LUSCA – test framework for network applications

Jure Rastić, Stjepan Groš, and Vlado Glavinić
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing
Address: Unska 3, 10000 Zagreb, Croatia
Telephone: +385 1 6129-935 E-mail: jure.rastic@fer.hr

Summary – Testing of network applications is a very time consuming process. Existing frameworks used for testing tend to be very powerful but are complex to deploy. LUSCA was created in an attempt to simplify and automate the process of examining the behavior of network applications by primarily tackling remote computer control, test set up and execution. LUSCA also provides a convenient way to gather test results and store them for further analysis and documentation purposes. Test parameters are defined using a straightforward XML based language, created for this specific purpose. This language identifies; test objects, subjects and logic controlling the test flow in a simple way. In the following paper LUSCA is introduced as a test framework designed for network applications; the idea behind it, its architecture elements and implementation approach.

I. INTRODUCTION

To create a network application that is safe, stable and conforms to the protocol, one must go through repeatable process of implementing and testing its design. To test a network application, developer must create a testing environment under which application is tested. This usually requires setting up some sort of computer network topology, installing and configuring appropriate software components on computers, deploying developed application and support files, starting applications, monitoring results of execution, gathering results for analysis and then analyzing program outputs.

Creating set of scripts for testing is a programming task itself that is usually appointed to application tester. We are aiming to avoid wasting unnecessarily long periods of time in scenarios which require complex network topologies, lots of networked computers running multiple applications all parallel monitored etc.

The objective is to flip the unfavorable ratio between the longer time needed for preparation of the environment and gathering of the results [1]; and the relatively short time required for a test run; to the advantage of the developer. Saving time and effort is the primary goal of the framework here presented.

The paper is broken in sections as follows. In the following section we review, in short, the two existing test frameworks. After that, in third section we explain LUSCA as testing framework – how the test definition works, framework architecture and implementation. Fourth section shows a simple example of creating test for LUSCA that provides an insight to test design. Fifth section holds the conclusion of paper with thoughts on when LUSCA should be used and it's final purpose.

II. TEST FRAMEWORKS

Test frameworks provide reusable and extensible mechanisms for managing of test environments, creating of test packages with a set of test cases, scheduling the test execution etc. To test an application, the tester can approach the process by treating the application as either black box [2] or white box [3]. During black-box approach, only externally observable behavior of the application is monitored; i.e. the information in the created files and logs, what type of network traffic is generated and how the application changes the environment in which it executes. During white-box approach, internal state of the application is observed. Therefore, the testing is done by, for example, looking at the internal state of the application and the values used during the test run, including which functions are called in which order.

Hence, based on those two approaches, test frameworks can be divided into two basic groups; frameworks that provide means for black-box testing and frameworks for white-box testing. Black-box frameworks provide the tester with tools that aid monitoring of the external application behavior and the test flow control. White-box frameworks give the tester a simplified definition of the targeted internal states of an application (or their parts) and the ability to verify the results.

Common practice is to do white-box testing (manually, or through unit tests) in the early stages of application development and to introduce black-box testing during the later stages [4].

We used Linux operating system and open source software as a building platform for LUSCA. There are some disadvantages of open source software such as the lack of quality or fitness guaranties and support issues that we took in consideration and we will compare LUSCA with two commonly used open source frameworks for testing.

In following subsections, these two test frameworks will be reviewed in short; JUnit [5], used for white-box testing, a well known framework for a JAVA programming language, and STAF framework [6] as a toolkit for black-box testing.

A. JUnit

JUnit is a regression testing framework used to implement unit tests in Java. Unit testing is performed to check that the individual modules or units of source code are working properly. A unit is a smallest testable part of an application. In a procedural design, a unit might be an individual program, function, web page etc. But in object oriented design, the smallest unit is always a class.

JUnit provides assertions for testing the expected results, test fixtures for sharing common test data and test runners for execution.

Creating a test comes down to writing a class with public method which is annotated with *@Test* string that asserts expected results on the object under test. To test if object's method returns *boolean* true value for example, user calls *assertTrue* method of JUnit's *Assert* class to perform validation of method's functionality.

