BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

PERFORMANCE OPTIMIZATION OF TESTING AUTOMATION FRAMEWORK BASED ON BEAKERLIB
OPTIMALIZACE VÝKONU AUTOMATIZOVANÉ TESTOVACÍ PLATFORMY ZALOŽENÉ NA BEAKERLIBU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

JAKUB HEGER

SUPERVISOR
VEDOUCÍ PRÁCE

Mgr. Bc. HANA PLUHÁČKOVÁ

BRNO 2017

Brno University of Technology - Faculty of Information Technology

Department of Intelligent Systems   Academic year 2016/2017

Bachelor's Thesis Specification

For: **Heger Jakub**

Branch of study: Information Technology

Title: **Performance Optimization of Testing Automation Framework Based on Beakerlib**

Category: Software analysis and testing

Instructions for project work:

1. Study how BeakerLib (integration test library) works.
2. Analyze performance of BeakerLib, design the metric of performance which would be optimized and identify the functional areas of BeakerLib and chosen harness to optimize performance (based on architectural review of the system, code review, code performance analysis).
3. Prepare and describe test set and environment for performance measurement.
4. Perform initial base line measurements, select at least one optimization and implement this optimization, e.g., by modification of BeakerLib code.
5. Check implemented optimization and discuss results.

Basic references:
* according to the instruction of the supervisor

Detailed formal specifications can be found at http://www.fit.vutbr.cz/info/szz/

The Bachelor's Thesis must define its purpose, describe a current state of the art, introduce the theoretical and technical background relevant to the problems solved, and specify what parts have been used from earlier projects or have been taken over from other sources.

Each student will hand-in printed as well as electronic versions of the technical report, an electronic version of the complete program documentation, program source files, and a functional hardware prototype sample if desired. The information in electronic form will be stored on a standard non-rewritable medium (CD-R, DVD-R, etc.) in formats common at the FIT. In order to allow regular handling, the medium will be securely attached to the printed report.

Supervisor: **Pluháčková Hana, Mgr. Bc.,** DITS FIT BUT

Beginning of work: November 1, 2016

Date of delivery: May 17, 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních.technologií
Ústav Inteligentních systémů
612 66 Brno, Božetěchova 2

Petr Hanáček
Associate Professor and Head of Department

Abstract

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v anglickém jazyce.

Abstrakt

Do tohoto odstavce bude zapsán výtah (abstrakt) práce v českém (slovenském) jazyce.

Keywords

Sem budou zapsána jednotlivá klíčová slova v anglickém jazyce, oddělená čárkami.

Klíčová slova

Sem budou zapsána jednotlivá klíčová slova v českém (slovenském) jazyce, oddělená čárkami.

Reference

HEGER, Jakub. *Performance Optimization of Testing Automation Framework Based on Beakerlib*. Brno, 2017. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Pluháčková Hana.

Performance Optimization of Testing Automation Framework Based on Beakerlib

Declaration

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením paní Mgr. Bc. Hany Pluháčkové. Další informace mi poskytl Mgr. David Kutálek. ??? Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

<div align="right">

. . . .
Jakub Heger
May 18, 2017

</div>

Acknowledgements

V této sekci je možno uvést poděkování vedoucímu práce a těm, kteří poskytli odbornou pomoc (externí zadavatel, konzultant, apod.).

Contents

Chapter 1

Introduction

This thesis was written in collaboration with Red Hat software company and focuses on a performance optimization of Red Hat's BeakerLib library, particularly its Journal feature. BeakerLib library is an open source shell-level integration testing library written mostly in Bash with some functionality in Python, which provides many convenience functions to simplify writing integration and black-box tests and also automates parts of testing process. One of the key features, uniform logging mechanism called Journal, has been numerously reported by its users to perform poorly. On this basis thesis analyzes its functionality a proposes solutions to increase performance.

The thesis is structured in a following way: chapter 2 introduces projects relevant to BeakerLib and its testing environment. Chapter 3 explains more in-depth how BeakerLib works, with focus on its Journal feature and analysis of its performance.

In the chapter 4 possible optimizations are discussed and chapter 5 focuses on implementation of proposed solutions. The chapter 6 then describes how was performance measured and in what environment. Chapter 7 is dedicated to analyzing measured results of individual implemented optimizations.

Lastly chapter 8 sums up implemented solutions and considers possible future work on BeakerLib library.

Chapter 2

Relevant projects

This chapter describes BeakerLib and projects relevant to it. First of all brief summary of BeakerLib itself is presented. Next section is devoted to Beaker system and the last section focuses on test harnesses.

2.1   BeakerLib

BeakerLib is a Linux shell-level integration testing library, providing convenience functions which simplify writing, running and analysis of integration and blackbox tests[18]. It is developed and maintained by Red Hat and operates under GNU General Public License. Main features of BeakerLib include:

- Journal - Uniform logging mechanism (logs and results saved in flexible XML format, easy to compare results and generate reports),

- Phases - Logical grouping of test actions, clear separation of setup / test / cleanup,

- Asserts - Common checks affecting the overall results of individual phases (checking for `exit codes`, file existence and content...),

- Helpers - Convenience functions for common operations such as managing system services, backup and restore of files and more.

BeakerLib was originally developed as a part of Beaker package but since then branched out as its own project and now is independent on Beaker.

This thesis focuses on BeakerLib Journal feature and problem it causes with long tests. Which is in more detail described in chapter 3.

