Software Testing Innovations Series

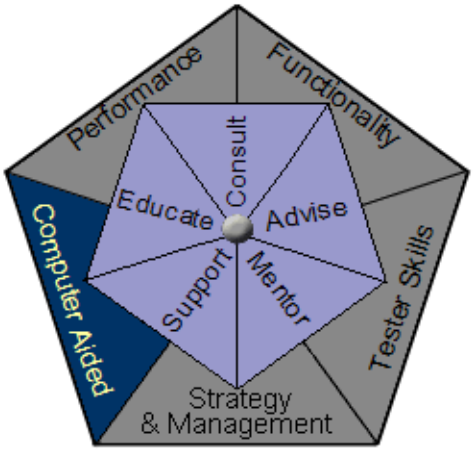
by:
R. Scott Barber

Automated Testing for Embedded Devices

As the number of new applications being developed for wireless/embedded devices such as PDAs, pagers, and cell phones increases, so does the demand for tools to automate the testing process on these new platforms. Through several consulting engagements, Chris Walters and I have had the opportunity to pioneer the use of Rational TestStudio to automate functional (or GUI) and performance testing of new applications developed to run on a variety of embedded devices. This automation is made possible by the use of emulators — the same emulators used by developers of applications for embedded devices. In this article, we're going to show you how to use Rational TestStudio to record and play back test scripts against emulators. Case studies and examples will illustrate this approach to automated testing for embedded devices. We do *not* address unit or code testing here.

This article assumes that you're already experienced at using TestStudio to automate testing on a PC or Windows-based machine. Furthermore, employing some of the methods discussed in this article requires significant technical expertise and/or involvement of an embedded device software developer. Because we explore concepts in this article that are often viewed very differently by the embedded device and the personal computing software development industries, we've consulted not only software testing experts but also a current instructor of embedded device programming at MIT, a notable employee of Draper Laboratory, and numerous experts on Web and client-server automated testing from across the United States.

Which Devices Are We Talking About Testing, Exactly?



Before we describe our approach to automated testing of applications for embedded devices, we need to clarify exactly which devices we mean. Let's start with a definition of *embedded devices*. Embedded devices are pieces of hardware that have software physically embedded on the device — that is, burned onto chips rather than temporarily written to a disk drive. Once the software and/or operating system is deployed onto an embedded device, it's essentially unchangeable.

For our purposes, there are actually three categories of embedded devices:

- Those with fully featured operating systems
- Those that are also real-time systems
- Those that are neither of the above

Embedded devices with fully featured operating systems, the first category, are the ones we're talking about testing. The operating systems on these

devices are fully featured because they allow third-party software to be installed and executed by the end user. For example, we know that it's possible, even easy, to install new software programs onto a PC or a PDA. Thus, the operating systems on the PC and the PDA are fully featured operating systems. By contrast, we can't install new software onto our home stereos or the anti-lock braking systems in our cars, devices that we would describe as having only an operating system kernel. Furthermore, embedded devices with fully featured operating systems typically have user interfaces that include interactive visual displays. This is important to note because it's how users interface with the system that we're testing when we do functional or GUI testing.

We aren't going to talk here about testing embedded devices that are also real-time systems. Real-time software and devices, such as missile guidance systems and anti-lock braking systems, operate at an extremely predictable rate, as opposed to devices where command processing time can vary greatly based on a variety of factors. On a real-time system, a late computation is a wrong computation and a missed deadline constitutes a critical failure of the system. While the overwhelming majority of real-time systems and devices are also embedded devices, the majority of embedded devices aren't real-time systems. (If you want to understand real-time systems better, see *Doing Hard Time: Developing Real-Time Systems with UML, Objects, Frameworks and Patterns* by Bruce Powel Douglass, which served as the source of our definitions here.)