To run the test, user executes *JunitCore* from *org.junit.runner* package with a parameter of class being tested. It runs all methods annotated as test methods and prints test results.

B. STAF

STAF is an open source framework designed to provide services for process invocation, resource management, logging, monitoring etc. It enables the user to focus on building high level automation solutions for testing.

STAF runs as a daemon process on each computer. A set of computers running STAF daemon is referred to as the STAF environment and it has peer-to-peer structure. STAF services are used through STAF libraries inside the test applications or wrapper scripts as tools which provide the required functionality for the test. Libraries are available for various programming languages such as C, Java and Perl.

Test definition is set by writing a group of programs or scripts which utilize STAF services such as LOG service for logging, FS service for file transfers, VAR service for storing variable values etc., all in a required order, to create a test case.

III. LUSCA TEST FRAMEWORK

LUSCA is an open source framework designed for testing of network applications on the Linux platform. Two main goals are:

– to simplify test definition and hold it in *one* single file
– to provide a service for test execution.

Test definition is set through a XML based programming language which holds all the information about files and computers required for the test. A package is created from the test definition; it holds XML itself, files, scripts etc. Tester uploads the test on the service and starts it. When the test is done, the tester can easily review the results that service provides.

LUSCA development started with defining what could be done with it in a simplest possible way while taking into consideration the methodology of manual automation in testing. As mentioned in the introduction, preparing a test comes down to creating scripts for deploying files on hosts, starting applications, detection of a relevant event inside of an application on a host,scripts for communication used to synchronize events between two or more hosts and for gathering results.

LUSCA provides means for tester to create just scripts for starting applications and parsing application events. The rest is done by describing file deployment and actions for certain event in testing environment.

In this first subsection coming up, we will illustrate the XML format used for setting up the test package; which files and scripts are required, how many computers are used in the test and the flow of the test. In the second subsection that follows, we will explain LUSCA's architecture and its components. Finally, in the third subsection we will relate how LUSCA was implemented.

A. Idea transformed to XML

Test definition contains three basic parts: test preamble, definition of resources (scripts, computers, etc.), test case flow definition.

Test preamble: details basic information needed for the test package documentation. It holds the name of the test author, date of creation and the textual description of the test scenario.

Definitions: entails information for the framework relating to files, scripts, tested applications and computers required for the test.

For *scripts* and *files*, user defines sources to fetch them if they are; remote (for example on HTTP or FTP server) or; local (inside the test package) and destination path on host computer. Script definitions contains additional information about it's input/output parameters and interpreter for script execution. Script execution will be expanded on further, in the paragraph *Execution* element.

Variables are used for storing values required for some script to run or result of scripts function. Variables declared here (within *Definitions* element) are commonly used for storing values required through all the test cases (additional variables can be declared for each test case). Declaring a variable requires setting it's default value.

User can define one or more *applications* which are to participate in the test. For every application, the user defines information needed for the test documentation, such as the short description, web page and the author list. To provide a source code for the framework, the user can set a local or a remote path to the application package. Additionally, a source code can be fetched from version control services such as SVN or CVS.

Computer definition is used for describing computers needed for test execution. For every network interface on every computer taking part in the test, the user defines individual IP address, netmask etc. Framework will set these computers up during the preparation phase of the test run.

Test cases: test package contains one or more test cases defined here following the common approach to divide whole test in set of test cases. User can define dependencies between test cases, so if the test case *A* depends upon the test case *B* and *B* fails, *A* will not execute. This way the user can optimize the time of the test package execution. Every test case definition contains three sub-elements, which describe a corresponding test phase: preparation, execution and conclusion. Preparation phase is used to perform additional set-up of computers and deployment of files. During execution phase main test execution is performed. Conclusion phase is used to sum test results or clean up test environment if needed. Diagram of test case element is shown in figure 1.

