Load Testing for the

Modeling Multiple Users in a Realistic Way Requires A Broad Approach

Here's how to simulate workloads using different methods of load generation.

By Alexander Podelko

Software systems can be as complex as life in a coral reef, and, like such ecosystems, software systems tend to grow rapidly. The literature is full of explanations about how to design scalable software, what best practices and design patterns to use, and even how to build models to predict performance.

While it *is* important to create scalable software, theories alone can't guarantee a required level of performance. Testing multiuser applications under realistic as well as stress loads remains the only way to ensure appropriate performance and reliability in production.

Load, performance, stress, scalability and reliability are all terms used to describe this type of testing. Despite efforts to define clear distinctions among them, none of these definitions is widely accepted. There are no clear distinctions, because these terms describe testing from different points of view.

Without delving too deeply into details, I would suggest these definitions:

- Load testing is testing that involves applying a load to the system.
- Performance testing evaluates how well the system performs.
- Stress testing looks at how the system behaves under a heavy load.
- Scalability testing investigates how well the system scales as the load and/or resources are increased.

Quite often, similar processes are used in all these

Alexander Podelko is principal performance engineer at Hyperion Solutions in Stamford, Conn. He holds a Ph.D in computer science from Gubkin University and an MBA from Bellevue University. He can be contacted at Alexander_Podelko@hyperion.com.

Diverse Environment

Photograph by Linda Bair

kinds of testing, and a term to describe the type of testing can be chosen depending on what looks most important in the context.

If you run a test simulating many users and measuring response times, what should you name your test? You can probably refer to it as either the load test or the performance test, and either term would be correct. However, these are not synonyms; rather, they describe different facets of the test.

The term "load testing" is used in this article because we are investigating ways to create load. Everything mentioned here applies to performance, stress, scalability, reliability and other kinds of testing, so long as the system is tested by applying load (while, for example, doing reliability testing by switching off the power is another story).

Based on classic functional testing from one side, and on system performance analysis from another, load testing is emerging as an engineering discipline of its own. Quite often, load testing is combined with tuning, diagnostics and capacity planning.

Sometimes it is difficult to separate performance and load testing. For example, performance testing of a poorly tuned system isn't very meaningful. The typical load testing process is depicted in Figure 1.

We explicitly define two different steps here: "define load" and "create test assets." The "define load" step means the logical description of the load we want to apply (for example, a group of users who log in, navigate to a random item in the catalog, add it to the shopping cart, pay and then log out, with an average 10-second "think time" between each pair of actions). The "create test assets" step means to convert the logical description into something that will physically create load during the "run tests" step. While for manual testing this can just be a description given to each tester, usually it is something else—a program or a script.

Before you can move forward from "define load" to "create test assets," you need to decide how you are going to generate the load. Load generation can be a simple technical step when you know how to do it for your system (compared with other, more complicated steps like collecting requirements, defining load or analyzing results). Unfortunately, load generation quite often is a challenging task for a new system, and it may be downright impossible in the available time frame. It's important to understand all the possible options here; a single approach may not work in all situations. The main choices are to generate load manually (this is really an option only if you have few users), use a load testing tool (software or hardware) or create a program to apply loads. Many tools allow you to use different ways of recording/playback and programming.

In this article I will discuss how to make realistic decisions about which approach and which tool may be most appropriate for any given situation. The material is based on experience involving business applications, so some

limitations may exist when dealing with other environments.

Record and Playback: Virtual Users

The mainstream approach to load testing (at least for distributed business and Internet applications) is to record the communication between two tiers of the system and then play back the automatically created script (usually after proper parameterization). The tools used are usually referred to as "load testing tools," and users simulated by such tools are usually referred to as "virtual users" (see Figure 2). The real client-side software isn't necessary to replay the scripts, so the number of simulated virtual users can be high; this figure is theoretically limited only by the available hardware (each tool has specific hardware requirements that will depend on the type and complexity of scripts).

Both recording and playback occur between the tiers, so the protocol used between the client and the server is the most important aspect.

Other factors, such as the language used in developing the system, the platform the server is deployed on, etc., are usually irrelevant for scripting, although they can give some hints about the protocol used for communication.

