance.html#Tuning_with_crashme_script**
- **http://web.stonehenge.com/merlyn/WebTechniques/col28.html**

(Note: The tools above all use *Perl*. They also all happen to rely on Unix-specific features not yet available in perl-win32. That will likely change soon…perhaps by the time you read this.)

Using this scripting approach to load testing means that you will incur the overhead of a script interpreter in your load generator, which means that it won't scale to very high loads. However, you can get to high tens of requests per second with this approach from even a good laptop.

Note that if your Web application, in part, uses non-http protocols such as *Java RMI* or *ActiveX* data-bound controls, scripting is still a viable solution. In fact, you could even still use *Perl* or *Python*, both of which can exercise *Java* objects or (on win32) invoke *ActiveX* objects. Or you could use the Windows Scripting Host, depending on your religious affiliation.

Low Complexity, High Traffic

An example of the "low complexity, high traffic" case would be a search engine—the request rate might be quite high, but there is little workflow to speak of (at most, some users might request a second page of results). Another example would be an ad server.

If you are in this quadrant, then you can use one of the "URL player" tools listed here. These tools are not capable of expressing complex workflows, but because they don't incur an interpreter overhead, they can generate far more load from the same computer resources than those that do.

Some example tools in the "URL player" category are:

- ApacheBench, in the apache distribution as "bin/ab" or at **http://webperf.zeus.co.uk/intro.html**
- Acme Software http_load, at **http://www.acme.com/software/http_load**
- HP Labs httperf, at **ftp://ftp.hpl.hp.com/pub/httperf**
- Binary Evolution VeloMeter, at **http://www.binaryevolution.com/velometer/velometer.vet**
- Microsoft WCAT, at **http://msdn.microsoft.com/workshop/server/toolbox/wcat.asp**

These tools each have their pros and cons (which these margins are too small to hold), but they are all free and downloadable, so you can figure that out yourself. And all but Microsoft's provide source code, so you can fix anything you don't like.

High Complexity, High Traffic

If your application is both high in complexity and high in traffic, then you are in trouble.

Of course, you can probably get some useful results by first doing some testing in the other quadrants, using a mixture of tools. Also, it is possible to perform testing in this quadrant, but it will cost you: you'll probably have to hire a consultant, and/or do some custom *C* programming, and/or buy a bunch of load generator machines.

SIX

Focusing on Script Writing Instead of Fault Isolation

Another inappropriate carry-over from functional testing is a fixation on the act of script writing. Writing the test scripts is the easy part, once you've gotten the hang of it. The *hard* part is finding the cause and repairing it when the system starts coughing up blood.

For example, I crashed the user registration system on this magazine's sister site, **STQE.net**, using a single ApacheBench command line. This was sufficient to cause the site to spew the following error message:

Microsoft OLE DB Provider for ODBC Drivers error '80040e4d'

[Microsoft][ODBC SQL Server Driver][SQL Server]Unable to connect.

The maximum number of '240' configured user connections are already connected.

System Administrator can configure to a higher value with sp_configure./memberprofile.asp, line183

Fifteen minutes was spent writing and executing the load test. How long would it take to repair the resulting crash? It can take hours, even days, just to find out what is wrong with complex systems, let alone fix the problems.

Monitoring

Because you'll be spending most of your time in analysis, it is every bit as important to invest in setting up and understanding system monitors as it is to work on tests. You'll need lots of monitors running while you run the test:

- OS-level monitors on your load driver, to make sure it isn't bottlenecking
- OS-level monitors on your server
- Custom component-specific monitors that the developers write
- Monitors provided with third-party server components, such as database server monitors

And of course, the load driver itself will be checking for errors coming back from the server. Note that your operations group should have some basic monitors available, since—after all—they *will* be monitoring the server when it goes live…right?