2.2   Beaker

Beaker is a full stack software and hardware integration testing system, with the ability to manage a globally distributed network of test labs[17]. It is Red Hat community project under GNU General Public License version 2 and is distributed in the form of RPM package.

Main functionality includes management of hardware inventory, on which Beaker can install wide variety of operating systems from Red Hat Linux family. Another notable part is Task library which contains rpm packages of individual tests which can be run on provided machines. Users then can specify which hardware they require with which OS

and tests they want to run on it through either command-line tools or web interface both of which are part of Beaker install package.

If Beaker meets given criteria in its inventory it installs Test harness to which it gives list of tests to be run. Test Harness install and executes them while continuously sending results back to Beaker where they are stored for specified period of time.

2.2.1 beaker-wizard

Beaker-wizard is a flexible, interactive command-line tool that is a part of Beaker RPM package. It automates creation of BeakerLib tests using predefined or user-defined templates to create all files that are needed to run BeakerLib test. It also offers integration with git and Bugzilla, which is a web-based bug tracking system.

For example user can use beaker-wizard with `-b` option which takes as an argument bug identifier from Bugzilla and beaker-wizard creates all the necessary files with properly set variables in them to connect that test to given bug using the information it found in the tracker.

All tests created for this thesis were generated by beaker-wizard.

2.3 Test Harness

Test harness is a software framework that automates test execution. It contains tests to be run, executes them and reports results. <expand>

Beaker's harnesses prepare provided machine for BeakerLib by setting environmental variables to proper values, and then consecutively execute each test, while continuously reporting back results. They are integral part of Beaker ecosystem, as they allow user to run long test sets, which would without harness require much of manual work.

2.3.1 Beah harness

Beah [16] is a default Beaker harness . <expand>

2.3.2 Restraint harness

Restraint [12] is an alternative Beaker harness which can, unlike Beah, run with Beaker or standalone without it. <expand>

2.4 Projects' relation

Relation between Beaker, Harness and BeakerLib is shown in figure 2.1. In this example user submits Beaker job containing three tests and hardware/software requirements for a machine the tests should run on. After Beaker reserves it, it installs operating system and Harness which then successively executes each test and uploads their results back to Beaker when user can access them.
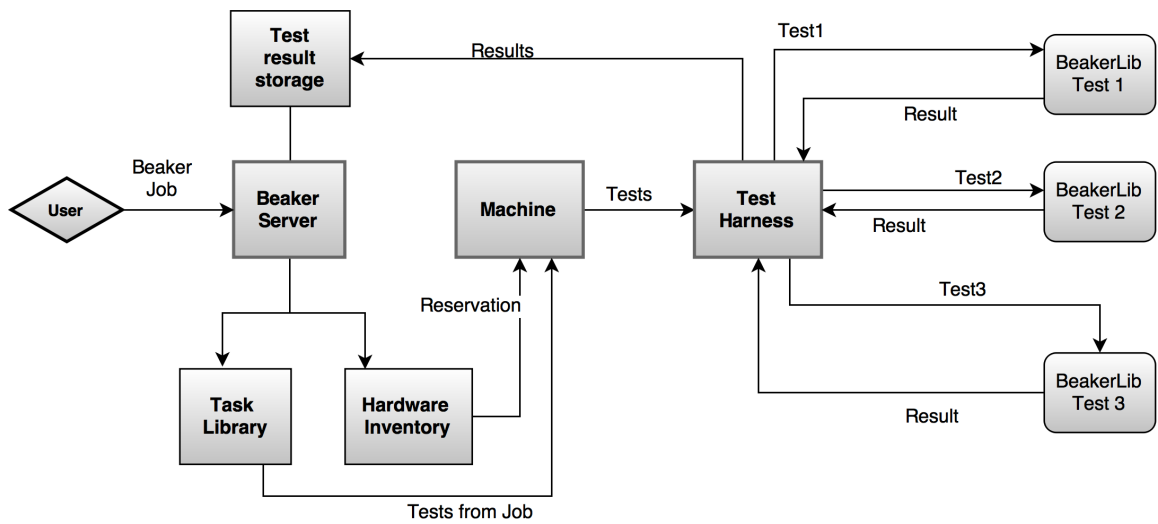
Figure 2.1: Beaker relation to BeakerLib

Chapter 3

BeakerLib

This chapter takes a closer look on inner workings of BeakerLib, with focus on Journal feature and performance issues it suffers from.

3.1 Important functions

As stated earlier BeakerLib is shell-level library with functions that make writing and running tests easier as well as examining their results. BeakerLib adds testing functions to `shell` functionality, so user can combine normal `shell` commands and constructions with helping functions which can make writing tests and examining their results easier. There is close to 80 of these functions (also known as **rlCommands**), description of most used ones follows:

- `rlRun()` - First argument of this function is any `shell` command, which is executed by `rlRun()`. Second is an expected `exit code` of first argument, it can contain one or more codes. Third argument is a comment. BeakerLib logs `FAIL` or `PASS` if expected exit code differs or not from actual one respectively along with comment. This is most used and important function.

- `rlPass()` - Manual assertion and logging of `PASS`. Useful when in combination with if statement which user doesn't want to appear in logs but still wants to log its result. Reciprocal function `rlFail()` exists as well.

- `rlAssertExists` - Asserts whether file given as a first argument exists.