Manual load generation isn't a real option if you want to simulate a large number of users.

The process is pretty straightforward when you test a simple Web site or a simple Web application with a thin client. Even a beginner in load testing can quickly create a few scripts and run tests. That's one reason why the record-and-playback approach is so popular.

However, there is a trap in that easiness, too: Load testing really embraces much more. Load testing results should be validated for correctness (if you don't see errors with the load testing tool, that doesn't always mean it works properly) and realism (using unrealistic scenarios is the easiest way to get misleading results).

Moreover, load generation is only one step in load testing; there are many other important parts (such as getting requirements or results analysis), as well as related activities (tuning, diagnostics).

Unfortunately, scripting can be challenging even for a Web application, not to mention other protocols. Recording a script and making it work can be a serious research task, and can often include many try-and-fail iterations. A good load testing tool can help if it supports your protocol.

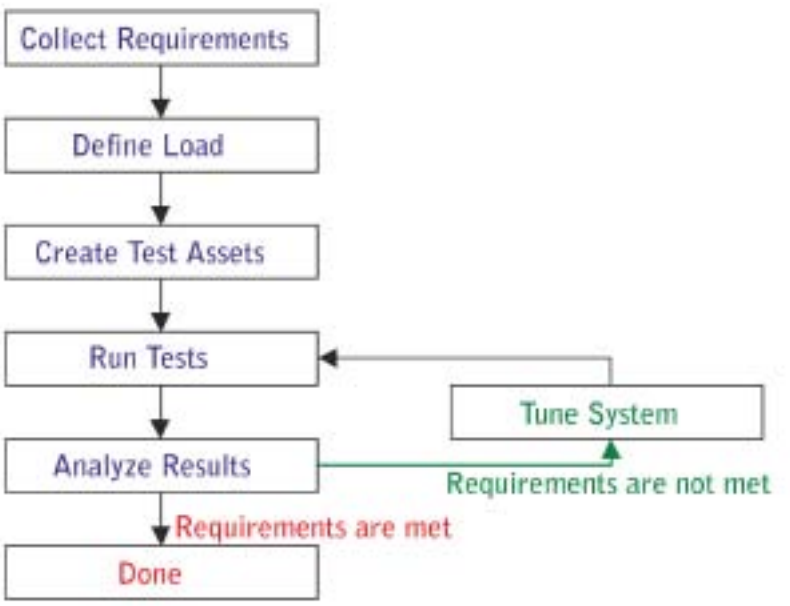
Load Testing Tools

There are a few tools supporting the record-and-playback approach for a variety of protocols. Usually, they are the most mature commercial products. Such enterprise-level load testing tools have many important features. The following features could be considered typical:

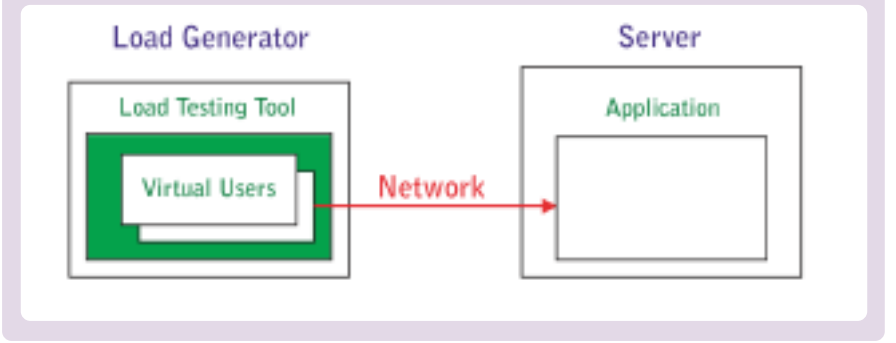
- The ability to record scripts automatically for different protocols
- A powerful scripting language
- Simulation of numerous users (limited mainly by hardware)
- The ability to coordinate test execution from several different computers
- Centralized test management and results analysis
- Support for different environments
- The ability to monitor environments
- The ability to use other approaches to load generation, including the ability to simulate GUI users as well as virtual users, and the ability to extend the scripting language and to make external calls
- The ability to interface with other development and test software: requirements gathering, test management, defect tracking, configuration management, etc.

