Does One Approach Fit All?

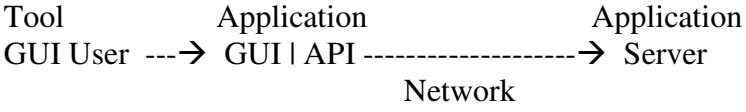
By Alexander Podelko, Arno Sokk, and Leonid Grinshpan

The most challenging task in load testing is to create valid and reproducible multi-user workload matching real-life user scenarios in a limited time. From practical point of view, we could differentiate two main approaches to load generation: recording and programming. The manual load generation isn't a real option if you want to have more than a dozen of users. It still could be sometimes useful to run one or few users manually in parallel to simulated workload to better understand what real users would experience.

Recording

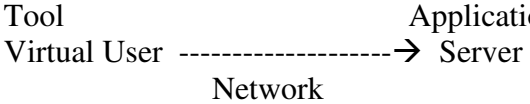
Recording is more common approach if we speak about load testing of complex systems. The main idea of recording is to record communication between two tiers of software and then replay an automatically created script (usually, of course, after proper parameterization). You need a special tool to use this approach.

There are two types of tools using recording approach. The first type records all actions of a real user: mouse moving and clicking, keystrokes. These tools usually are used for functional and regression testing. Examples are Mercury WinRunner or Rational Robot. They record and playback communication between user and client GUI. Users, simulated using such tools, are often referred as GUI users.



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

The main problem with such tools is that they require a real machine for each user, so it is almost impossible to use them for a large number of users – you need the same number of physical boxes as the number of users to simulate. These tools still could be useful in combination with virtual users.
The second type of tools records communication between two tiers of the system (on API or network level) and then replays an automatically created script (usually after proper parameterization). Users, simulated using such tools are often referred as virtual users. The real client-side software isn't necessary to replay the script so the number of simulated virtual users is usually limited only by available hardware.



There are several universal load testing tools on the market today that support such an approach (and are much more specialized). The following features could be considered as standards for such tools:

- Ability to record scripts automatically for different protocols
- Simulating numerous users (limited mainly by hardware configurations)
- Centralized test management and result analysis
- Coordinated test execution from several computers
- Support for different environments
- Ability to simulate GUI users as well as virtual users
- Ability to monitor environment

A list of supported features differs for different load test tools. Examples of powerful universal tools are Mercury LoadRunner (www.merc-int.com), Segue SilkPerformer (www.segue.com), Rational TestStudio (www.rational.com), and Compuware QALoad (www.compuware.com).

There is a lot of specialized tools, especially for Web technologies. If the number of used technologies is limited it has sense to check such tools. It wasn't an option for Hyperion considering multiple technologies used in our products.

We successfully use Mercury LoadRunner in our group, earlier we used Rational TestStudio (PerformanceStudio, preVue-C/S). The choosing of the right load-testing tool is a separate large and interesting topic and is out of the range of this paper. Depending on specific requirements one or another tool could fit better. All examples here are for Mercury LoadRunner and Rational TestStudio just because we worked with them.

We have been using recording approach in most projects but, unfortunately, it has several serious limitations:

- Usually doesn't work for testing components
- Each particular load testing tool supports the limited number of technologies (protocols).
- The question of workload validity in case of sophisticated logic on the client side is open.

All that usually is not a problem in case of web applications using browser as a client, but it becomes a serious problem when you need to test different protocols across the whole software lifecycle, as we need at Hyperion.

Each load testing tool supports the limited number of technologies (protocols). New or exotic technologies are not usually on the list. Vendors of load test tools add new supported protocols all the time, but we often haven't time to wait when the specific protocol would 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 SMB (Server Message Block) protocol, later succeeded by 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, and Java RMI (Remote Method Invocation), used for communication between two remote Java programs.

Although some tools claim support for these protocols, but script recording and parameterization still far from straightforward and often require a good knowledge of system internals. The question of workload validity is also opened. A good illustration of possible problems is the code below.