- `rlAssertGrep()` - Function logs `PASS` when pattern given as first argument matches in a file which is given a second argument. Optional flags are passed to `grep` and behave the same way.

- `rlAssertRpm()` - Function asserts `PASS` when package given as first argument is installed. Optional arguments allow specifying particular version, release or arch of the package.

- `rlAssertDiffer()` - Asserts whether two files given as argument differ in their content.

- `rlJournalStart()` - This function is used at the start of each test. It is essential for proper run of the test as it initializes BeakerLib's outputs, which described later in

7

this chapter. Reciprocal function `rlJournalEnd()` must be called at the end of the test.

- `rlPhaseStart()` - This function starts user-defined phase. Function takes two arguments, first one is a type of phase, second one is a name. Phase must be ended by calling `rlPhaseEnd()`. Phases are more closely explained in the next section.

3.2 Phases

BeakerLib divides tests into logical groups called Phases. There are three predefined types of phases:

- Setup - Preparing conditions for the test (such as creating temporary files, starting needed system services and so on), started by calling `rlPhaseStartSetup()`,

- Test - Main phase for testing, started by calling `rlPhaseStartTest()`,

- Cleanup - Reverting changes made by the test, started by calling `rlPhaseStartCleanup()`.

Apart from predefined Phases, user can also define own phases by calling `rlPhaseStart()` function. First argument of the function is a one of two types phase can have:

- WARN - If any rlCommand in phase of this type fails, whole phase will result in Warning state,

- FAIL - Similar to previous type however this time resulting in Failed state.

Basic phases Setup and Cleanup are WARN type, Test phase is a FAIL type.

The result of the whole test is the same as the worst result of any phase in the order: Failed, Warning, Passed. Asserts must not be used outside of phases, if such a case occurs, a new phase opened, its result set to FAIL, then adding the stray assert into the new phase and finally to end it.

This division helps with examining the result of test as it shows which phase, if any, causes fail in BeakerLib's output.

3.3 BeakerLib Test

BeakerLib test is a Bash script which at the start `sources` BeakerLib environment by using file *beakerlib.sh* which is described later in this chapter. Then it can run regular Bash functions as well as rlCommands. If the test is supposed to run in Beaker or other related project, 3 files must exist:

- *runtest.sh* - Script containing the test,

- *PURPOSE* - Plain text file with description of what the test does, it is included in BeakerLib's output which is discussed in one of the following sections,

- *Makefile* - Makefile with instructions how to execute the test, its type and other information used by Beaker or other projects.

Example test 3.1 shows how basic BeakerLib test looks.

```
1  # Include Beaker environment
2  . /usr/bin/rhts-environment.sh || exit 1
3  . /usr/share/beakerlib/beakerlib.sh || exit 1
4
5  PACKAGE=beakerlib
6  # Start of Journal
7  rlJournalStart
8      # Start of Setup Phase, creating temp directory where test will take place
9      rlPhaseStartSetup
10         rlAssertRpm $PACKAGE
11         rlRun "TmpDir=\$(mktemp -d)" 0 "Creating tmp directory"
12         rlRun "pushd $TmpDir"
13     rlPhaseEnd
14     # Start of Test Phase, testing touch and ls commands
15     rlPhaseStartTest
16         rlRun "touch foo" 0 "Creating the foo test file"
17         rlAssertExists "foo"
18         rlRun "ls -l foo" 0 "Listing the foo test file"
19     rlPhaseEnd
20     # Statr of Cleanup phase, temp directory is deleted
21     rlPhaseStartCleanup
22         rlRun "popd"
23         rlRun "rm -r $TmpDir" 0 "Removing tmp directory"
24     rlPhaseEnd
25  rlJournalPrint
26  rlJournalEnd
```

Listing 3.1: Example of basic BeakerLib test

3.4 BeakerLib's output

BeakerLib produces three kinds of outputs. Two file formats and a console one in case of local testing or three file formats when testing remotely.

3.4.1 journal.txt

journal.txt is a plain text file with human readable record of test's progress. After end of each phase, copy of the file is sent to Beaker for storage. Snippet of *journal.txt* generated by Example test 3.1 is shown in 3.2.

```
 1 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 2 :: [   LOG    ] :: Setup
 3 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
 4 :: [   PASS   ] :: Checking for the presence of beakerlib rpm
 5 :: [   LOG    ] :: Package versions:
 6 :: [   LOG    ] ::   beakerlib-1.15-1.fc25.noarch
 7 :: [   PASS   ] :: Creating tmp directory (Expected 0, got 0)
 8 :: [   PASS   ] :: Command 'pushd /tmp/tmp.3iXfiT4GiR' (Expected 0, got 0)
 9 :: [   LOG    ] :: Duration: 1s
10 :: [   LOG    ] :: Assertions: 3 good, 0 bad
11 :: [   PASS   ] :: RESULT: Setup
12 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
13 :: [   LOG    ] :: Test
14 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
15 :: [   PASS   ] :: Creating the foo test file (Expected 0, got 0)
16 :: [   PASS   ] :: File foo should exist
17 :: [   PASS   ] :: Listing the foo test file (Expected 0, got 0)
18 :: [   LOG    ] :: Duration: 0s
19 :: [   LOG    ] :: Assertions: 3 good, 0 bad
20 :: [   PASS   ] :: RESULT: Test
21 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
22 :: [   LOG    ] :: Cleanup
23 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
24 :: [   PASS   ] :: Command 'popd' (Expected 0, got 0)
25 :: [   PASS   ] :: Removing tmp directory (Expected 0, got 0)
26 :: [   LOG    ] :: Duration: 0s
27 :: [   LOG    ] :: Assertions: 2 good, 0 bad
28 :: [   PASS   ] :: RESULT: Cleanup
29 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
30 :: [   LOG    ] :: /performance/beakerlib/Performance/example_test
31 :::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
32 :: [   LOG    ] :: Phases: 3 good, 0 bad
33 :: [   PASS   ] :: RESULT: /performance/beakerlib/Performance/example_test
```