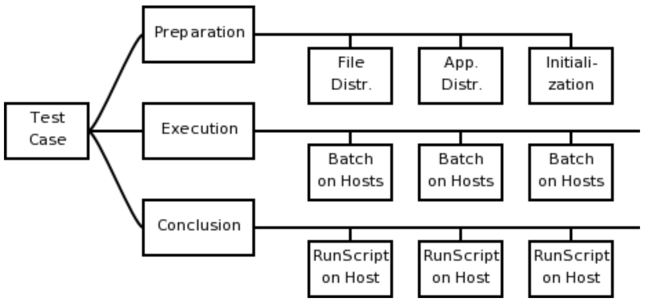


FIGURE 1: TEST CASE ELEMENT DIAGRAM

Preparation element defines on which host to deploy previously defined files and applications (scripts are automatically deployed based on script execution on a host). Also, the user can define execution of scripts on a host at this point to additionally configure the host environment.

Execution element defines batches (*Batch* sub-elements) that are executed on a set of hosts and also the signal which states that the test has passed. Each batch starts when a host receives a signal on which the batch is triggered (signal creation is explained later in the text). For every batch, the user can define additional variables that exist during execution of the test case. A batch is defined with a *Command* element that has a corresponding list of *RunScript* (i) and *ResultFile* (ii) sub-elements. These two elements are the *heart and soul* of the test flow definition in LUSCA.

(i) RunScript element defines which script is to run with which input variables and where the script output values should be stored. A simplified diagram of the *RunScript* element is shown in figure 2.

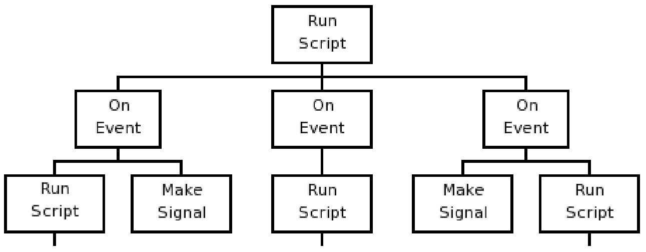


FIGURE 2: RUNSCRIPT ELEMENT DIAGRAM

Scripts can be used for various tasks; application execution, calculation, file parsing etc. During a script execution, the framework parses the script standard output and changes the variable value when a predefined value output line is received. This is the way the script communicates with the framework. To set a value of a variable used as storage for the script output, the script generates a line with an index of a defined output variable and it's new value. To inform the framework of an event which happened in the environment, the script generates an event line with a corresponding event name and additional information which can later be used for test analysis. The user can define a set of actions for an expected event in the *Actions* sub-element. To take appropriate action on the host computer triggered by an expected event, the user defines another *RunScript* element, the *OnEvent* sub-element, therefore creating a nested script execution. This script can also have a defined set of actions for events it generates. This way, a single event can trigger a whole tree of scripts to be executed. With this technique, the framework provides a simple tool to achieve almost any required task on one host.

To take action on a different host within the environment, the used defines a *MakeSignal* action for an expected event. When the script generates a corresponding event, the framework then generates a signal which is sent to another host. Type of signal can be *info*, *error* or *fatal*. Info signal is a normal signal, used for notifying remote host. Error type of signal is used to state that something went wrong with the test but it doesn't need special handling (is used for documenting purposes). Fatal type of signal fails the test; the framework stops test execution on this signal and documents the fact that the test has failed.

This technique, combined with use of variables enables easy communication between different hosts within the test environment. Additional test flow control can be achieved through a *ResultFiles* element. Simplified diagram of this element is shown in figure 3.

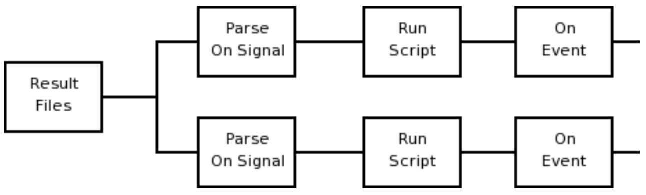


FIGURE 3: RESULT FILES ELEMENT DIAGRAM

(ii) ResultFiles element with it's Parse sub-element is used to simplify the process of parsing a file at a certain point of test execution (for example; downloaded data or log file). The task of parsing the result file is also given to the script. The parsing starts when the host receive the expected signal (for example; a signal sent when the download of data is finished). Multiple scripts can be defined to parse a file on receiving the same signal. Script that is parsing a file is also able to generate events or set a variable value.