For OS-level monitoring on NT, perfmon is excellent. On Solaris, Sun supplies the "SE Toolkit" (which is also now commercialized as Symon). Linux lags a bit, but xosview and KTop provide some help. And of course, all Unixes have a variety of command line tools like vmstat and netstat for monitoring particular aspects of the OS.

The Problem Could Be Anywhere

The bottleneck could be at any layer in your system. For example, in the hypothetical system shown in Figure 3 (previous page), a bottleneck might be found at:

The Web Server It might refuse connections beyond a certain level of concurrency, because someone forgot to apply the appropriate OS patches or kernel configurations.

The Application Server It might crash because it isn't thread safe, or might produce erroneous results because it doesn't generate differently-named temp files for concurrent users, and so on.

The Database Server It might slow way down when you mix in some writes with your reads, because of lock resolution or some other problem.

Note that *none* of the above examples would be found in single-user testing. Note also that the solution to a problem is not always located where the problem is found. The database might be the bottleneck, but after a certain point

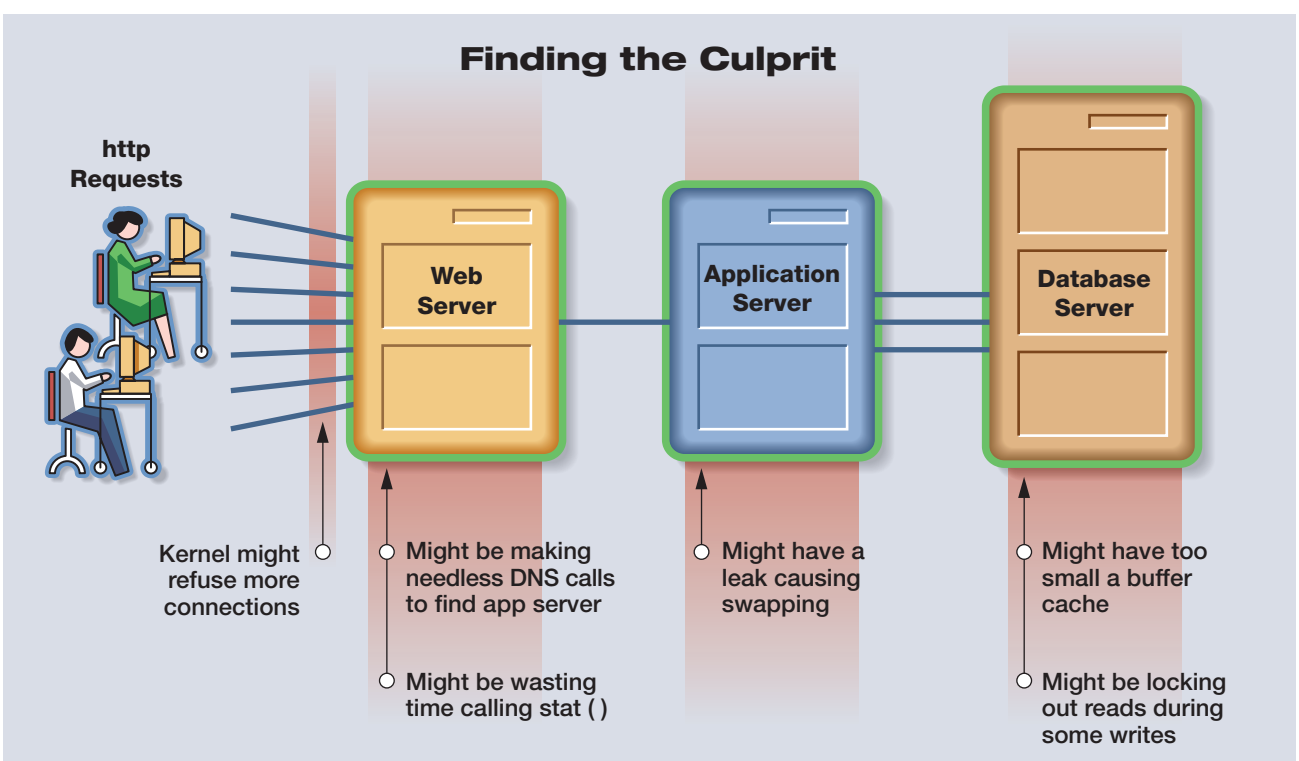

CHART BY ANNIE BISSETT

FIGURE 3 Faults and bottlenecks can occur at several points in a system.