Listing 3.2: Example of journal.txt

3.4.2 Console output

If the executed test is connected to an `interactive shell` similar, human-readable, output to the *journal.txt* is also printed to console's standard output (`stdout`). Apart from *journal.txt*'s content console's output is complemented by executed command's output. Also `shell`'s output is colored for increased readability. Figure 3.1 shows snippet of such output.

```
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::
:: [   LOG    ] :: Test
:::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::::

:: [  BEGIN   ] :: Creating the foo test file :: actually running 'touch foo'
:: [   PASS   ] :: Creating the foo test file (Expected 0, got 0)
:: [   PASS   ] :: File foo should exist
:: [  BEGIN   ] :: Listing the foo test file :: actually running 'ls -l foo'
-rw-rw-r--. 1 jheger jheger 0 May 15 03:20 foo
:: [   PASS   ] :: Listing the foo test file (Expected 0, got 0)
```

Figure 3.1: Snippet from console's output

3.4.3 TESTOUT.log

If the executed test is not connected to an `interactive shell`, the same text generated for Console's output is printed into the file *TESTOUT.log*. This is mostly the case when

executing a test remotely (in Beaker for example), where it is not possible to see the Console output.

3.4.4 journal.xml

Last output is an XML[1] file. XML is a markup language, designed to store and transport data[19].

journal.xml is stripped off of executed commands' own output, but core information (such as which commands were executed, whether they passed or failed and so on) is kept. Also metadata about the test run (time of execution, which component was tested and more) as well as information about the hardware and software test was run on are added. *journal.xml* is sent back to Beaker same as *journal.txt* where it is available for further processing by automated tools. It also serves as a source of information about current state of the test during its execution, for example whether there is currently an open phase or how many failed tests or phases there are so far. Example of *journal.xml* generated by Example test 3.1 is shown in 3.3.

```xml
1  <?xml version="1.0"?>
2  <BEAKER_TEST>
3    <package>beakerlib</package>
4    <pkgdetails sourcerpm="beakerlib-1.15-1.fc25.src.rpm">
5  beakerlib-1.15-1.fc25.noarch </pkgdetails>
6    <beakerlib_rpm>beakerlib-1.15-1.fc25</beakerlib_rpm>
7    <beakerlib_redhat_rpm>beakerlib-redhat-1-6.fc16</beakerlib_redhat_rpm>
8    <starttime>2017-05-15 09:47:44 CEST</starttime>
9    <endtime>2017-05-15 09:47:45 CEST</endtime>
10   <testname>/performance/beakerlib/Performance/example_test</testname>
11   <release>Fedora release 25 (Twenty Five)</release>
12   <hostname>localhost.localdomain</hostname>
13   <arch>x86_64</arch>
14   <hw_cpu>4 x Intel(R) Core(TM) i7-6600U CPU @ 2.60GHz</hw_cpu>
15   <hw_ram>19496 MB</hw_ram>
16   <hw_hdd>459.8 GB</hw_hdd>
17   <purpose>PURPOSE of /performance/beakerlib/Performance/example_test
18  Description: example test created by beaker-wizard
19  Author: Jakub Heger &lt;jheger@redhat.com&gt;
20   </purpose>
21   <log>
22     <phase endtime="2017-05-15 09:47:45 CEST" name="Setup" result="PASS"
23  score="0" starttime="2017-05-15 09:47:44 CEST" type="WARN">
24       <pkgdetails sourcerpm="beakerlib-1.15-1.fc25.src.rpm">
25  beakerlib-1.15-1.fc25.noarch </pkgdetails>
26       <test message="Checking for the presence of beakerlib rpm ">PASS</test>
27       <message severity="LOG">Package versions:</message>
28       <message severity="LOG">  beakerlib-1.15-1.fc25.noarch</message>
29       <test command="TmpDir=$(mktemp -d)"
30  message="Creating tmp directory (Expected 0, got 0)">
31  PASS</test>
32       <test command="pushd /tmp/tmp.3iXfiT4GiR"
33  message="Command 'pushd /tmp/tmp.3iXfiT4GiR' (Expected 0, got 0)">PASS</test>
34     </phase>
35     <phase endtime="2017-05-15 09:47:45 CEST" name="Test" result="PASS" score="0"
36  starttime="2017-05-15 09:47:45 CEST" type="FAIL">
37       <pkgdetails sourcerpm="beakerlib-1.15-1.fc25.src.rpm">
38  beakerlib-1.15-1.fc25.noarch </pkgdetails>
39       <test command="touch foo" message="Creating the foo test file (Expected 0, got 0)">
40  PASS</test>
41       <test message="File foo should exist ">PASS</test>
42       <test command="ls -l foo" message="Listing the foo test file (Expected 0, got 0)">
43  PASS</test>
44     </phase>
45     <phase endtime="201-05-16 09:47:45 CEST" name="Cleanup" result="PASS" score="0"
46  starttime="2017-05-15 09:47:45 CEST" type="WARN">
47       <pkgdetails sourcerpm="beakerlib-1.15-1.fc25.src.rpm">
48  beakerlib-1.15-1.fc25.noarch </pkgdetails>
49       <test command="popd" message="Command 'popd' (Expected 0, got 0)">PASS</test>
50       <test command="rm -r /tmp/tmp.3iXfiT4GiR"
51  message="Removing tmp directory (Expected 0, got 0)">PASS</test>
52     </phase>
53     <message severity="LOG">JOURNAL XML: /var/tmp/beakerlib-dI2ochw/journal.xml</message>
54     <message severity="LOG">JOURNAL TXT: /var/tmp/beakerlib-dI2ochw/journal.txt</message>
55   </log>
56  </BEAKER_TEST>
```