Conclusion element is used to act on the specified host when the the test is done. Some of the frequently required actions are; to clean up files not required any more, to run an analysis script which on a set of files to create a conclusion about the test run (and store it in the summary file), or to restore a previous state which has been changed during test execution.

Presented introduction to LUSCA programming language for defining a test package is a powerful tool in the hands of an application tester. It provides simple ways to achieve interaction between the hosted application and the framework (through events) and between two or more hosts (through signals). The tester can still use his favorite scripting language for operations. Flow is controlled naturally through an event driven system. When the user defines the test, he packs it with all required files in one package to be used for test execution. The package is then uploaded to LUSCA service which in turn creates the required test environment and executes the test.

B. Architecture

Architecture of LUSCA is a centralized one with main process as a center role of system and facility for test storage. This process communicates with user through it's console, that way providing a service for test execution. The purpose of second process is to execute test batches by taking role of one host in test. Architecture of LUSCA is shown in figure 4.

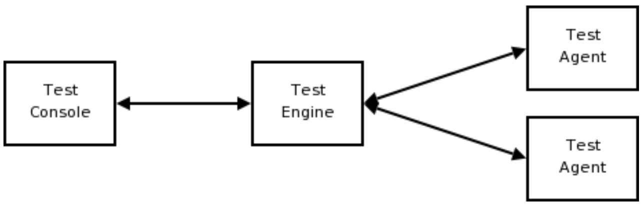


FIGURE 4: LUSCA ARCHITECTURE

Role of every service shown in architecture figure is described further in text.

Test agent: the role of host described in test. It executes preparation scripts, executes one or more test *Batch* elements as defined in test description and ends test with the execution of conclusion scripts. During test execution agent generates signals which are sent to test engine. Agent also receives signals from engine. There is no communication between agents in environment – all communication is done through test engine. When agent receives a signal it starts batches that should be triggered on this signal and/or starts parsing script triggered on this signal. On end of test execution test agent packs all result files and event log in one file that is then stored on engine service.

Test engine: Role of engine process is to coordinate tasks of agent services and to provide a console service to user. The user only sees the engine service through its console interface; and cannot see the agents *behind* it. During test execution engine dispatches signals received from one agent to all agents that have action on this signal. When the test is done the engine receives all result packages from agents that acted as hosts for the test, and stores them. The user can fetch these files whenever needed, through the test console.

Test console: is used as interface for uploading the test to LUSCA service and retrieving results. User can see the history of the test run and get the results of the test. When a test fails, the tester can forward the logs created by the test run, to the developing team for information. After they correct the bug, the tester can start the execution of the whole test package or just one test case through the console.

C. Design and implementation

After defining the test description and the architecture of LUSCA, the next step was to implement the required structure elements. Python [7] was the chosen programming language for implementation, because of its clear syntax rules, ease of use and efficiency in acquiring the needed functionality.

It was decided that the underlying protocol for communication within the framework will be SOAP. Functionalities of the engine and the agent are implemented as SOAP services. Test engine is a SOAP service which provides remote methods needed for the test console and the test agents. Test agent is a SOAP service that provides remote methods needed for the engine to establish the required communication; signal passing, variable retrieval etc. The test console is currently available through the web interface of the test engine. Additional clients can connect to the engine's console SOAP service. HTTP protocol is used for transfer of files between the engine and the agents to avoid size and time overhead of transforming the file into a required format for SOAP. Services in the framework are shown in figure 5.

