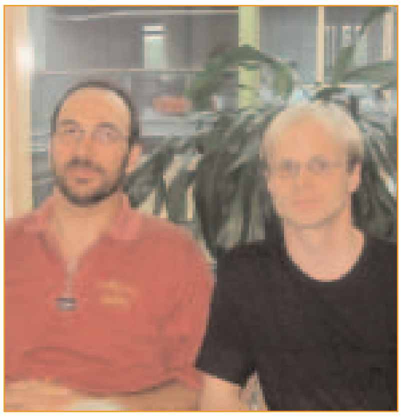

Perl wisdom

When Sauli Karhu (right of picture) and Eero Kareoja found that they needed more features in their test automation software, and more powerful hardware to run it, they built and used their own tools instead. They tell us why and how

It's year 2000. Contracts are dropping in constantly: our company is expected to launch mobile games through many telecom operators world-wide. Very soon our game services are to be launched with heavy marketing in many countries. Two testers are sitting and pondering over the situation: The development team is doing hard work on the software platform that will be the basis for these services. The question is: How should we ensure the quality of all this?

In this case we chose an untypical approach, at least when it comes to testing tools: we needed heavy test automation, but didn't go for commercial testing tools. We did everything ourselves, with Perl. This article tells why and how.

The system under test, the game platform, is a kind of an application server supporting all major interfaces used in the mobile environment: SMS, WAP, i-mode, HDML etc. The presentation layer has been isolated from the core system. As the core system has an HTTP interface, we can consider it as an 'extended web server', and it can be tested like any web server. Testing of the presentation layer is not discussed in this article, so the methodology explained can be applied to any web server.

Choosing test tool and strategy

There were no written requirements for testing. The implied requirement was anyway something like this: "In a very short time, with very limited resources, testing should verify the reliability, performance and functionality of the game services". As you can see, the starting point for testing could be described as "about normal".

Focus on right things

Because of the limited time and resources we had to strictly prioritize our targets. We did this by starting from a high level perspective: asking what are the 1) most critical, 2) important and 3) good to have features of our services as experienced by our customers? Then, by estimating the resources needed for creating a required test system and the time to execute the tests, plus some other weight factors, we gradually came up to a matrix that we could use as a basis for planning and distributing our working time to different tasks. It turned out that the availability and performance of the system should be the first and most critical issue in our work.

By that time, almost three years ago, we were using a commercial and well-known test tool. Very soon we were facing restrictions set by our testing environment. It became clear that either we should make large investments both on the hardware and also on the add-on features of the test tool, or we should find some totally different approach to the problem. Before deciding on this, we decided to carry out some experiments with Perl.

Why Perl?

Selecting Perl for the purpose was fairly easy for the following reasons:

- Perl is a general-purpose scripting language, with very large number of add-on features. in particular, the LWP module, for handling HTTP-requests, played an important role in our case
- Perl is an excellent tool for parsing large text files, mainly thanks to its support for regular expressions. This feature is important, as very large log files must be analyzed after lengthy test runs
- it's a portable language; that is, if you write a script in eg Windows, you can run it in Linux and vice versa
- the language syntax is quite free: A Perl program might look like a simple shell script, a complex C/C++ program, or a mix of both. This makes Perl appealing to many kinds of programmers
- it's an intuitive script language, which makes it easy to learn (except the regular expression concept, which for beginners needs some time to digest). Also, for more experienced programmers, "the first guess often works", which makes programming very productive.

Graphical interfaces can be created with Tk. Initially this wasn't a big issue for us. However, when we at a later stage wished to separate the testing tool developers and the test team, the importance of an intuitive and user friendly interface grew significantly.

Perl is free and can be downloaded from the internet. The Perl homepage at **www.perl.org** is a good starting point. It's also an excellent documentation source. Windows users can download Perl from various sources such as **www.activestate.com**.

