Introduction

Nowadays every big software project needs a powerful and reliable build system. Any down time on the production line is highly undesirable. That's why the build system itself should be tested carefully. Such testing is used inside Nokia's Maemo department [1]. This article describes the build system's architecture, scheme to test it, choices of frameworks and supplementary software, and problems and issues which we faced with.

Build System in a nutshell

The subject to test in our case is a build system which has been developed and used in Maemo department. See Illustration 1.
The build system is a cluster of independent builder boxes, a common Data Base (DB) and a bunch of the User Interfaces (UIs) such as Web, XMLRPC, and Email. One host usually takes care about multiple interfaces although they might be installed on different machines. In few words system could be imagined as data flow between builder boxes and user interfaces through DB.

The main logical portion inside the system is a user's Request which contains all necessary data to produce the build. The Requests are filed in a certain form through Web or Email UIs, or compiled as a procedure call via XMLRPC interface.

**Proposed scheme to test the engine**

To perform complex functional testing of the Build System we proposed the following scheme.

The test system creates specific Requests with certain input. In case we use the Web UI to access the build system, the input represents an entry point to a site (base URL), necessary information to authenticate, and data that should be specified in an HTML form as the Request parameters.

The expected results and real results are compared during the Request's life cycle. The page with the state of Request is refreshed several times and the request state field is checked. We recognize a set of request states that could be divided to two groups: Request is done (pending, failed, delayed), and Request is not done (new, current). The request states and flow diagram are depicted on Illustration 2. When the Request is done the test system compares the expected state and real state of the Request and raises an exception if a mismatch is detected. Otherwise the checking stage is continuing up to the point where all of the expected results have been got.
To avoid an endless loop the timeout of the run is specified per each test case. If the Request is not done within a given time, the test system breaks the checks and raises an exception.

We chose the Web UI to test through because our users usually do the build requests via such interface. In other words we are testing from the user's point of view.

**Chosen frameworks**

To create necessary testing environment we investigated current state of existent frameworks and tools. Additionally we must take into consideration following requirements:

- We use Python, so any framework proposed should either have Python binding or be written in Python itself
- ECMA/JavaScript interpreter is a must as the build system's Web UI contains lots of AJAX pages
- Web content on server side should not be modified
- OpenSource software is preffered

During investigation we decided to use the Selenium framework (Selenium remote control, in particular) [2], Iceweasel/Firefox (it is required by the Selenium framework), Python binding for Selenium, Debian 5.0 Lenny OS.

A Request in the build system could take a long time to run (on old fashion PC it approximately equals to 5 to 15 minutes on special test data). Due to this restriction we decided to do the test cases that are independent from each other. Thus, the entire work could be parallelized in time.
The testing frameworks for Python which run the test cases in parallel are nose [3] and TestOOB [4]. We chose the TestOOB as a simple multi-threaded wrapper of standard Python module unittest [5].

**Why Selenium?**

During the preparation of the test system we had checked the status of several frameworks, such as Twill, Selenium, Windmill, STIQ and others.

**How it works**

The following illustration shows the Selenium Remote Control workflow (see Illustration 3). The python driver sets up a network connection to the Selenium Remote Control server which is run as daemon. Any open request provokes the Selenium Remote Control server to start a browser window with certain configuration and plugins, one of them is the Selenium Core. In our case we apply specific template to the browser to authenticate properly via HTTPS.

*Illustration 3: Selenium Remote Control workflow*

The detailed description could be found in [2]. Supplementary documentation links are provided below in the Useful links subsection.

**Pros and cons**

We chose Selenium framework due to following reasons:
• Python is one possible language to write test cases
• It uses real browser that could run ECMA/JavaScript code on web pages
• Certain configuration of Selenium allows to not modify web content on server side
• It is a free, OpenSource tool

However, we have met few disadvantages of this framework:

• It is written in Java, thus it requires to have an Java installed, and any fix in Selenium itself will be done slower (we assume we don't know Java and have no resources to look into the code)

• Any extension should be written in JavaScript (although we have no need to extend the functionality, the possibility to do such thing is restricted)

• The packaging for Debian is absent

**Useful links**


• [http://www.advogato.org/article/874.html](http://www.advogato.org/article/874.html) – *Testing Python Web applications using twill and wsgi_intercept*

• [http://oreilly.com/catalog/9780596527808](http://oreilly.com/catalog/9780596527808) – *An Introduction to Testing Web Applications with twill and Selenium*

**Performance measurement**

The first version of the testing system runs test cases one-by-one. Due to high load on build farm the test cases took a long time (see Illustration 4). We decided to run them in parallel mode. Following chart shows how this change impact on performance (see Illustration 5).

A red line on both charts reflects the overall time to execute dependent to number of test cases. Meanwhile, a green line reflects time to execute test cases without testing system and Build System polling overhead.
The testing environment contains enough nodes in the Build System (more than we have test cases in our example), each node has several workers and common load balancer. We run 6 threads, and black vertical line on Illustration 5 defines this number.

The formula shows how we calculate time to execute of test cases in one-by-one mode

\[ t_{execute} = \sum_{i=1}^{n} (Tend_i - Tbegin_i), \]

where \( Tend \) is a timestamp when a Request is done and checked, and \( Tbegin \) is a timestamp when the system gets a Request in state current.
In case of parallel running we use another formula to calculate time to execute as follows:

\[ t_{\text{execute}} = T_{\text{end last}} - T_{\text{begin first}}, \]

where \( T_{\text{end last}} \) is a timestamp when last request in the bunch is done and checked, and \( T_{\text{begin first}} \) is a timestamp when the system gets first request from the bunch in state current.

As we can see the parallel mode keeps time to execute less than 40 seconds independently of the number of test cases. However, when queue is full the rest of test cases are waiting, and required time to do the job is higher. In case we run tests one-by-one the time to execute is past 50 seconds just for bunch contains 2 or more tests. In the result we have decreased a time required to run tests approximately in 5 times (green line, 6 tests). The real bunch of tests takes few hours to run if we consider one-by-one mode. That's why an introduction of parallel mode saves a big amount of time. Thus, on production line we can quickly deploy bug fixes.

**Problems and issues**

Besides described Selenium framework disadvantages the other issues we have to point to:

- There is no understandable error message in Python driver related to connectivity problems, e.g. network connection between test system and target host is broken, just timeout exception

- When running with a new host which requires certificate approval the firefox window is stuck, so, in this case the firefox template should be updated every time when new host is required to test

- Sometimes the network connection between Selenium server and Selenium driver is stalled, in the result Selenium server should be restarted

- Selenium log file is growing quickly when debug mode is on

**Summary**

Thus, using Selenium framework in conjunction with Iceweasel/Firefox, TestOOB framework and other accompanying OpenSource software we have built testing system which is used with AJAX-based web interface and which allows us to significantly reduce time to execute test cases.

All test cases reflect users input data and their expectations of the result. Our test data including test cases is used as a reference point to do performance testing as well.
References