The list of supported features varies from tool to tool. Examples of powerful multiprotocol tools are Mercury LoadRunner (www.mercury.com), Segue SilkPerformer (www.segue.com), IBM Rational Performance Tester (www.ibm.com/software/rational) and Compuware QALoad (www.compuware.com). For a Web-

1. THE LOAD TESTING PROCESS



2: RECORD AND PLAYBACK: VIRTUAL USERS



only commercial tool, Empirix e-Load (www.empirix.com), having some features of enterprise-level load testing tools, is probably the best-known example.

These five vendors accounted for 95 percent of the worldwide market for distributed automated software quality commercial tools in 2003, according to market researcher IDC.

Many other specialized tools are available, especially for Web technologies. If the number of technologies you'll use is limited, it makes sense to check out such tools (it wasn't a real option for us, considering the multiple technologies we have been working with). Most of them can be found at www.softwareqatest.com/qatweb1.html and at www.testingfaqs.org/t-load.html.

Not all the tools listed at these sites support record and playback, and some require programming scripts from scratch.

The recording abilities of various tools differ significantly. Enterprise-level load testing tools usually can work in more sophisticated environments and can do more correlation automatically (such as getting real cookies, session IDs, etc., from the server, instead of recorded values).

Another area of differentiation among tools involves the infrastructure services they include (test coordination, results analysis, monitoring, integration with other tools, etc.). Most inexpensive or free tools, unfortunately, are weak in this regard.

One more tool worth mentioning is Microsoft's Application Center Test (ACT), which comes with Visual Studio, although it is rather limited in functionality. The Visual Studio 2005 Team System for Software Testers, in beta now, will include a more powerful load testing tool.

There are many open source tools available as well. For example, the following link currently includes some 21 tools: www.opensourcetesting.org/performance.php.

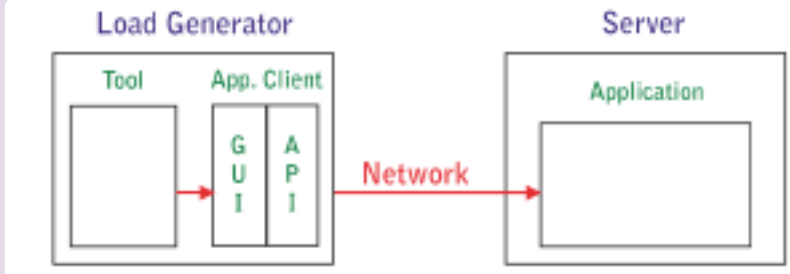
Unfortunately, most open source tools have limited functionality. Probably OpenSTA and Apache JMeter are the best-known and most mature open source tools. OpenSTA (www.opensta.org) is a Web load testing tool originally developed as a commercial tool by Cyrano. OpenSTA stands for Open Systems Testing Architecture. Another branch of the Cyrano code is a commercial tool, QuotiumPRO from Quotium (www.quotium.com).

Apache JMeter (jakarta.apache.org/jmeter) is a 100% Pure Java tool for load and performance testing of HTTP and FTP servers, as well as arbitrary database queries (via JDBC).

Probably the most ambitious open source project is the Eclipse Test & Performance Tools Platform (www.eclipse.org/tptp/index.html).

Load testing appliances (for example, Spirent Avalanche) can be useful for simulating a big number of simple Web users. Scripting is usually limited with these products. It is interesting that Spirent is a partner of Mercury, and it positions its hardware load generator as a complement to Mercury's LoadRunner to create a heavy but simple background load.

3: RECORD AND PLAYBACK: GUI USERS



Choosing a Load Testing Tool

Generally, it would be wrong to say that any one tool is better than another, but one tool can certainly fit better in a particular environment than another. Many factors beyond functionality can affect the choices you'll make. Here are some:

- Familiarity with the tool and other tools from that vendor
- Familiarity with the language that a tool uses (many are based on standard languages such as C, Basic or Java)
- Support
- Price
- The vendor's perspective

On the other hand, it's always good to keep in mind that a load testing tool is only a tool. While you probably need a sophisticated set of tools to create a luxury furniture set, you only need a hammer to nail a picture hanger to the wall.