Here is an example of recording with RMI protocol:

```
_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));
```

Here is the real code producing this RMI communication:

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

Here the client polls the server each 300 ms to check the status and gets the result as soon as it is ready. Without knowledge of the real code it is almost impossible to parameterize the script properly – it could call getInstance before the result would be ready that would generate error.

Programming

Programming is another approach to load generation. A straightforward way to create a multi-user workload is to develop a custom test harness (a special program to generate workload). It requires access to the API or source code and some programming work. It is often used to test components. No special testing tool is necessary.

In some simple cases it could be the best solution (from a cost perspective, especially if there is no purchased load testing tool). A simple harness could spawn some threads and each thread will simulate a real user.

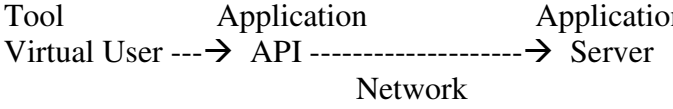
We used this approach for component load testing in several projects successfully (and, of course, this approach is widely used by developers). However, efforts to update and maintain the harness increase drastically as soon as you need to add such features as, for example:

- Complex user scenarios
- Centralized test management and result analysis
- Coordinated test execution from several computers
- Ability to run GUI users as well as virtual ones

If you have numerous products (as Hyperion does) you need really to create something like a commercial load testing tool to assure all necessary performance and reliability testing. It probably isn't the best choice for a small group.

Custom Load Generation

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



The implementation of this approach (we call it custom load generation) depends on the particular load testing tool. For Rational TestStudio and Mercury LoadRunner, the more mature way is to create an external C dll (or shared library for UNIX) and then call functions defined in the dll from the tool's native script language (VU script for Rational TestStudio, Vuser script for Mercury LoadRunner; both are C-like script languages).

Another way to implement this approach appeared in the latest versions of load testing tools: to create a script in a programming language (Visual Basic, Java, VBScript, JavaScript, C and C++ for Mercury LoadRunner; Java, Visual Basic and shell script for Rational TestStudio) with the help of templates and special tool-supplied functions.

These are the significant advantages of this custom load generation approach:

- Eliminates dependency of the third-party tool's ability to support specific protocols
- Leverages all the features of commercial tools and allows using them as a test harness
- No need to implement multi-user support, data collection and analysis, reporting, scheduling, etc. This is inherent in the third-party tool.
- 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 quickly create a performance testing environment (as it was for SMB, DCOM and RMI, in our case) without developing a full-scale custom harness.

But this approach has one more advantage: it allows managing the workload in a more user-friendly way and simplifies parameterization.

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 probably need to re-record and re-parameterize the script.

When you implement custom load generation, the real query, for example, could be read from an input file and changing the query becomes very easy: you just change the input file without 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 API is usually more stable. Just install the new build and run the test. There is no new recording and parameterization needed.

But, of course, there are some considerations to keep in mind for the custom load generation approach:

- Requires access to API or source code
- Require additional programming work
- Requires commercial tool license for necessary number of virtual users
- Minimal transaction that could be measured is an external function
- Usually requires more resources on client machines (since there is some custom software)
- Results should be cautiously interpreted (insure that there is no contention between client stubs)

Implementations for Rational TestStudio (PerformanceStudio)

The mature way to create a custom software client in Rational Test is to implement it as an external dll. It should be a set of C functions compiled as a dll (the functions could be written in C++ and declared as extern “C”).

This external dll should then be placed in a specific directory and references to it are added to the script properties.

There are some limitations. Only a limited set of types can be function arguments:

C types	VU types
Int	Int
char *	string /*read only */
char *	string:maxsize /*writable */

int *	Int[], int[][], int[][][]
char **	string[], string[][], string [][][]

For example, the external dll for C function void DoSomething (int n) would be:

```
extern "C"{__declspec(dllexport) void DoSomething(int n);}
#include "windows.h"

void DoSomething(int n)
{
        …   //some processing
}
```

VU script (C-like Virtual User language script using by Rational TestStudio) to call this function would be:

```
#include <VU.h>
external_C proc DoSomething(n)
int n;
{}
int p=3000; //a parameter

{
  start_time ["T1"];
  DoSomething(p);
  stop_time ["T1"];
}
```

Since we use Rational TestStudio as a framework to run this VU script we get a comprehensive set of available reports, abilities to simulate numerous users from several machines and create complex scenarios as well as other useful features of this load testing automatically.