The most significant reason the testing methods we're about to discuss aren't appropriate for real-time systems is that they're likely to cause failure of real-time devices just by their execution. Most real-time devices are so precise in the way they operate that virtually any external interaction, other than those the device was designed for, will affect the system in some way as to make any detected results potentially unrepresentative of what the results would be on the actual device. Furthermore, since real-time systems are coupled so tightly with the hardware they reside on, it's nearly impossible to create an emulator reliable enough to use for testing. Several companies have developed specialty testing software to test these types of devices. Rational, for example, has added Rational Test RealTime to their testing suite to test these types of devices. This testing isn't really what people in the PC world think of as functional testing or what those in the embedded device world refer to as GUI testing, but rather a combination of extended unit (code) testing with system and integration testing. Each of these is based on testing individual inputs and outputs of functions and modules, rather than testing how users interface with the system.

Figure 1 shows how embedded devices with fully featured operating systems relate to other types of devices that we're not going to talk about testing. From this diagram, you can see that cell phones, pagers, PDAs, and WinCE devices don't really fit exclusively into either the PC or the embedded device category. These devices do have user interfaces that need to be tested, like traditional applications, but don't reside on platforms that are conducive to loading automated testing software. That's where the use of emulators comes in.

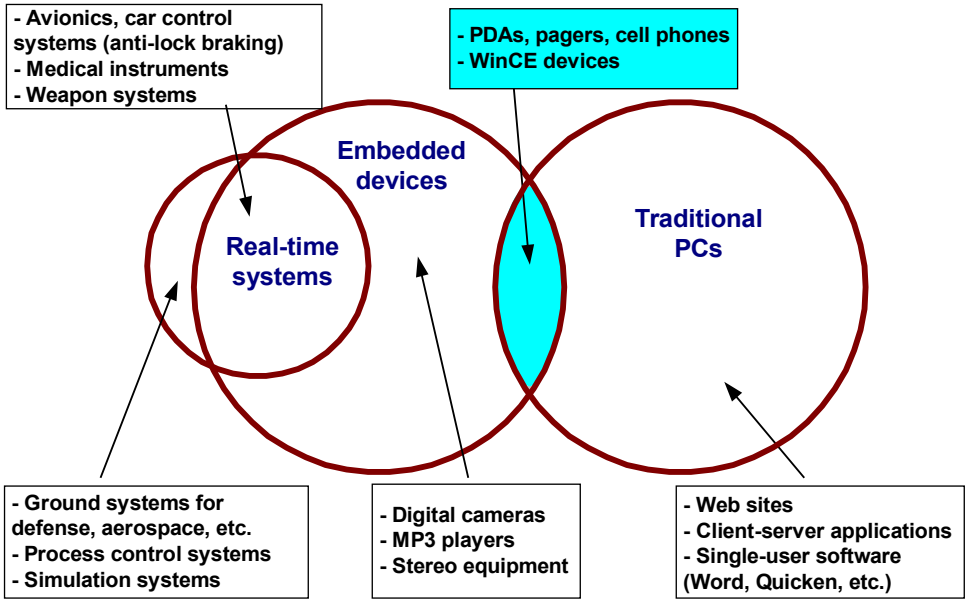


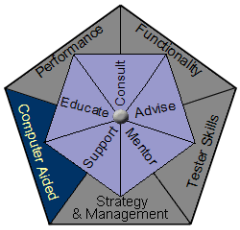
Figure 1: How embedded devices with fully featured operating systems relate to other devices

Using Emulators with TestStudio

To test our embedded device applications, we want to employ PC-based automated functional testing, which is the type of automation used to increase functional and regression test efficiency on Web-based and client-server applications. In some cases, we'll also be able to conduct performance testing on our embedded device applications. In order for us to take this approach, though, the application being tested must have a user interface that's accessed through a computer (generally a PC) that can also house automated testing software, such as Rational Robot. If there's no user interface, or if the user interface resides exclusively on a system that can't also support an automated testing tool, neither functional, regression, nor performance testing can be automated as it traditionally is in PC-based testing engagements.