you just can't tune a database anymore, and someone will have to think about other solutions: changing the *SQL* that the application sends to the database, for example, or caching the results.

SEVEN
Being Disorganized

As tempting as it is, the right approach to load testing is to not just pick some random workload, fire it off, and see what happens. Okay, I know you won't be able to control yourself, so you can do that once, but then you have to buckle down and get organized:

Be clear on your goal. Are you qualifying different hardware configurations (for capacity planning or vendor comparison), or are you testing the software? Are you trying to determine how much a load the system can take, or how fast it will be at a pre-determined request rate? Are you checking whether the server can stay up for two days under a constant load, or testing whether it can recover from a sudden burst above its nominal capacity? You might have more than one of these goals, but keep them clear in your mind, and don't confound them in any one testing exercise.

Start small. At first, do just one workflow type, at a low rate. Make sure that works. If you don't do this at first, you'll just end up doing it later when your big test fails; you'll have to backtrack with smaller, narrower tests in order to debug *why* it failed.

Keep records. At the time, you think you'll remember the configuration of the system and how you tested it and what the results were, but in fact you will have forgotten by the next day.

EIGHT
Thinking of Workload in Terms of Hits and Users

People like to talk about "hits" and "users" in Web sites. That is because those quantities can be monetized. That doesn't mean they are terribly informative in defining load.

What does it mean to have a "million users"? Does that mean that one million users log in simultaneously at 9:00 each morning? Are those new users or repeat users? What are they doing once they get to the system?

What does it mean to have a "million hits a day"? How many of those hits are just images or static files?

If you had to do capacity planning for a highway system, would you just ask how many legal drivers exist in the area? No, of course not. You'd also want to know how often they drive, where they go, and how fast a driver they are.

An analogous case holds for Web sites. You like to know the absolute number of unique active users the system has to know about, the number and length of user sessions (like a driver's commute), and the number of each kind of workflow they carry out in those sessions.

This is put in a *workload definition*. Table 1 shows a simplified example.

For simplicity, we just assume that every user has a one-hour session. The "User Likelihood" is the likelihood that a user who is using the system at any point in time is of that class—it is *not* a proportion of all the users who have ever used the system (i.e., it reflects the activity level of users). Note that this table approximates the load with a *small* number of what we are calling "workflows" (see more about workflows in the "About Tools" sidebar later in this article).

Now suppose that we have decided that our system will have to support 10,000 simultaneous user sessions (again, we are interested in sessions, not static users). From that, we can calculate the aggregate request rate for each work-

TABLE 1

USER CLASS	USER LIKELIHOOD	WORKFLOW	RATE	RESPONSE TIME GOAL
Guest	10%	Signup	1/hour	5sec
Newbie	70%	Simplequery	10/hour	2sec
		Checkout	1/hour	10sec
Poweruser	20%	Complexquery	30/hour	5sec
		Checkout	0.1/hour	10sec

TABLE 2

WORKFLOW	RATE
Signup	10k * 10% * 1/hour = 1k/hour ~ 0.3/sec
Simplequery	10k * 70% * 10/hour = 70k/hour ~ 20/sec
Complexquery	10k * 20% * 30/hour = 60k/hour ~ 20/sec
Checkout	10k * 70% * 1/hour + 10k * 20% * 0.1/hour = 7200/hour ~ 2/sec

flow (see Table 2). Note that we use just single-digit precision in the results—there is no point in greater accuracy, given how much approximation there is.