Usually functions (like DoSomething in the example above) included in the dll are a wrapper around specific C or C++ API functions. In simpler cases the C API could be directly called from the VU script, without the creation of wrapping functions.

Unfortunately if the software is written in another language it isn't too simple to use this approach. For example, to run a custom client written in Java, the java virtual machine (JVM) was started each time with parameters to run the particular report by the Win32 CreateProcess function.

This probably wasn't the most elegant approach but it worked and allowed all necessary performance testing to be completed.

Rational 2001 TestManager can work not only with SQABasic (for GUI testing) and VU language scripts but also with Java, Visual Basic and shell scripts (test script services). It could significantly simplify using custom clients written in these languages.

For example, instead of the awkward starting of JVM by CreateProcess for each report for Java custom client as in the above example (or some other non-trivial transformation), it is possible to create a Java script that will do everything, something like (without exception handling):

```
import java.io.*;
import com.rational.test.tss.*;

public class hr extends com.rational.test.tss.TestScript
{
  public void testMain(String args[])
  {
     int n=30000;
     myClass mc;

     new hr();
     mc = new myClass();

     TSSMeasure.timerStart("T1");
     mc.doSomething(n);
     TSSMeasure.timerStop("T1");
  }
}
```

This Java script is now just a standard script for Rational TestStudio and utilizes all features of the load testing tool. Results could be seen through a set of reports available in Rational TestStudio.

Implementations for Mercury LoadRunner

Using LoadRunner, a custom software module could be implemented as an external dll. LoadRunner could use the lr_load_dll function to load the external dll (shared library) in Vuser script or it could be defined globally in the vugen.dat file. Then functions defined in the dll could be used without declaration in the script.

For the external dll described above with void DoSomething(int n) C function, a simple Vuser script would look like:

```
#include "as_web.h"

Action1()
{
```

```
  int p=3000; //a parameter

  lr_load_dll("c:\\DoSomething.dll");
  lr_start_transaction("T1");
  DoSomething(p);
  lr_end_transaction("T1", LR_AUTO);
  return 0;
}
```

It is just a standard Vuser script that is running inside the LoadRunner framework. It could be a part of a complex scenario and LoadRunner, which produces a comprehensive set of various reports, would analyze its results.

Another way could be to program a script in Visual Basic, Java, VBScript, JavaScript, C or C++ using templates and LoadRunner's functions (just create a new script and start to add code).

Java script, for example, could look like:

```
import lrapi.lr;
public class Actions
{
        public int init() {
                return 0;
        }//end of init

        public int action() {
           int n=3000;
           myClass mc;
           try {
                mc = new myClass();
                lr.start_transaction ("T1");
                mc.doSomething(n);
                lr.end_transaction ("T1", lr.AUTO);
           } catch (EssException x){System.out.println("Error: " + x.getMessage());}
        }

        public int end() {
                return 0;
        }//end of end
}
```

Summary

This paper described our experience of multi-user workload simulation using different methods of load generations: recording, programming, and a mixed method (custom load generation): implementing low-weight 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 manage test execution.

The custom load generation eliminates some limitations of the recording approach:

- It usually doesn't work for testing components
- Each particular load testing tool supports the limited number of protocols
- Inflexibility of low-level scripts
- Workload validity is questionable in case of sophisticated logic on the client side

Of course, the custom load generation has some disadvantages and requires knowledge of API – no sense to use it where recording works fine (like thin-client Web applications), but sometimes (often enough in our case) it is a very good way to simplify the creation of a multi-user workload.