Well, we can't load Rational TestStudio onto a PDA to automate functional tests directly on the application being developed, nor can we plug (most) PDAs into the corporate network to record performance tests. So to get around this problem, we'll use an emulator. An *emulator* is a piece of software that allows an application written for one operating system to run on another operating system. For example, you may remember the days when running Microsoft Office products on a Macintosh computer required an emulator. The emulator generally caused the application to run more slowly than it would have on a PC, but otherwise it operated identically. It's the same idea with emulators for embedded devices. (Note that emulators are different from simulators, which generally allow users to experience what a thing will look and feel like but which don't function the same way behind the scenes. For example, a flight simulator may allow a person to interact with controls that are like a real plane's and get visual feedback on the display, but the code to run the simulator isn't often the same code that's used in the actual plane.)

Developers who write applications for embedded devices with fully featured operating systems often write and unit test their software on a Windows or Unix platform through the use of an emulator that



allows the software under development to think and act like it's in its native environment, which would be the embedded device. It makes sense that if the emulator is sufficient as a development environment, it's also valuable as a test environment. It's important to perform only functional tests (and performance tests if the emulator has the additional feature described below) against emulators, and specifically only against emulators that the developers feel are accurate. If you're not sure if your development team uses or trusts an emulator, you should ask. You must also remember that this type of testing doesn't replace unit or code-level testing.

Some emulators of wireless devices actually connect to the wireless gateway via Internet or network, so the result is the same as using the actual device to connect wirelessly to the gateway through a cellular tower. Emulators that do connect to the gateway, allow interaction with the rest of the world. For example, using an emulator for a RIM handheld device, you can send and receive email just like you would on the actual device. With this kind of emulator, performance tests can be recorded and executed just like they can against traditional applications, subject to the limitations mentioned below in the section on performance testing. All of the same considerations apply.

The approach to emulator-based testing is very simple. Just install the emulator for the device you're testing on your PC or Windows-based computer, configure it according to its documentation, install the application you want to test, and start performing your GUI or functional testing the same way you always have. The requirements and test-planning processes are identical. Test cases can be written the same way you're used to writing them, and test scripts (manual or automated) can be recorded and executed with Rational TestStudio the way you're already familiar with. We won't belabor the point by detailing the RUP test process here, but it all applies the same as if you were testing a client-server or Web application.

There *is* one additional step to the functional testing process: manual verification of test results on the embedded device(s). On rare occasions, particularly with a new emulator, there may be differences in how the application interacts with the actual device. Now, before you stop reading and decide, "If I have to execute my tests manually anyway, I'm not going to automate!" remember that even with traditional automated testing, you generally have a test environment and a production environment, and sometimes you have to manually verify your test results in the production environment as well. Granted, this is normally because of data issues or dynamically generated Web sites that differ based on the referring URL, or even because some users of the application will be accessing the system through an operating system that your automated testing system doesn't support. Regardless of the reasons, this manual verification of automated tests is probably already part of your testing process.

If differences are detected between the application running on the emulator and the application running on the actual device, those differences need to be reported to not only the application developer but also the developer of the emulator.

Using Emulators to Automate Functional or GUI Testing

Now we'll focus on using an emulator with TestStudio to automate functional or GUI testing. We'll discuss the benefits and drawbacks to this approach when applied to functional testing and will then present a case study and an example.

Benefits

So, what are the benefits of using this approach? They're the same as the benefits of automating tests against traditional applications. Automated tests are consistent, faster to execute, and repeatable once they're developed. Rational says any test that's likely to be executed five or more times is more efficient to automate than to execute manually each time. In the case of embedded devices, we would say that rule of thumb should be reduced to three times. There are two reasons for this: first, there are many fewer objects to verify in an embedded application, and second, it takes most users much longer to interact with a phone or a PDA than it does for them to interact with their PC. Think about how much longer it takes you to type a new address and phone number into your cell phone than it does to type that same address and phone number into Microsoft Outlook on your desktop PC. You may not mind typing that information five times on your desktop PC, but after about the second time on the cell phone, you'll no doubt be ready to automate.

Since automating against an emulator is often your only viable approach to regression testing, this is a huge benefit. Every change to the application, as you know, may have unexpected effects on other parts of the application. Relying on repeated manual functional testing to catch all possible regression testing issues is unreliable at best. Experience shows that once something is verified to work on both the emulator and the actual device, as long as it continues to work on the emulator it's a very safe bet that it will also continue to work on the actual device.