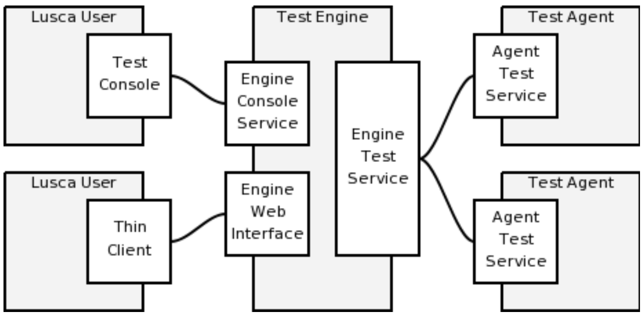


FIGURE 5: LUSCA SERVICES

The test engine supports running of multiple tests at the same time. Number of concurrent tests which can be run depends on the number of available agents logged in at that time. If a test requires more agents than available, the engine cannot start the test and so it informs the user. One agent can participate in one test at one time.

To implement a set of services that conforms to the ideas presented, one would need to create a complicated system of threads or processes for serving requests in an asynchronous way. The code would be very hard to read without a bigger picture of the whole system in mind so we started to search for some kind of a programming tool or library that would make the developing process as easy as possible. After researching for a time, we stumbled upon Python’s Twisted library [8] – an event driven networking framework written in Python and licensed under a MIT license. Reading through the documentation, ideas came up as to how to employ it in LUSCA. Using its *reactor* module, *Deffered* objects for scheduling processing tasks, *web* module for designing SOAP services – one can create a complex asynchronous server and client application that is fast, robust and secure, if needed. The task of programming seemed a lot simpler with such a powerful tool available.

We used *SOAPublisher* class to create LUSCA services by grouping functionalities into service paths – HTTP addresses. For engine and agent service there is a *SystemProtocol* and *TestProtocol* with its children paths for every running test package. *SystemProtocol* path is used to gather all methods that are required for internal communication between the engine and the agents. When an agent starts, it first logs in to the engine service and provides it with information about itself – the agent. The engine stores info for the agent in the available agents list. *TestProtocol* path is used for communication during the test run. Engine's *TestProtocol* creates additional, children paths for every test package that is being run. Agent's *TestProtocol* does the same. Every message passed regarding the test package execution is communicated via these paths.

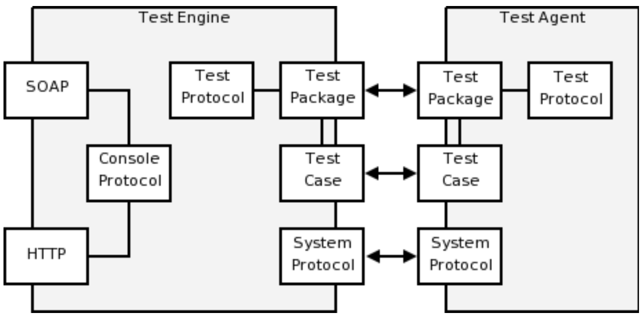


FIGURE 6: LUSCA SERVICE PATHS

Figure 6 shows paths created during test execution of one package with one test case. For example, when an agent generates a signal (as resulting action for event), it calls for remote method *rcv_signal* on the path that holds this test case method on the engine. The engine then starts a function which searches in *signal-listening host's* hash table for hosts using this signal as a trigger. The engine calls remote method *rcv_dignal,* on paths created for each host needing this signal. Receiving host starts its function which searches *signal-required action* hash table and executes triggered actions.

IV. USAGE SCENARIO

We will describe a simple test illustrating how a LUSCA user should approach the test design and creation. For test scenario, we will use a HTTP server application developed by the user. While adding new features, the developer needs to know that they don't break the basic functionality already tested, so he takes the option of automating the test. He uses *wget* as a HTTP client to download served files. To test the robustness of his web server at one point, he needs to test it by creating 1000 of GET requests for served file from 10 computers lying around him in the lab. He wants to parse HTTP traffic on every host that runs *wget*.