Listing 3.3: Example of journal.xml

---

[1]eXtensible Markup Language

3.4.5 BeakerLib directory

Described files are saved into a BeakerLib test directory created for each individual test.

If the test is run locally, temporary directory is created on system with `mktemp` command, which creates pseudo-random name.

If run on Beaker a unique **TESTID** is generated for each test. This ID serves as a name for test directory as well as an identifier which Beaker later uses when connecting test results with correct test. It is also important in case where restart is a regular part of a test. Upon restarting the test machine the same TESTIDs are relayed from Beaker to Harness with information which tests were already run. Harness then continues with execution of unfinished tests, starting with test that caused the restart, in the same BeakerLib directory it did before, where there are partial results of the test, so it can continue where it left off.

3.5 Source files

This section describes a few of BeakerLib's source files, relevant to this thesis.

- *beakerlib.sh* - Starting point of every tests. It is `sourced` at the beginning of each test and in turn `sources` all other BeakerLib files.

- *testing.sh* - Contains definitions of the most used rlCommands as well as some internal functions.

- *journal.sh* - Provides Bash-side Journaling functionality. Functions from this file process information about what to log and relay them to *journalling.py* (with `rlj` prefix) or query the *journal.xml* to obtain information about the current state of the test(with standard `rl` prefix).

- *journalling.py* - `Python` script responsible for creating most of BeakerLib's outputs. It creates and modifies *journal.xml* file.

- *logging.sh* - Complements *journalling.py* in creating Console output by printing output produced but commands called with `lRun()`.

3.6 Analysis of slow performance

It was reported that BeakerLib suffers performance problems when running long tests. Time of processing each rlCommand grew longer after many (several hundreds and more) were used. Analysis of library was problematic due to lack of documentation, complex structure and uncommented code, however thorough investigation of the source code indicated that problem lies with generating *journal.xml*.

Script *journalling.py* is called after each rlCommand to log its result into *journal.xml*. This isn't big problem with small test as the *journal.xml* file takes up only a few kilobytes, however when the file takes up dozens or hundreds of kilobytes, repeated loading the file from disk, parsing, adding a line of log and then saving the file back to the disk adds significantly more load to CPU[2]. Running larger tests therefore becomes quite time consuming and considerably slows down testing as a whole. This has been determined as the main focus

---

[2]Central Processing Unit

of the thesis since it probably is the most significant performance bottleneck. Influence of used Harness was thought to be negligible and won't be focused on in this thesis.

Figure 3.2 illustrates simplified version of how `rlRun()` propagates through different functions from BeakerLib files (which are `sourced` at the time test execution, depicted by rounded rectangles) and how it is logged into the Journal. Figure also partially reveals complicated environment of BeakerLib, where every rlCommand is processed by many internal functions, making understanding and developing BeakerLib problematic.
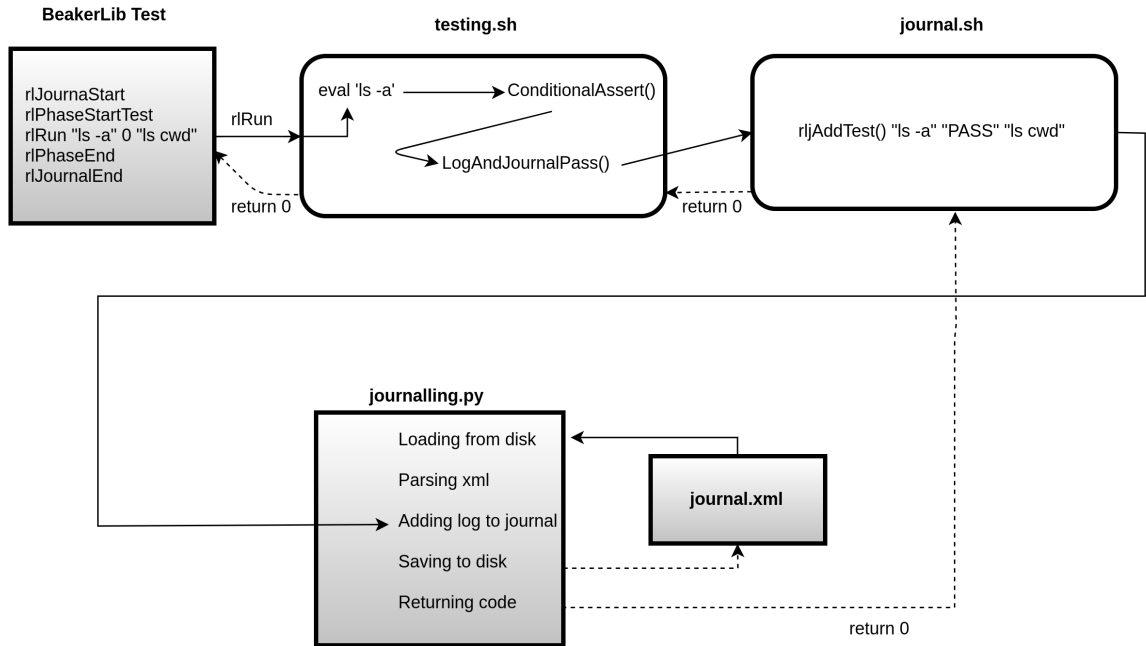