Drawbacks

As mentioned above, the biggest drawback to emulator-based automated functional testing is that the emulator may not accurately represent the behavior of the actual device. Again, this is generally only a problem encountered on new devices or emulators. Once the first difference is detected between the behavior and the actual device, extra care should be taken to double-check the accuracy of the emulator.

The other drawback is that it's often easier to interact with an emulator than with an actual device, so the tester doesn't get a true sense of application usability. Emulators will normally allow for the use of mouse and keyboard, while actual devices obviously don't. While this doesn't directly affect functional testing, it's always a good idea to do some amount of testing on the actual device to evaluate its usability.

Case Study

During February and March 2001, Chris and I were asked to functionally test a new application written for the Blackberry RIM pager and handheld devices. Nondisclosure agreements preclude our describing the specific application, but the following case study discusses the approach used, the lessons learned, and some ideas for the use of Robot GUI scripts that we developed against the Blackberry emulator (see Figure 2).

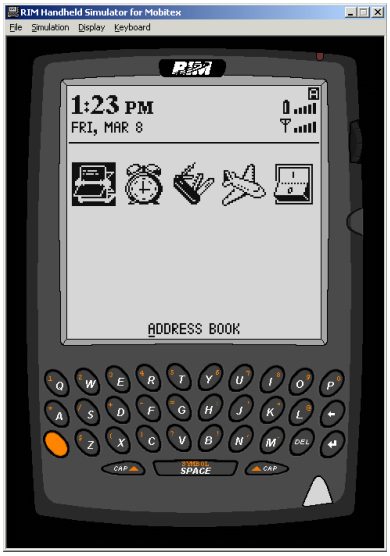


Figure 2: Blackberry RIM Handheld emulator

Initially, the project manager proposed the use of Rational Suite Enterprise to track requirements (RequisitePro), defects (ClearQuest), and test cases and manual test scripts. All tests were planned to be executed on the Blackberry pager and handheld devices. After all of the requirements were entered and test cases and manual test scripts were written, the manual tests were begun.

After a short time, the testers complained that the devices were difficult to type on for extended periods of time. One tester had been trying out the idea of automating test scripts against the emulator. Because this tester met with significant success, the remainder of the tests were automated.

During the rest of the engagement, no deviation was detected between the emulator and the actual devices. With this particular emulator, it wasn't possible to verify actual text on the screen, so a region image verification point was used. The results screen was also written out to a file for future reference along with all of the input data. This file was printed and delivered to the client so they could validate results if they wanted, and was used by testers to manually spot-check results on the actual device.

While performance testing wasn't part of the engagement, the project team tested the concept of recording and playing back VU scripts against the emulator. Since this emulator actually accessed the gateway via Internet, the scripts were successful. We were able to send email from several virtual users and collect response times from the gateway but were unable to actually view statistics on the gateway to evaluate the response times fully.

Example

We can't share with you any of the actual scripts that we used to test the Blackberry devices, so instead we'll show you an example of functional testing that uses the Motorola i85s Phone emulator and a memopad application written by Chris.

We used the GUI script shown in Listing 1 during development of the memopad application, both for its value as a regression test and as a training exercise. The script launches the memopad application, creates a new memo, adds text to the memo, takes a screen shot of the memo, saves the memo, and