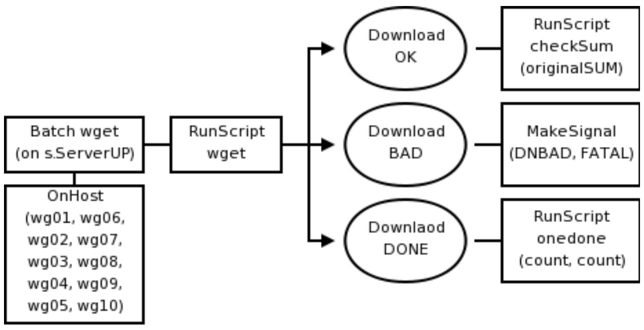
A. Manual testing

Developer creates shell scripts for parsing HTTP traffic using *tcpdump* tool, script for parsing downloaded file on the client side, using the file's MD5 sum and simple scripts for starting web server and *wget*. To run the test manually, he needs to set up his testing computers, fetch source from a SVN repository to server's host computer and execute the starting scripts on the computers. He starts parsing script on every client's host and looks at script summary on his remote terminal. When the test ends, he collects the server's log file. Then he documents the test run – did it pass or not, the test start and end time, the fetched version of application and the description of test flow.

B. Testing with LUSCA

Test can be defined by creating three different jobs which need to be run on 11 hosts. In the *Definition* element, the user defines needed computers, previously described scripts and SVN repository for tested application. He creates one test case with *Batch* elements for every job.

First *Batch* defines the job for hosts running HTTP client task. Batch runs *wget* script on the signal from server computer, which states that server application is running. Batch is shown on figure 7.



Batch wget (on s.ServerUP)
OnHost (wg01, wg06, wg02, wg07, wg03, wg08, wg04, wg09, wg05, wg10)
RunScript wget
Download OK
Download BAD
Downlaod DONE
RunScript checkSum (originalSUM)
MakeSignal (DNBAD, FATAL)
RunScript onedone (count, count)

FIGURE 7: WGET BATCH

Script which executes *wget* can generate 3 events; *Download_OK*, *Download_BAD* and *Download_DONE*. First event states that one download of the file is complete and the file can be checked. Action is to run *checksum* script with parameter of file's original MD5 sum passed from the *originalSUM* variable. Second event is generated when a file download fails. Action for this event is to create *fatal* signal, which will result in a test fail. Third event states that the client downloaded the file 1000 times. Action is to run a script which increases a variable whose value represents how many computers finished their job. This script generates event *All_DONE* when the counter becomes 10. Action for this event is to generate a signal on which the test case passes (not shown in figure).

For server job, the user creates *Batch* that runs on a predefined signal to start the test from test engine (*TE.start* signal from test engine generated automatically when the test is started). It runs *startServer* script that can generate 2 events. First event states that the application seg-faulted. Action is to make a fatal signal that results in a failed test. Second event states that the server is ready for requests. Action is to create info signal *ServerUP* that triggers *wget* batches on their hosts. Server's *Batch* is shown in figure 8.

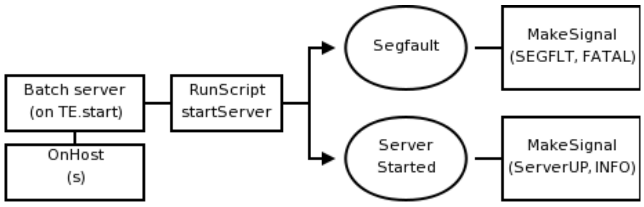

Batch server (on TE.start)
OnHost (s)
RunScript startServer
Segfault
Server Started
MakeSignal (SEGFLT, FATAL)
MakeSignal (ServerUP, INFO)

FIGURE 8: SERVER BATCH

Third batch is created to monitor HTTP traffic. It is also triggered on ServerUP signal and runs *trafficParse* script which then generates *Traffic_BAD* event. This event states that something went wrong in traffic between hosts. The resulting action is the creation of a signal which fails the test. Batch is shown in figure 9.