Figure 3.2: Logging of rlRun to Journal

The next chapter describes proposed solutions to analyzed problem with their pros and cons.

Chapter 4

Solution of Journaling problem

Sections in this chapter provide possible solutions to the Journaling problem. Besides explaining the principle of each solution, the sections also discuss their advantages, disadvantages and potential issues.

4.1 Change of XML parser

XML parser is a program which can turn XML document into structured object in RAM[1]. Depending on implementation of the parser, that object is then easier to access by the program as it may provide methods to navigate the object and search it or potentially modify.

Parsing of XML in BeakerLib is done by *journalling.py* script by `Python` module `xml.dom.minidom`[14].

`xml.dom.minidom` is a native part of `Python` from version 2.0 and provides minimal implementation of the DOM[2] interface, with an API[3] similar to that in other languages.

I decided to change parser to different one, to measure whether it will provide better performance. Because of reasons of backward compatibility with RHEL 5[4] which needs to be supported by BeakerLib, the choice of XML parsers was limited to native modules of `Python 2.4.3`. Two additional XML parsers were present for given version.

- lxml - The `lxml` XML toolkit is a pythonic binding for the C libraries libxml2 and libxslt. It combines the speed and XML feature completeness of these libraries with the simplicity of a native `Python` API[2].

  It works similarly to `xml.dom.minidom` in the way that when reading XML object from a file, it will read it whole, builds an object out of it and provides methods for the object to allow access to it.

- xml.sax [15] - `xml.sax` originated as a parser for `Java`[4]. In `Python` it was released with version 2.0. It differs from `xml.dom.minidom` and `lxml` where the two mentioned parsers work with a whole XML file, `xml.sax` emits events as it goes step by step through the file[13]. Using this approach means less memory has to be allocated for

---

[1]Random Access Memory
[2]Document Object Model
[3]Application Programming Interface
[4]Red Hat Enterprise Linux 5

XML handling and therefore makes it ideal when working with very large amount of XML data.

I decided to implement `lxml` parser as it is supposed to be faster and less demanding on memory than `xml.dom.minidom`[3], while keeping its intuitive interface.

4.2 Change in calling journalling.py

Next proposition to make BeakerLib faster is in a way *journalling.py* is called. The assumption being that repeated parsing of XML document slows BeakerLib the most, reducing the number of times it was parsed was then the highest priority.

4.2.1 Queue file solution

First solution is to create a new, temporary **queue file**, which will act as a kind of buffer. rlCommands will behave as before apart from creating BeakerLib journals, but instead they will write message into the queue file. This file will be read and processed only when necessary, that is at the end each phase, when journals are sent to Beaker.

Disadvantages

The way BeakerLib is designed now it in most cases expects some form of return value from *journalling.py* immediately after adding a log to a journal. Performed logging either returns code indicating success of failure or `string` with information about the current state of test. This presents problem as there is no way how to communicate back these information when parsing is postponed.

4.2.2 Daemon-like solution

Second solution is to rewrite *journalling.py* script to have daemon-like behavior.

`Daemons` in Unix are long-running background processes that answers requests for services[9].

This solution will run XML parser as a separate background process for each test. The XML object will be stored in memory, and parsed as whole only at the beginning of journal creation and in case of restarting the test run.

This way BeakerLib can receive response about current test state immediately while still keeping CPU load minimal. Daemon-like solution however brings different obstacles.

Disadvantages

An independent, potentially long running process daemon is more vulnerable to unplanned events such as unexpected exit. This must be addressed by both daemon (to exit as safely as possible) and by the rest of BeakerLib (to detect that daemon is no longer running and to behave accordingly).

Communication

Inter-process communication between running test and daemon has to be created for test to inform which rlCommand is supposed to be logged and for daemon to respond with current state of XML document. This two-way communication must be synchronous to

assure BeakerLib and daemon process their respective messages in correct order. Following options were considered:

- Unix sockets - <expand>

- named pipes - Named pipes are device files. They allow inter-process communication by reading it and writing into is as if regular file, however under normal circumstances the read/write is a blocking operation[6]. This means if one process opens pipe for reading, it will hang there until another process opens the pipe for writing. This feature can be used for synchronization of communication between processes.

I chose to implement communication through named pipes because synchronization issue is taken care of because of the way named pipes are designed.

Chapter 5

Implementation of proposed solutions

This chapter describes how the proposed solutions were implemented. Each solution has its own section that describes implementation details and obstacles that were found and had to be solved during the implementation. During changing of parsers I discovered and fixed few bugs present in current implementation of *journalling.py*.