closes the application. The script goes on to open the application a second time, open the just-saved memo, take a second screen shot, and close the application. The script saves a file with the data that was entered into the memo and both screen shots. This way the tester can have side-by-side visual confirmation that the script appears as it should after it's been saved and reopened.

```
'Company: Noblestar
'Author: Chris Walters

Option Explicit
Dim Result As Integer

Sub Launch
  StartApplication "C:\java\j2mewtk1.0.3\bin\emulatorw.exe -Xdescriptor:"
  Window SetContext, "Caption=Select A JAD file To Run", ""
  Browser SetApplet,"JavaCaption=Select A JAD file To Run",""
  'EditBox Click, "Type=EditBox;Name=File name:", "Coords=83,15"
  InputKeys "C:\Documents and Settings\Administrator\Desktop\Memopad.jad"
  'Run
  PushButton Click, "Type=PushButton;Name=Run"
  Window SetContext, "Caption=Motorola_i85s", ""
  'Launch
  Window Click, "", "Coords=153,299"
End Sub

Sub AddMemo   'New
  Window Click, "", "Coords=153,299"
  InputKeys "{RIGHT}test"
  DelayFor 1000
  InputKeys "{ENTER}"    'Close
  Window SetContext, "Caption=Motorola_i85s", ""
  Window Click, "", "Coords=54,300"   'Save Changes
  Window Click, "", "Coords=151,299"
End Sub

Sub Edit(memo As String)   'Open
  Window Click, "", "Coords=154,340"
  InputKeys "{RIGHT}" & memo
  DelayFor 1000
  InputKeys "{ENTER}"
  Result = WindowVP (CompareImage, "Caption=Motorola_i85s", "VP=Memo Check")
  'Close
  Window Click, "", "Coords=55,298"   'Save Changes
  Window Click, "", "Coords=152,300"
  Window CloseWin, "", ""
End Sub

Sub Main
  Launch
  AddMemo
  Edit("ing")
  Launch
  Edit("")
End Sub
```

Listing 1: Script to test our memopad application on the Motorola i85s Phone emulator

Using Emulators to Automate Performance Testing

As mentioned earlier, emulators that connect to the wireless gateway can be used to do performance testing. It's important to note that this type of performance testing doesn't test the actual performance of the device, nor in most cases does it incorporate any of the wireless part of the network into the testing. What it does test is some of the functions of the wireless gateway, and any application servers, databases, and the like on the server side. As Figure 3 shows, performance testing using emulators will generate load to every part of the infrastructure that the application developer has any control over.

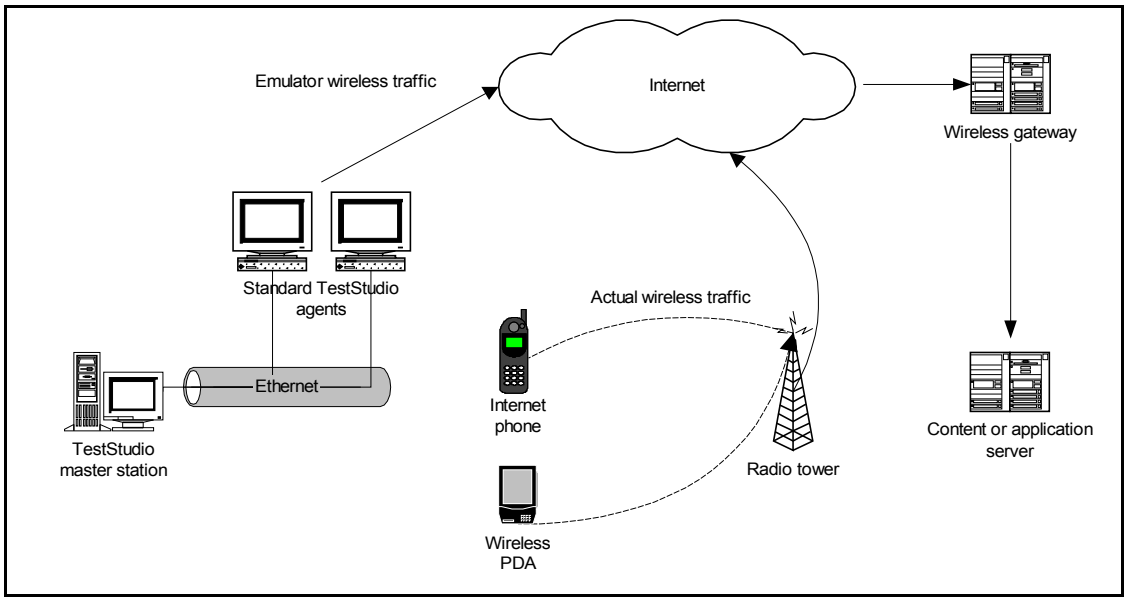
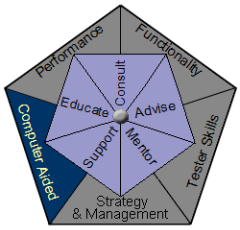