Limitations

We have been using the record-and-playback approach in most of our projects, but unfortunately, it has several serious limitations:

- It usually doesn't work for testing components.
- Each particular load testing tool supports a limited number of technologies.
- The workload validity in the case of sophisticated logic on the client side is not guaranteed.

These limitations are usually not problems in the case of simple Web applications using a browser as a client, but they can become a seri-

ous problem when you need to test different protocols across the whole software life cycle.

Each load testing tool supports a limited number of technologies (protocols); new or exotic technologies are not usually on the list. Vendors of load test tools add new supported protocols continually, but we often do not have time to wait for a specific protocol to be added—as soon as we get a new product, we need to test it.

For example, we were not able to use recording for the Server Message Block (SMB) protocol, which was later succeeded by the Common Internet File System (CIFS) protocol. It is used when two Microsoft network systems communicate over a network. Its commands are embedded within the transport protocols, like TCP/IP.

Back in 1999, we weren't able to use recording for Microsoft DCOM (Distributed Component Object Model), used for communication between two remote COM components, or Java RMI (Remote Method Invocation), which is used for communication between two remote Java programs.

Although some toolmakers claim their products support these protocols, that support may not work in all environments. Script recording and parameterization is still far from being straightforward, and it often requires a good deal of knowledge about system internals. The question of workload validation is also opened. An illustration of possible problems is shown in the code below. Code Listing 1 shows an example of recording RMI protocol.

RMI CODE EXAMPLES

Listing 1

```
_integer =
   _ireportserver.executeJob(_designjobobject);
_ireportserver.getStatus(new Integer(3));
_ireportserver.getStatus(new Integer(3));
_ireportserver.getStatus(new Integer(3));
_iinstance = _ireportserver.getInstance
   (new Integer(3));
```

Listing 2

```
joID = poReportServer.executeJob(djo);
bStatus = true;
while (bStatus) {
     bStatus = poReportServer.getStatus (joID);
     Thread.sleep(300); }
poReportServer.getInstance(joID);
```

Listing 2 shows the real code producing this RMI communication.

The client polls the server every 300 ms to check the status and to get the result when it is ready. Without having any knowledge of the real code, it will be almost impossible for you to parameterize the script properly—it just calls `getStatus` three times and then calls `getInstance`, even if the result won't be ready yet.

It is possible, therefore, that the record-and-playback approach won't work in your environment, or that using the approach will be too time-consuming and inflexible (as happened many times for us). When you encounter such problems, it's a good time to check out some alternatives and add them to your arsenal.

Record and Playback: GUI Users

Another type of tool that uses the recording approach records all the actions of a real user: mouse moving and clicking and keystrokes. Such tools are usually used for functional and regression testing. Examples are Mercury WinRunner, Mercury QuickTest Professional and Rational Robot. These tools record and play back communication between the user and the client GUI. Virtual users, which are simulated by means of such tools, are often referred to as "GUI users" (see Figure 3).

These tools simulate users in the most accurate way: They really just take the place of an actual user. You get end-to-end response times identical to what real users would see.

For load testing, these GUI tools are usually used in conjunction with the load testing tool from the same toolmaker, which coordinates the execution of multiple GUI scripts and collects the results.

The main problem with such tools is that they require a machine for each user, so it's almost impossible to use them for a large number of users—you'll need the same number of physical boxes as the number of users being simulated. Some tools have the ability to run one user per Windows Terminal Server session, which significantly increases the scalability of that solution (probably up to the low hundreds of users, from a practical point of view).

Another workaround, from Mercury, for example, is to use the low-level graphical Citrix protocol. Still, this is a significantly less scalable approach than record and playback with virtual users, because you'll need to have full working client software, which adds significant overhead on load generating machines.

These tools also could be useful in combination with virtual users to verify VU scripts, to get end-to-end timing, or to increase the number of use cases during load testing reusing functional testing scripts (if, of course, the functional testing tool matches the load testing tool).