The Superhuman Virtual User

It would seem awfully wasteful to emulate those 10k user sessions by starting up 10k processes (or even 10k threads); each of those processes/ threads would then spend most of its time sleeping (just like a real human). Why not just emulate all those users and sessions with a smaller number of users and sessions, which are superhumanly fast? In that way, the server will get the same number of aggregate requests, just from fewer connections.

Depending on the level of load you want to produce, you may have to take this approach—and depending on the architecture of your system, it might make no difference. But there are two potential problems with it:

1. **Unrealistic reflection of per-session server resource consumption.** In many applications, there are server-side resources that are consumed per each live session. Thus, there can be a big difference between a thousand sessions going at human speed, and ten sessions going at one hundred *times* human speed.

2. **Unrealistically low concurrency.** While you might be producing the desired aggregate request rate with this approach, your virtual superhuman users are always waiting for their previous request to complete before sending the next one. This is unrealistic; your eventual real 10k users certainly won't be coordinating with each other to ensure that each does not send another request before the previous one finishes. Bob in Chicago won't wait for Mary in Seattle to see her Web page finish loading before he clicks on a link.

There are some load test tools (such as the "URL player" http_load) which do not present this dilemma; they can generate actual connections for all those thousands of separate users. But then you sacrifice the ability to emulate some complex workflows. This is an example of the kind of trade-off necessary in the complexity-traffic matrix shown earlier.

One solution to the problem is to take a composite approach: use one technology to test the ability of the server to handle many concurrent sessions (which don't do much), and use another technology which can emulate complex workflows but takes the superhuman-virtual-user approach (in order to be at all scalable).

Determining the Workload

So how do you get this workload information? The traditional QA approach would be to demand a Marketing Requirements Document, and refuse to do any work until one is produced. Of course no reader of this magazine would ever do something so futile as that.

Another approach is to just guess. At least then you are clear about your assumptions, and sometimes it's all you can do. Furthermore, it'll be pretty easy for you to change the numbers anyway once you get your system set up.

Often you can get actual data: this is because your company has, of course, already launched this application (why wait for testing?), or has launched a previous version that is similar to this one. So you can mine the Web server logs to determine usage statistics.

There are lots of Web log analysis programs available, many of them free (e.g., analog and Webalizer). What you need to do is identify some telltale URLs that are signatures for different workflows, and then find out how often each is requested each day. You'll probably want to know the total request rate, as well as how many different users contributed to that request rate. As a sanity check, you'll also want to scan for users who didn't use *any* of your telltale URLs.

Some Real-World Examples

Here are two real-world examples of problems that might be discovered.

1. I was working with a client to test an Internet application whose throughput goal for a particular kind of request was 50/sec. On our first load test, we hit 80/sec. We were ecstatic. Then the server locked up completely after a minute of load. We discovered after a few hours of debugging (it was a complex system) that there was a leak in a module linked into the Apache Web server. The Apache children (the forked httpd processes maintained to handle requests) would grow until all virtual memory was consumed. Then the system would panic.

2. With that same client and application, I decided to also mix in another request type, which performed a write operation. I hesitated over whether I should even bother, because the throughput requirement for that operation was just 1/sec. But it was a good thing I did, because the first time we loaded the system with both requests at their goal throughputs, the response time was unacceptable. Our database administrator discovered that we had been over-aggressive with our bitmap indexes—these were great for read performance when *only* reads occurred, but the extra indexes destroyed write performance, and there was a derivative effect on the read performance. So we took those indexes out.

NINE
Cold Starts

The diners at a restaurant don't all arrive at exactly 6:00 p.m. Similarly, Web sites don't go from receiving zero traffic to very high traffic instantaneously. Yet load testing tools are quite capable of emulating this unrealistic circumstance, so it is easy to get sucked into analyzing problems that don't matter.