Figure 3: Performance testing using an emulator that connects to the wireless gateway via the Internet

The approach to emulator-based performance testing is quite similar to emulator-based functional testing. The main difference is that most wireless applications don't use protocols that are supported by Rational. Many of them are TCP/IP based but don't often use HTTP. This means that the script will likely not recognize variables that should be part of datapools and may not recognize session IDs. You'll have to be familiar with the protocol to be able to effectively edit the scripts. To illustrate this, let's compare a couple of code samples. Listing 2 is a code sample from a traditional HTTP TestStudio VU script.

```
/*
   ->-> Session File Information <-<-
    Created: Mon Mar 18 15:01:31 2002
                                                                    Name:
\\Deathstar\Repos\DefyWire\TestDatastore\DefaultTestScriptDatastore\TMS_Sessions\Noblestar.w
ch
     Type: Rational Robot - API
        (with Wininet HTTP)
        (with Winsock1 Data)
*/

#include <VU.h>
{
push Http_control = HTTP_PARTIAL_OK | HTTP_CACHE_OK | HTTP_REDIRECT_OK;
```




```
push Timeout_scale = 200; /* Set timeouts to 200% of maximum response time */
push Think_def = "LR";
Min_tmout = 120000;    /* Set minimum Timeout_val to 2 minutes     */
push Timeout_val = Min_tmout;
push Think_avg = 0;

noblestar_com = http_request ["Noblest~001"] "noblestar.com:80",
    HTTP_CONN_DIRECT,
  "GET / HTTP/1.1\r\n"
  "Accept: */*\r\n"
  "Accept-Language: en-us\r\n"
  "Accept-Encoding: gzip, deflate\r\n"
  "User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)\r\n"
  "Host: noblestar.com\r\n"
  "Connection: Keep-Alive\r\n"
  "\r\n";

  { string SgenURI_001; }
  SgenURI_001 = _reference_URI; /* Save "Referer:" string */
```

Listing 2: Code sample from an HTTP TestStudio VU script

Compare that to the sample in Listing 3, taken from a recording of a custom Java 2 Micro Edition (J2ME) application called Picotop™ developed by Defywire.

```
/*
   ->-> Session File Information <-<-
    Created: Mon Mar 4 16:30:56 2002
                                                                                Name:
C:\Projects\DefyWire\TestDatastore\DefaultTestScriptDatastore\TMS_Sessions\PicoTop.wch
     Type: CS-Network
*/

#include <VU.h>
{
push Timeout_scale = 200; /* Set timeouts to 200% of maximum response time */
push Think_def = "LR";
Min_tmout = 120000;    /* Set minimum Timeout_val to 2 minutes     */
push Timeout_val = Min_tmout;
push Think_cpu_threshold = 150;

D192_168_21_118 = sock_connect("picotes~001", "192.168.21.118:1974");

{ INFO SERVER "192.168.21.118"="192.168.21.118"; } /*1*/

set Server_connection = D192_168_21_118;

push Think_avg = 0;        /* 0 < 150 (cpu processing delay) */

sock_send
  "CS_LOGIN:69:<du>NOBODY<et>CLEARTEXT<pr>c71754327bca7d<rt>357<x1>8978d"
      "959a8d040</>";

sock_nrecv ["picotes~002"] 41;

set Think_avg = 209;             /* 209 >= 150 (user think time delay) */

sock_send
  "GENERIC_MSG:61:CS_LOGIN_UIDPSWD:41:<ui>EEarth<pwd>password<devfree>26"
"5948</>";
```

Listing 3: Code sample from a recording of a custom Java 2 Micro Edition (J2ME) application

What you see is that the scripts start the same, but the recording for the Picotop™ application uses a custom protocol. TestStudio captures this protocol correctly, but to completely understand it, some explanation by the developers of the application was required.

Benefits

Emulator-based performance testing for a wireless application does have significant benefits and drawbacks. It's interesting to note that most of the benefits and drawbacks actually involve the same issues.