Manual Load Generation

Manual load generation isn't a real option if you want to simulate a large

Photograph by Jay Adkins

number of users. Still, in some cases it can be a good option when you need load from a few users and don't have proper tools available, or if you face big problems with scripting. Sometimes a manual test can be a good option in earlier stages of testing—for example, to verify that the system can support concurrent work, or to diagnose locking problems.

One of the concerns to keep in mind with manual testing is that even when each user follows an exact scenario, time variations can occur. The tests are not exactly reproducible, due to variations in human input times, so this approach can hardly be recommended as a long-term solution, even when few users are involved.

It still could be useful to run one or a few users manually in parallel to virtual users' simulated workload, so as to better understand what real users might experience. That's a good way to verify test results: If manual response times match what you see for scripts (keep in mind that virtual users don't have client-side overhead), you have one more proof that your scripts are correct.

Programming

Programming is another approach to load generation. A straightforward way to create a multiuser workload is to develop a special program to generate that workload. Such a program will require access to the API or the source code, as well as some programming work. This approach is often used to test components. No special testing tool is necessary.

In some simple cases, programming could be the best solution from a cost perspective, especially if there is no purchased load testing tool. A starting version could be quickly created by a programmer who is familiar with the API.

A simple test harness, for example, could spawn some threads, and each thread, simulating a real user, could include the same sequence of API calls as the real software employs for that use case. Such a harness should work if the API works. You don't need to worry about what protocol is used for communication.

We have used this approach successfully for component load testing in several projects, and, of course, this approach is widely used by developers. However, efforts to update and maintain the harness increase drastically as soon as you need to add such features as complex user scenarios, centralized test management and results analysis, and coordinated test execution from several computers.

If you have numerous products, as we did, you really need to create something akin to a commercial load testing tool to ensure that all necessary performance and reliability testing will be done. That probably isn't the best choice for a small group of testers.

Custom Load Generation

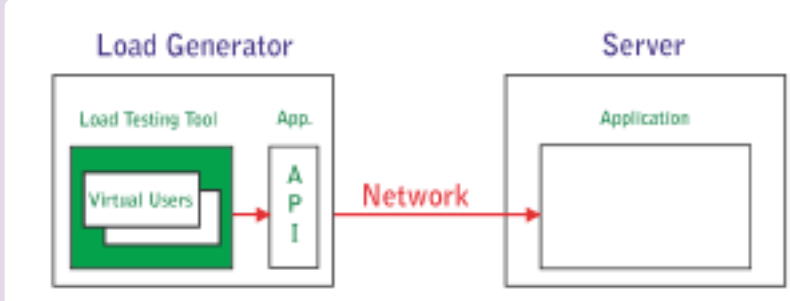
Originally, we used the record-and-playback approach (load testing tools), or we created special programs to generate workload (custom test harnesses) in cases where recording didn't work. Since we experienced numerous problems applying these two approaches to new products utilizing the latest technologies, we came to favor a mixed approach: Develop lightweight, custom software clients (client stubs) to create the proper workload, but use powerful commercial tools to manage them and analyze the results.

The implementation of this approach (we called it "custom load generation"; see Figure 4) depends on the particular load testing tool being used.

For the Rational load testing tool and Mercury's LoadRunner, the original way was to create an external C DLL (or shared library for Unix) and then call functions defined in the DLL from the tool's native scripting language (VU script for Rational TestStudio, Vuser script for Mercury LoadRunner; both are C-like scripting languages).