5.1   Change of XML parser

As mentioned before I chose to change original XML parser to `lxml`. Only changes in source code were in file *journalling.py* as it is only part of BeakerLib that directly works with journal's XML object. Most of the changes were in `xml.dom.minidom`'s method for creating new XML element and assigning value into it. The biggest difference between given parsers is that `lxml` does not provide as many helping methods as `xml.dom.minidom` does. For example in `lxml` there is no method `getElementsByTagName()` to search XML object by a tag name. Instead `lxml` supports `xpath`[20] syntax for searching the object. xpath[1] is part of XSLT[2] standard. It can be used to navigate through elements and attributes in an XML document.

Another example of difference is an approach for accessing element's children. While `xml.dom.minidom` has dedicated methods and attributes such as `hasChildNodes()` which returns `bool` value or `childNodes` which is an iterable attribute of element's children, `lxml` has more low level implementation. It treats elements as `Python lists` so `hasChildNodes()` can be replaced with simple `len(element) != 0`.

Because preliminary performance measurement showed faster test execution with `lxml`, it was decided to implement the rest of the proposed solutions with this parser.

5.2   Queue file solution

This section deals with implementation of queue file solution. It is divided into subsections that discuss files I designed or changed during implementation.

---

[1]XML Path Language
[2]eXtensible Stylesheet Language Transformations

5.2.1   Queue file

Queue file was designed in a way so it was simple to implement, in a human readable format for potential test debugging and easy to extend by new, future functions that will work with it. It is a plain text file, each line containing one buffered message for `Python` script to process later, on demand. Messages are kept in the same format as original solution uses for calling *journalling.py* to preserve consistency with the exception that they are now escaped, which is described in section.

5.2.2   journal.sh

Creation of queue file, by using `touch` command, was added to function `rlJournalStart()` which initializes Journaling functionality. Using `touch` assures that if the queue file already exists (which happens when test run is interrupted and started again), its content is not deleted (in case of restart of the testing machine as described in section 3.4.5).

It now also `exports` new variable `BEAKERLIB_QUEUE`, with queue file's path, into test's environment so `Python` script *queued_journalling.py*, can later access it.

Original calling of *journalling.py* script, which is a main functionality of *journal.sh*, was replaced in one of two ways:

- Delayed calling - New function `rljPrintToQueue()` takes all arguments that were originally meant for *journalling.py* and instead prints them into the queue file, where it will be processed by *queued_journalling.py* later during execution of the test. This concerns functions which do not necessarily require response about current test state from *journal.xml*. In original solution responses to these functions were only `exit codes` which they did not utilize in any way or if they did theirs functionality was re-implemented. Namely functions that use delayed calling are: `rlJournalPrint()`, `rljAddTest()`, `rljAddMetric()`, `rljAddMessage()`, `rljRpmLog()`

- Immediate calling of *queued_journalling.py* - Essentially the same as the original solution. These functions require immediate response. Using this way of calling won't save on any CPU load (in fact the load will be slightly higher than before because of operations related to queue file processing), however in typical BeakerLib test these functions are in minority compared to previous type of calling. Functions and the response they require are:

    - `rlJournalStart()` - requires confirmation that journal was initiated successfully,
    - `rlJournalPrintText()` - requires *journal.txt* which is generated from current *journal.xml*,
    - `rlGetTestState()` - requires number of failed asserts in the test so far,
    - `rlGetPhaseState()` - requires number of failed phases in the test so far,
    - `rljAddPhase()` - requires immediate print of name of the new phase to Console output,
    - `rljClosePhase()` - requires result of closed phase, to send it to Beaker along with Journal,
    - `rlJournalEnd()` - requires immediate print of *journal.txt* which is generated from *journal.xml*.

Function `rljAddTest()` is the cause of the most calls of *journalling.py* in original solution, therefore had the highest need to be moved into group of functions with Delayed calling. However it does require knowledge of current state of the test. That being situation when Assert (rlCommand using `rljAddTest()` for Journaling) is used outside of a phase, such information is held only in current *journal.xml*. To solve this problem functionality of `rljAddTest()` had to moved into *queued_journalling.py* script, discussed in the next subsection.

Apart from printing to queue file, `rljPrintToQueue()` also has to escape given arguments. This needs to be done because firstly some of the arguments originating from user may contain newline character which would break the „one queued command per line" rule in queue file's format and secondly so *queued_journalling.py* may process it with `optparse` module. Escaping is done with `printf` Bash builtin[1], specifically its `%q` option which causes `printf` to output in `shell-quoted` format.

5.2.3 queued_journalling.py

File *queued_journalling.py* originated from *journalling.py* but it differs in several ways.

Now when it is called, it first parses current *journal.xml* and then calls new method `updateXML()` with parsed XML object as an argument. This method opens queue file and finds last line it accessed in previous call. From there it reads queued lines, parses each with `Python`'s `optparse` module and modifies the XML object accordingly in the similar way it did originally, this time however without parsing *journal.xml* each time as the XML object is passed as an argument to appropriate methods.

When it reaches end of file, it makes a mark (by adding a line at the end of the queue file with a number already processed lines) for future readings and returns to the original call coming from one of the *journal.sh*'s Immediate calling functions. After modification from that function it generates response and returns it to *journal.sh*.

Exceptions to this behavior are:

- `rlJournalStart()` - This function doesn't access queue file but only initializes XML object and returns an `exit code` whose value depends on whether the initialization was successful,

- `rlJournalEnd()` - This one makes sure every queued command was processed as it is an exit point from the test and so last opportunity to modify *journal.xml*.