The most obvious benefit, of course, is that being able to automate and simulate multiple concurrent users of a wireless device makes a potentially complex task very easy. Without the use of emulators, there are only two ways to accomplish this task. One is to obtain a large number of the wireless device you want to test, load and configure the application-under-test, then find some way to perform the activities you want to test on all the devices at the same time (often referred to as the "get a bunch of interns" method). The other is to write an interface to allow a single computer to directly access and control a wireless device (in a manner similar to Rational's TestAgent software), but that's a very complex solution.

The other significant benefit results from the fact that using emulators to test puts stress only on the wireless gateways and the content or application servers and thus eliminates the wireless communication component from the test. This is good because the application developer has no control over the performance of the wireless network. By only stressing the parts of the infrastructure that the developer has control over, you can be assured that if the application performs acceptably through the emulator, any performance issues encountered in production are a result of the wireless connection.

Drawbacks

The obvious drawback is that the wireless communication component won't be included in any performance statistics, and there'll be no way to verify what the actual user response time will be. While it's possible to independently verify performance metrics obtained from the Wireless Service Provider (WSP) about the capabilities of the radio towers, that's beyond the scope of this article. Nor is there any way to determine if the WSP is providing the speed or bandwidth agreed upon in the Service Level Agreement (SLA).

The workaround for this is what we refer to as a manual performance spot-check. This involves manually performing actions during execution of various load scenarios and timing the response. The average times from the automated performance test can then be subtracted from the observed times to determine what portion of the total response time is due to gateway and server performance vs. WSP performance.

Case Study

During March 2002, we conducted performance testing for the Picotop™ application to determine the optimal settings for various host machines. The testing used the Motorola i85s Phone emulator, as seen in Figure 4. As a result, the correct hardware and configurations were determined for various user volumes. Unfortunately, this product isn't available in a freeware version to demonstrate the

performance results, and the actual results haven't been approved for public release. We can summarize by saying that the performance numbers collected have served as a significant selling point for the application.

Figure 4: Motorola i85s Phone emulator

Example

Since you can't do a walkthrough of the work done on the Picotop™ application, this example uses a publicly accessible MIDP application that generates network traffic that can be recorded from the emulator. The program muTelnet (which you can download) is a complete telnet client that runs on the i85s phone. This allows you to generate traffic from the emulator that would be identical to someone using the phone to log in to a telnet server and perform a myriad of actions. To demonstrate this, you can record traffic as you go to a site like the one chosen for this example, the Victoria Free-Net of British Columbia, Canada.

After downloading the JAR and JAD files from muTelnet, take a moment to familiarize yourself with the application before recording. To start the application, click the button under Launch with the muTelnet entry selected in the list. When the application begins, after clicking the green answer button, you can access the menu at any time by clicking the top center gray button. First, start a timer called "connect" to record the connection to the site. Choose Connect from the menu, enter the address of the system under test (victoria.tc.ca), and click OK. After the connection has been made, the welcome screen will scroll past and you can stop the first timer. Then begin a second timer called "login" and type "guest" as a username to begin the login process. To submit this entry, choose Input > Character from the menu and select ASCII. At this screen, enter the numeric value "10," which corresponds to the

line feed character.