Testing in Windows and Linux

When testing with the same scripts on Windows and Linux, it very soon becomes obvious that Perl runs faster in Linux than in Windows. With the same hardware, the generated load was about doubled when using Linux. This applies especially for time-consuming file processing and other CPU-intensive operations. Windows has its strengths also, especially if you are used to the Windows environment. We noted these advantages with ActivePerl, compared to Perl in Linux:

- The text editors are superior in Windows. This applies especially to users who aren't very familiar with Linux and its editors
- ActivePerl's OLE/COM interface enables integration with other Windows applications. However Perl in Linux is also able to deal with Excel files
- The installation of external modules is easier in Windows than in Linux (except for Linux hackers)

The biggest disadvantages with Windows are a) the performance and b) the process model. In Linux you can easily create a large number of parallel processes running test scripts, eg for load testing. This is not possible in Windows; at least it's not this simple. Adding to that the performance difference between Windows and Linux, we didn't even try to create a load test system that works in Windows.

To sum up, we used Linux for running the load test scripts. Function test scripts were run conveniently in Windows; also log processing scripts were occasionally run in Windows, to get longer coffee breaks.

Load simulation - the first attempts

Immediately when there was something to test, we went there with Perl and created a simple loop to call the URL repeatedly, as

A simple load generator

Here is a very simple script that generates HTTP request load toward a chosen HTTP server for one minute. Be careful when chosing the URL! Do not run it on someone else's site - they will complain to your ISP. Save the Perl example below as loadGenerator.pl and start it like this:

```
C:\>perl loadGenerator.pl http://localhost/index.html
loadtest.log
```

Wait for one minute for the script to finish. Look at the results in loadtest.log.

```
#!/usr/bin/perl

# This makes this program run for one minute. Could
# be provided also as a command line argument
$runTime = 1;

# Here we call the Time::HiRes module that provides
# sub-second time measurements
use Time::HiRes qw(gettimeofday);

# Here we initialize the functionality for HTTP
# requests and responses. You don't have to
# understand the details of this, it works anyway
use LWP;
require HTTP::Request;
require HTTP::Response;
$ua = LWP::UserAgent->new;

# Take in the command line parameters used when
# calling this script
$url     = $ARGV[0];
$logFile = $ARGV[1];

# Very simple error checking: Check that the input
# values aren't empty. If problems found, show the
# user how to use the script and quit the execution.
if ( $url eq "" || $logFile eq "" ) {
        die "\nUsage: $0 <url> <log file>\n";
}

# Open a file for writing into the current directory
open LOG, ">$logFile" or die "Cannot open log file:
$logFile\n";

print “Printing to $logFile...\n”;

# Get time of this machine before testing starts:
$startTime  = time();

# Calculation of the time when the test should end:
$endTime    = $startTime + (60 * $runTime);

# The $now variable is used for getting the current
# time during the test session
$now = time();

# These variables get the high-resolution timestamps
# before and after the HTTP requests
$hiResTime1 = 0;
$hiResTime2 = 0;

# Needed in the calculation inside the loop below
$elapsed = 0;

# This counter is added to the log file for tracking
$counter = 0;

# Write the first line in the log file:
print LOG "Measuring response times for URL:
$url\n\n";
# This loop runs until end of time is reached
while ( $now < $endTime ) {

        # Steps the request counter:
        $counter++;

        # Get the high resolution time
        # of the server, store it for a while…
        $hiResTime1 = gettimeofday;

        # Here the HTTP request is made
        $request = HTTP::Request->new(GET => $url);

        # Getting the HTTP response object
        $response = $ua->request($request);

        # If the HTTP server responded correctly,
        # get another high resolution time stamp,
        # calculate the response time
        # and write it to the log
        if ( $response->is_success ) {
                $hiResTime2 = gettimeofday;
        $elapsed = $hiResTime2 - $hiResTime1;
                print LOG "$counter\t$elapsed\n";
        }
        else {
                print LOG "ERROR: no_success\n";
                die "Request nr $counter failed\n";
        }
        # What is the time now? (Needed in the next
        # iteration of this loop)
        $now = time();
}
# Adding some analysis to the end of the log file
$average = $counter / ($now - $startTime);
print LOG "\nAverage: $average requests/second\n";
```

rapidly as possible, for a given time period. The script for doing this is shown above. It also logs the HTTP response times in a text file.

We started varying this simple script. The first finding was: in the case of a system taking HTTP parameters, it's very simple to generate the parameter values on the fly. Here are a few things that were done:

Session testing: Loop a URL that starts a session; modify the URL before the request, for example using a counter, so that a new session is started for each request.