Most systems will perform poorly or even fail under such a blitzkrieg attack. Operating systems take time to swap in previously unused text or data pages. Caches take time to build up. Web servers take time to increase the number of running threads or processes

to accommodate load level. But none of that matters in the real world, because you aren't going to be resetting your whole system on a regular basis—just every once in a while, on off hours, when there is plenty of time to spin up.

What matters most is the ability of your system to handle "steady state" load. One way to approach this is to run one mild load test, and don't even bother with the results. Then start the "real" load test. Even with that approach, there will probably be some warm-up necessary, so you'll want to run the test for a while (a few minutes minimum) to wash out the outliers in your measured response times.

TEN

IgnoringErrors

When you get errors, stop testing until they are fixed. No response time measurements are relevant if you are getting errors. (How much work is required of the server to return an error?)

This principle is different from functional testing, where it is often worthwhile to continue performing other tests after you find your first bug.

ELEVEN

Bottlenecking the Load Driver

Perhaps the single most common mistake in load testing (besides not noticing that the server is returning errors) is bottlenecking at the load driver. You might get a false sense of confidence in your system's ability to handle load because you aren't generating the load you think you are.

There are many ways driver bottlenecking might occur:

Disk Spending time writing the load testing results to a log file, rather than sending the next request out.

CPU Spending time running a script language interpreter, rather than sending the next request out.

RAM Spending time swapping virtual memory, rather than sending the next request out (not all your virtual users can fit in RAM).

Network Bandwidth If your average request rate times your average request-response size exceeds the bandwidth between your driver and your server, you will bottleneck at the network.

Sockets Your OS had better allow for you to open as many TCP/IP sockets as you need.

How powerful must the hardware be for your driver? To a certain extent you'll have to figure that out as you go—in some sense, you are really load testing both your driver and your server. Relative to the hardware you have for your server, the driver hardware can be more puny if the server has to do a lot of work for every request. If the requests are "easy," then the driver may have to be practically as powerful as the server. In fact, often a ready source for the driver hardware is the hot spare machine that your operations group has thoughtfully purchased to use when the server has a hardware failure. (Right?) Using identical hardware for the driver and server also has the benefit of allowing you to use the same OS monitoring tools on both hosts.

Because of all the different things that might slow down the load driver, it

AUTHOR SOAPBOX

About Tools

There are many load testing tools available, both free and commercial. A good list may be found at http://www.charm.net/~dmg/qatest/qatweb1.html (see Webinfolink icon at the end of this article).

Most of these tools have their origin in Web server benchmarking or in GUI functional testing, two ill trees whose shadows have stunted the growth of application load testing as a technology in its own right.

Most of the available commercial load testing tools are retreads of GUI functional test tool technology. Having failed to achieve any great successes in their original market, those vendors have moved on to fresh territory.

Load testing requires the use of a *small* number of test scripts—say, five or so. While there are far more functional tests that must be performed on the system, load testing consists of cutting it down to the basic small set of user workflows that are representative of the load. To load test a restaurant, would you order everything on the menu? Of course not. You don't need to include Peking Duck in your load test—just Kung Pao Chicken and a few other popular dishes. In the same way, it is not necessary to include all system functionality when load testing a server.

Because only a few scripts are used, there are no significant savings in sharing the supposedly valuable "test assets" from functional testing, even if those scripts matched user workflows—which they usually don't. Nor are test generation features a huge benefit, as long as script editing is still necessary in the end (and it is, regardless of what vendors say). And the requirement of having full-fledged GUI test and scripting technology makes for poor scalability in test driver hardware. ApacheBench and http_load can produce at least an order of magnitude greater request throughput than one of these tools, on the same driver hardware.