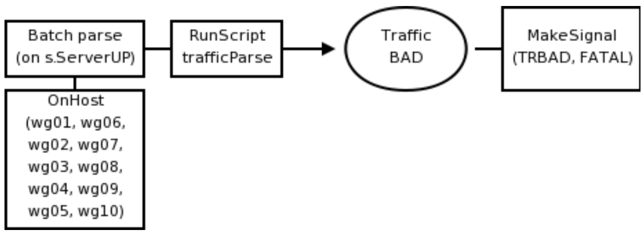

Batch parse (on s.ServerUP)
OnHost (wg01, wg06, wg02, wg07, wg03, wg08, wg04, wg09, wg05, wg10)
RunScript trafficParse
Traffic BAD
MakeSignal (TRBAD, FATAL)

FIGURE 9: PARSE BATCH

Here shown batches are simplified to provide some basic insight into test definition using LUSCA's language for test definition.

Test is executed in LUSCA environment with 11 agents – one as server and 10 as download hosts. User uploads the test on the engine, through the test console or web interface and starts it. After the test is done, the user fetches LUSCA log of events and signals generated during the test run, using the test console. Logs and test definition can be used to generate documentation about the test case.

V. CONCLUSION

We started from a basic idea that the test flow should be as easy as possible to define by a person who doesn't need to know any programming languages. LUSCA's creation was based on these simple ideas and should be used as a base tool for test execution.

Existing frameworks for testing provide powerful tools to perform testing of typical applications but tend to complicate test creation and deployment when it comes down to running a test executed on multiple computers. Set-up of computers is still done manually, the starting of a test is still done manually on every computer, gathering results in one place is still done manually.

We designed LUSCA to primarily be a *service*, rather then *framework* which helps a user to create and run the test by overcoming presented problems. Additional effort was made to provide a way to fetch files from remote computers. Support for source code version control is provided within the option to define SVN or CVS server as a source for application's code. It allows a user to create the test only once and run it against the latest version of code when it is needed.

LUSCA saves time by eliminating time needed for:

– setting up computer network with use of agents installed on remote computers who will take care of network setup as described in test definition,
– file and application deployment – files needed for test are deployed only *once* when user uploads the test package to engine service,
– test result gathering – result files, log of events and signals are gathered by LUSCA and stored on engine service for easier retrieval through console;

To save efforts needed for test flow design we created a simple language that contains all instructions needed for LUSCA to execute the test. Also, graphical tool for test design will be made to generate described XML.

User will need some time to get used to creating test flow definition which is controlled by events and signals as defined in this paper, but we find it more naturally then creating a procedural description of test.

During test design, user should take in consideration the way LUSCA was designed and implemented to run tests. Signals are sent and received through test engine, variables are stored and fetched from test engine – all communication is done through SAOP protocol which is relatively process/data-heavy for these tasks. LUSCA was designed to ease the pain at the stage of the job which *hurts* the most – in automation of network computer configuration, designing a test flow end executing it.

VI. ACKNOWLEDGMENT

This work has been carried out within projects 036-0361994-1995 Universal Middleware Platform for Intelligent e-Learning Systems funded by the Ministry of Science and Technology of the Republic of Croatia, and IKEv2 Step2 project funded by Siemens Networks.

LITERATURE

[1] Introduction to Software Testing
URL: http://www.onestoptesting.com/introduction/
[2] Nilesh Parekh, Software Testing - White Box Testing Strategy
URL: http://www.buzzle.com/editorials/4-10-2005-68350.asp
[3] Nilesh Parekh, Software Testing - Black Box Testing Strategy
URL: http://www.buzzle.com/editorials/4-10-2005-68349.asp
[4] Kerry Zallar, Practical Experience in Automated Testing
URL: http://www.methodsandtools.com/archive/archive.php?id=33
[5] JUnit, Testing Resources for Extreme Programming,
URL: http://www.junit.org/
[6] Software Testing Automation Framework (STAF),
URL: http://staf.sourceforge.net/
[7] Python Programming Language,
URL: http://www.python.org/
[8] Twisted – Trac,
URL: http://twistedmatrix.com/