Database testing: A loop submits HTML forms into a database. If the data is modified, for example with a counter before each request, we get unique data items into the database.

Combinations, using multiple simultaneous tests: eg what happens to the response times of certain requests under different load conditions (requests/minute + large number of open sessions)?

Trying to really get to the limits: A script that starts many simultaneous load generators. Each reports the responses and response times to a file. A separate script processes the files and creates a report.

A master script to control a large number of other scripts: For example a test lasting several days with pre-defined and varying load levels.

With this approach we could live for a while. Since we were able to test almost anything without wasting time on preparations and planning (!) many very critical server problems were found and solved long before the releases. Note that we didn't even try to create very realistic usage scenarios at this time, it was simply out of focus.

Also later, after having developed a more sophisticated test system, we often went back to this simple load generator script, when there was an early build of new system to be tested. A copy of the file, some modifications here and there and voilá, in a few minutes we are stressing the system with heavy request traffic! After a few months we had a folder with myriad of customer- project- and whatever-case specific scripts. It's very important to note that we are doing focus-oriented testing, not software architecture development.

Load simulation – Improved

The requirements on load, performance, stress and reliability testing grew rapidly. The next natural step was to develop the basic script to behave like a user in a typical usage scenario: to let it log in and do some actions, some pre-programmed and some 'intelligent', based on responses from the SUT. For added reality, the time delays between the requests were randomized between two defined values.

So, one 'user script' simulates one user. A master script was created to start the user scripts. We noted that the number of user scripts running in one machine is limited; in our case it was mainly the RAM about that limited the number of user scripts. The testing machine must not start virtual memory managing, ie using a swapfile. Using 350MHz Pentium IIs with 256MB RAM and running Linux, we found the optimal number of simultaneous user scripts was around 50 per machine.

We went on to write a simple script which runs on the SUT machine and communicates with the test scripts, running on the testing machine(s), using Telnet - ie the script is a simple Telnet server; we named it *serverStatus.pl*. It can also be described as an 'agent'. The test scripts query this server and it returns information about the status of the load status of the SUT hardware. Since our SUT was running under Linux, the script finds most of its information in the /proc directory; however there are many ways to achieve the same thing under Windows.

Finally, the master script was developed to keep running while the user scripts do their work. The master script keeps polling serverStatus.pl about SUT load status and the user scripts communicate with the master script through telnet. The logging is improved with the Sys::Syslog module, enabling a very nice feature: all scripts log into one file. Afterwards, a separate reporting script processes the data in the file and writes a performance report file in Microsoft Excel format. It's especially important not to try to do too much analysis during testing where performance is being measured; it's better to dump all the information into a log file and process it later.

Conclusions

After almost three years we can see that:

- although our quality goal was pragmatically set to 'good enough', we actually achieved more than the required quality for this business.
- our Perl-based test system adapts flexibly to changes in the SUT.
- testing with Perl is efficient: We are able to execute the first load test of a new SUT in matter of minutes. More sophisticated tests take a few hours.
- since Perl is easy to install, we could very easily arrange big test sessions by harnessing any available computer to take part in the testing.

Many scripts originally developed for testing have passed the organization borders. For example some of our log processing scripts are needed daily in the operation of the production services, for trouble analysis and statistics.

Last, but not least: The development of this test system initiated a significant competence boost in every tester involved: Instead of learning a proprietary test script language, they became masters of a common and very useful programming language.

This approach to testing does not necessarily fit all organizations and projects. Our system was built for a SUT which was not too complex, and to cope with many parallel, very turbulent projects. Of course, we also had programming skills already in the team; however, as others have pointed out, all test automation projects - including those using commercial tools - need these skills to be successful.

Next issue: functional testing with Perl PT

It's FREE to attend the UK's premier exhibition for testers

QBIT LIMITED

TestExpo™

• demonstrations • presentations • keynotes
• meet the top companies in testing under one roof

London • 18th Sept 2003

Free tickets: www.testexpo.co.uk

Please email exhibit@testexpo.co.uk for an exhibitor's prospectus

Including the QBIT BREAKFAST FOR SENIOR MANAGERS: join a select panel of business and technical experts to discuss today's issues in testing and quality. By invitation only: tel +44 (0)1702 290558 or email breakfast@testexpo.co.uk to attend