Another way to implement this

4: CUSTOM LOAD GENERATION



CORRELATED WINSOCK SCRIPT

Listing 3

```
lrs_create_socket("socket0", "TCP", "LocalHost=0",
   "RemoteHost=ess001.hyperion.com:1423", lrsLastArg);
lrs_send("socket0", "buf0", LrsLastArg);
lrs_receive("socket0", "buf1", LrsLastArg);
lrs_send("socket0", "buf2", LrsLastArg);
lrs_receive("socket0", "buf3", LrsLastArg);
lrs_save_searched_string("socket0", LRS_LAST_RECEIVED, "Handle1",
    "LB/BIN=\\x00\\x00\\v\\x00\\x04\\x00", "RB/BIN=\\x04\\x00\\x06\\x00\\x06", 1, 0, -1);
lrs_send("socket0", "buf4", LrsLastArg);
lrs_receive("socket0", "buf5", LrsLastArg);
lrs_close_socket("socket0");
```

Listing 4

```
send  buf22 26165
"\xff\x00\xf0\a"
"\x00\x00\x00\x00\x01\x00\x00\x00\x01\x00\x03\x00"
"d\x00\b\x00"
"y'<Handle1>\x00"
"\b\r\x00\x06\x00\f\x00\x1be\x00\x00\r\x00\xd6\aRN"
"\x1a\x00\x06\x00\x00\x00\x00\x00\x00\x00\x00\b"
"\x00\x00\x00\xe7\x00\x00\x01\x00\x03\x00\x04\x00"
"\x10\x00\xcc\x04\x05\x00\x04\x00\x80\xd0\x05\x00\t"
"\x00\x02\x00\x02\x00\b\x00<\x00\x04"
"FY04\aWorking\tYearTotal\tELEMENT-F\tProduct-P"
"\x10<entity>\t\x00\x02\x00"
```

approach appeared in the later versions of load testing tools: creating a script in a programming language (such as Java or Visual Basic) with the help of templates and special tool-supplied functions.

There are some significant advantages to the custom load generation approach:

- It eliminates your dependency on a third-party tool's ability to support specific protocols.
- It leverages all the features of commercial tools and allows the use of them as a test harness.
- It takes away the need to implement multiuser support, data collection and analysis, reporting, scheduling, etc. This is inherent in the third-party tool.
- It ensures that performance testing of current or future applications can be done for any protocol used to communicate among different tiers. In some instances it is the only way to generate load (as it was for SMB, DCOM and RMI in our case) without developing a full-scale custom harness.

However, there are also some important considerations to keep in mind for the custom load generation approach:

- It requires access to the API or source code.
- It requires additional programming work.
- It requires an understanding of system internals.
- The client environment should sit on all of the load generator machines.
- It requires a commercial tool license covering the necessary number of virtual users.
- The lowest-level transaction that can be measured is an external function.
- It usually requires more resources on client machines (because there is some custom software).
- The results should be carefully interpreted to ensure that there is no contention among client stubs.

Custom load generation has an additional advantage: It allows the workload to be managed in a more user-friendly way, and it simplifies parameterization in some cases.

For example, if you record socket-level traffic, recording and parameterization could take a lot of time. And if you need to change the workload (for example, use new queries), it is almost impossible to change the parameterized script to reflect the new workload. You will probably need to re-record and re-parameterize the script.

When you implement custom load generation, the real query could be read from an input file. Changing the query becomes very easy: You just change the input file without making any changes in the script.

The same is true if different builds of the software are tested. Small changes could impact a low-level-protocol script, but the API is usually more stable. Just install the new build and run the test. There is no new recording and parameterization needed.

Custom Load Generation Examples

All of the examples below are for Mercury LoadRunner, just because it is the tool we use the most. Similar things can be done with the Rational performance tool and probably some other tools.

The first example is a multidimensional analytical engine. Originally, the main way to access it was through the C API. Many products use it, including the Excel add-in. It is possible to record a script using the Winsock protocol (a low-level protocol recording all network communication); Winsock scripts are quite difficult to parameterize and verify.

Listing 3 shows a small extract of a correlated Winsock script. Another part of the script includes the content of each sent or received buffer, as seen in Listing 4.

The script consists of many pages of such binary data. We have a full methodology for how to correlate such scripts, but it is extremely time-consuming (you should go through all pages of the binary data and replace hard-recorded handles with parameters).