For the purpose of this example, that's enough traffic to demonstrate what network traffic would be generated from a phone. Stop the "login" timer, disconnect, and stop recording. This should generate the script you see in Listing 4. (Note that we have no affiliation with this site or anyone using it, so please don't run a multiuser suite against it. The Canadians might not be happy with you.)

```
/*   ->-> Session File Information <-<-
    Created: Fri Mar 29 13:19:33 2002
     Name: C:\DefaultTestScriptDatastore\TMS_Sessions\Telnet.wch
     Type: CS-Network*/
#include <VU.h>{
push Timeout_scale = 200; /* Set timeouts to 200% of maximum response time */
push Think_def = "LR";
Min_tmout = 120000;    /* Set minimum Timeout_val to 2 minutes     */
push Timeout_val = Min_tmout;
push Think_cpu_threshold = 150;

vtn1_victoria_tc_ca = sock_connect("telnet001", "vtn1.victoria.tc.ca:23");
{ INFO SERVER "vtn1.victoria.tc.ca"="199.60.222.3"; } /*1*/
set Server_connection = vtn1_victoria_tc_ca;
push Think_avg = 0;       /* 0 < 150 (cpu processing delay) */

start_time ["connect"];
sock_send "`fffd01`";
sock_nrecv ["telnet002"] 15;
sock_send "`ff`";
sock_nrecv ["telnet003"] 3;
set Think_avg = 0;       /* 0 < 150 (cpu processing delay) */
sock_send "`fb18fffb1ffffa1f00150006fff0fffc23fffc27fffc`$";
sock_nrecv ["telnet004"] 15;
sock_send "`fffd01fffa1800`ansi`fff0`";
sock_nrecv ["telnet005"] 1268;
stop_time ["connect"];

start_time ["login"];
set Think_avg = 621;      /* 621 >= 150 (user think time delay) */
sock_send "guest\n";
sock_nrecv ["telnet006"] 60;
set Think_avg = 341;      /* 341 >= 150 (user think time delay) */
sock_send "`fffe03fffc01`";
sock_recv ["telnet007"] "$"; /* 504 bytes */
stop_time ["login"];

sock_disconnect(vtn1_victoria_tc_ca);
pop Think_cpu_threshold;
pop [Think_def, Think_avg, Timeout_val, Timeout_scale];}
```

Listing 4: Script recorded while accessing the Victoria Free-Net site from the emulator

As you can see, the traffic includes small amounts of readable text, such as the username, and a great number of hexadecimal characters. To understand the protocol, the best choice is to work with a developer who can explain the data. Nevertheless, with one datapool entry for the username, the recorded script in Listing 4 could easily be used to test the connectivity and logon scalability of this telnet server.

Summary

Using emulators to automate testing is a viable method to reduce overall testing time and gather data not easily obtainable through traditional, nonautomated methods. As long as the limitations of this method are understood, significant benefits can be achieved by using Rational TestStudio to automate both functional and performance tests against wireless emulators.

Acknowledgments

The original version of this article was written on commission for IBM Rational and can be found on the IBM DeveloperWorks web site

About the Authors

Scott Barber is the CTO of PerfTestPlus (www.PerfTestPlus.com) and Co-Founder of the Workshop on Performance and Reliability (WOPR – www.performance-workshop.org). Scott's particular specialties are testing and analyzing performance for complex systems, developing customized testing methodologies, testing embedded systems, testing biometric identification and security systems, group facilitation and authoring instructional or educational materials. In recognition of his standing as a thought leading performance tester, Scott was invited to be a monthly columnist for Software Test and Performance Magazine in addition to his regular contributions to this and other top software testing print and on-line publications, is regularly invited to participate in industry advancing professional workshops and to present at a wide variety of software development and testing venues. His presentations are well received by industry and academic conferences, college classes, local user groups and individual corporations. Scott is active in his personal mission of improving the state of performance testing across the industry by collaborating with other industry authors, thought leaders and expert practitioners as well as volunteering his time to establish and grow industry organizations. His tireless dedication to the advancement of software testing in general and specifically performance testing is often referred to as a hobby in addition to a job due to the enjoyment he gains from his efforts.

Chris Walters is a recognized innovator in the areas of automated testing and wireless applications. His background is in software and network architecture, security engineering, database design and administration, programming, and management, and he’s had years of experience in performance engineering.

About PerfTestPlus

PerfTestPlus was founded on the concept of making software testing industry expertise and thought-leadership available to organizations, large and small, who want to push their testing beyond "state-of-the-practice" to "state-of-the-art." Our founders are dedicated to delivering expert level software-testing-related services in a manner that is both ethical and cost-effective. PerfTestPlus enables individual experts to deliver expert-level services to clients who value true expertise. Rather than trying to find individuals to fit some pre-determined expertise or service offering, PerfTestPlus builds its services around the expertise of its employees. What this means to you is that when you hire an

analyst, trainer, mentor or consultant through PerfTestPlus, what you get is someone who is passionate about what you have hired them to do, someone who considers that task to be their specialty, someone who is willing to stake their personal reputation on the quality of their work - not just the reputation of a distant and "faceless" company.