I'm not suggesting the free tools because they are free; I'm suggesting them because I believe the current crop of commercial tools brings a set of features that are a distracting load of their own.

is important to have a load testing tool that can be configured to achieve a set constant rate of requests (or at least to raise an alarm when the rate drops below a certain level). The alternative model of "send requests as fast as you can" or "send requests as fast as N superhuman virtual users can" results in load test results that are not reproducible—they are vulnerable to the vagaries of the resources available on your load testing driver.

TWELVE

Not Reproducing the Production Environment

To do things right, you need to have a dedicated test lab for load testing. It needs to have an isolated network. It has to have hardware that is as close to your production hardware as possible. There can't be other users on the system. If you put system components on separate hosts in production, you need to do that in your test lab as well (this wouldn't matter for functional testing).

Presumably your application has some sort of database (or at least a file system). It is important for you to get this database to be populated with a realistic amount of synthetic data (or better, real data copied from your previous production database). This is because performance of a server can be drastically affected by database size. The four-way *SQL* join that some engineer thought was so clever might run quite fast on a small database, but may never even complete on a realistically-sized database.

Naturally enough, you also need to vary the data used by your virtual users. While you don't need a large number of test scripts in load testing, you *do* need to parameterize the running scripts with a wide variety of data. Because of caching, it isn't much of a challenge to a server if it can satisfy all the requests it receives with the same row in a database table every time.

Note that even in the ideal case, your test lab is not going to have the same physical networking combinations as your live system: (a) your test lab is probably in your internal corporate network, while your production system is probably networked through your co-located ISP connection, and (b) you've got one fat pipe between your load driver and your server—as opposed to a fat pipe between your server and your Internet firewall/router, which is connected to the Internet "cloud" (upstream providers, peering connections, and so on).

If you need to emulate a large number of low-bandwidth connections (which will therefore hold onto the TCP/IP connections for a while), then consider using http_load, which supports such emulated bandwidth throttling.

There are alternatives to automatic load testing, such as a limited production release (i.e., only publishing a URL to a select set of users). You might want to do that anyway, as it has the added benefit of getting you some usability feedback. With a limited release, you can also sniff out problems specific to your production setup. For example, your operations team might have not ever noticed that they have 20% packet loss on their 1.5mbs T1 line to the Internet. You'd never see that problem on your internal test lab, probably running on a 100mbs LAN.

THIRTEEN

Not Knowing When to Stop

There will *always* be some bottleneck in the system. But you only have to be able to identify them until the performance is acceptable.

It is tempting to keep tweaking the system after that point, to make it far surpass your performance goals. But keep in mind that you've got other testing to do too. There is far more to making a Web site or Web application a success than load testing. A fast site can still be unacceptable. You are much better off freezing a *known good* configuration of the system, and spending your testing and engineering resources on other things. STQE

Why Your Web Site May Still Suck

Load testing won't prevent you from producing a Web site that sucks. And rest assured, it might—everyone else's does. You're probably using html frames, aren't you? And your marketing department has mined your site with a bunch of pdf files, and your designer gave you an "entrance tunnel" for a home page, and half of the links don't work, and most of the html isn't actually valid, and your *JavaScript* raises errors on half your users' browsers, and your *Java* applets lock up the other half of the browsers, and no user can figure out how to navigate, and you choke the user with 100KB of images on every page, and the contact information is incorrect (or correct but no one will respond), and your site map is inaccurate, and your search form will crash on any search phrase with punctuation in it, and most of the content is out-of-date, ungrammatical, and misspelled.

The only thing keeping you out of http://www.webpagesthatsuck.com is that your operations staff can't keep your Internet connection and your server up at the same time, thus saving your company the embarrassment of actually exposing users to this horror.

*Mark D. Anderson is the founder of Discerning Software (*www.discerning.com*), a consulting firm specializing in Internet application architecture whose clients have included Lycos, SkyMall, and NetObjects. He is a co-founder of Rec.Net, and is on the technical advisory board of several other Internet startups. Mark holds three patents, and has graduate degrees from MIT and from Berkeley. You can email him at* mda@discerning.com.