Scripts are almost impossible to parameterize; if you need to change anything in the query (for example, run it for another city), you need to

continued on page 30 ►

USING AN EXTERNAL DLL

Listing 5

```
lr_load_dll("c:\\temp\\lr_ess.dll");
pCTX = Init_Context();
hr = Connect(pCTX, "ess01", "user001","password");
...
lr_start_transaction("Mdx_q1");
sprintf(report, "SELECT %s.children on columns,
   %s.children on rows FROM Shipment WHERE
   ([Measures].[Qty Shipped], %s, %s)",
   lr_eval_string("{day}"),  lr_eval_string("{product}"),
   lr_eval_string("{customer}"),
   lr_eval_string("{shipper}"));
hr = RunQuery(pCTX, report);
lr_end_transaction("Mdx_q1",LR_AUTO);
```

YAHOO! News

DSW Data Theft Much Larger Than Estimated

"If We Find the Vulnerabilities, Then We Will Have to Fix Them."

Identity Thieves' Secret Weapon

Personal data on 310,000 may have been stolen

LexisNexis tells of tenfold hike in victims

Retailers Scramble to Bolster Web Site Security

Suspected Hacker Taps Into Air Force Files

Without a Trace

THE OTHER TOP STORIES

Data theft spurs credit card al

◄ continued from page 28

start again from scratch.

For this reason, an external DLL was made for major functions. Listing 5 shows a script using this external DLL. This is almost the whole script (except for a few technical lines), instead of many pages of binary data.

There is an MDX query there, which is generated using day, product, customer and shipper as parameters, so we hit the different spots of the database and avoid caching effects. We can create scripts for each function that was included in the DLL, those covering the main functionality of the product.

Another example is a middleware product: software without a GUI interface, only an administrative console.

We were given functional test scripts in Java from the QE group. The middleware product can use HTTP (with major application servers) or TCP/IP (as a standalone solution). It's possible to run a test script and record HTTP traffic between the script and the server. It is HTTP, but it is just binary data inside the HTTP request body. You can't do anything with the scripts; you can only play them back as is. You need to start from scratch if you want to make a small change.

The solution that we finally settled on was the creation of LoadRunner scripts directly from the test scripts. Just put Java code inside the template and add tool-specific statements (such as lr.start_transaction and lr.end_transaction). Listing 6 shows how the beginning of the script looks.

Why not create a simple program that will start many such scripts in parallel? It is an option, but you'll need to implement all of the infrastructure (coordination, results analysis, monitoring, etc.) yourself. Such work is usually not the right choice for a small group working with several different products, but it does make sense when your tools provide a diverse infrastructure.

Unfortunately, as noted above, you'll find that most of the inexpensive or free tools are weak in providing the necessary elements for this.

Settle on a Strategy

Of course, there is no single best approach to load generation, nor is there a best load testing tool. Various approaches or tools may prove better or worse in a particular context. It is quite possible that a combination of tools and approaches will be necessary in a complex environment.

Choosing the right strategy for load generation can be a challenging task. As you dig into the details of the various tools you might use for a particular project, try to see the big picture of what is available, as well as what can be used for this and for other projects.

Evolve and Adjust

In this article I have described our experience with multiuser workload simulation using a number of different methods of load generation, including record and playback, programming and custom load generation. The latter approach involves implementing lightweight custom client software and running it with a commercial load testing tool, which is used as a harness to collect, analyze and report results, as well as to manage test execution.

Select the set of methods that seems most appropriate to you, and then evolve and adjust your approach to yield the best results.

Remember, software systems often operate in environments that are as diverse and scalable as a coral reef. Testing them under realistic conditions that reflect all that diversity is the only way you can ensure that they will perform reliably when deployed. ☒

A MIDDLEWARE SOLUTION

Listing 6

```
import lrapi.lr;
import com.essbase.api.base.*;
import com.essbase.api.session.*;
...
public int action() {
String s_userName = "system";
String s_password = "password";
lr.enable_redirection(true);
try {
lr.start_transaction("01_Create_API_instance");
ess = IEssbase.Home. create
  (IEssbase.JAPI_VERSION);
lr.end_transaction
  ("01_Create_API_instance", lr.AUTO);
lr.start_transaction("02_SignOn");
IEssDomain dom = ess.signOn(s_userName,
  s_password, s_domainName, s_prefEesSvrName,
  s_orbType, s_port);
lr.end_transaction("02_SignOn", lr.AUTO);
```