As mentioned in previous subsection, functionality of `rljAddTest()` had to be altered. Given that Bash side of BeakerLib had no way of knowing if the test was added outside of phase at the time of writing this operation into the queue file, this action had to be resolved when *queued_journalling.py* processed the queue file. New method `testOutOfPhase()` was implemented which is called when assertion outside of phase is detected and it performs the same process as when this event happened in original *journal.sh*, described in chapter 3 in section Phases.

5.2.4 Problems with implementation

Main goal of this solution was to reduce number of times *journal.xml* is parsed, by delaying as many Journaling operations as possible, while keeping BeakerLib's outputs the same. The way BeakerLib is designed now it is not possible, because some information is always

lost when operations are delayed. In case of this implementation I was able to keep *journal.xml*, and therefore *journal.txt* as well, the same as with original solution, however at the price Console's output (therefore *TESOUT.log*'s too as it is Console's output printed to file) which is now missing all information usually given by functions from Delayed calling category. Only complementary output created by functions from *logging.sh* remain.

Solving this issue would require more extensive changes to BeakerLib's design which I decided not to implement for now so Queue file solution remains only as a proof of concept.

5.3 Daemon-like solution

This section describes individual changes I made to BeakerLib's design in order to implement Daemon-like solution.

5.3.1 journal.sh

Function `rlJournalStart()` in this implementation creates named pipe using `mkfifo` and then `exports` its path into environment. Then it spawns *daemon_journalling.py* process in the background with `&` operator and stores its `PID`[3].

Every call of *journalling.py* in original implementation was replaced with new function `rljCallDaemon()`, which takes the same arguments as original function. When this new function is called it firstly escapes given arguments using similar way as in queue file, this time however another function had to be created. `rljCallDaemon()` passes its argument to the function `escapeArguments()` which uses `printf` and `echo` Bash builtins to escape arguments in loop which are then caught back in `rljCallDaemon()` with `$()` construct[5] for catching output. It is implemented this way to avoid using temporary file.

After arguments are escaped, `rljCallDaemon()` checks whether the daemon is still running with `kill -0 $DAEMON_PID` call.

`kill` program is used to send signals to processes. If used with `-0` option, no signal is actually sent but error checking against the process is still performed and it returns `0` when process with given PID is running[8]. This is done to make sure the daemon is still running before pipe writing operation. If the daemon wasn't running before writing to pipe, the test would hang there indefinitely, so if the daemon is not running, the test exits with error.

After this check is performed, `rljCallDaemon()` writes to named pipe escaped message, where it waits until daemon reads it and responds. Response is read as a next action, decoded from a format that will be discussed in the next subsection and then the response is returned to function that called `rljCallDaemon()`. This is repeated until end of the test is reached, where function `rljJournalEnd()` sends `signal` with `kill` to end the daemon.

Function `rlJournalPrintText()` had to be reworked slightly. This function generates *journal.txt* output from *journal.xml* and has two main usages:

- It is a standard part of BeakerLib test where it is called directly right before test ends by calling `rlJournalEnd()`. It prints *journal.txt* to `stdout` if the test is connected to `interactive shell`.

- It is used by internal functions during test execution to continuously generate partial *journal.txt* files and send them back to Beaker if using Beaker and harness or storing them to disk when no harness is used.

---

[3]Process identifier

Reworked function now accepts one optional argument and passes it to `rljCallDaemon()` and it was added to all currently implemented BeakerLib functions that call `rlJournalPrintText()`. In original solution there was a simple way how to have *journalling.py* print either to `stdout` or to catch the same output into variable using `$()` construct or redirecting it to a file, because each call of *journalling.py* was a separate process whose output could be controlled. When using daemon however, this is no longer possible, either all output would be caught or none at all. A way how to differentiate which *journalling.py*'s output is supposed to be printed to `stdout` and which is supposed to be returned to *journal.sh* through named pipe had to be implemented. Functions that need to catch the daemon's output now use `rlJournalPrintText()` with the optional argument. How it affects *daemon_journalling.py* is described in next subsection. Currently unused function `rlJournalPrint()` was reworked is the same way.

5.3.2 journalling_daemon.py

journalling.py script again originates from *journalling.py*. This time however, it is designed to run in endless loop, instead of returning after one executed action.

Before the daemon enters the endless loop, it performs checks whether environment is prepared for it (whether named pipe exists or it can access test's PID). Only if all checks are successfully verified it enters the loop, otherwise exits with error.

In each iteration it checks whether test's process is still running analogously to how `rljCallDaemon()` does it. Then it reads the named pipe and waits there until `rljCallDaemon()` writes to it. After message is read, method `parseAndProcess()` is called and it parses the message and acts upon it.

If the message received comes from `rlJournalStart()` the XML object is initialized and stored in `global` variable `jrml` so it its accessible to all other methods that use the XML object.

As was stated in previous subsection, change in outputting behavior had to be implemented. When optional argument `toVar` is detected all functions related to printing instead of `print` function store their respective outputs in variables. These are gradually appended to each other and at the end of all printing they are instead returned through named pipe back to *journal.sh* where individual functions can catch them.

Any other message call the same methods as in original implementation with the difference which is that now the methods use the `global jrnl`.

When a message is processed, `parseAndProcess()` must encode the response, because original implementation was able to respond with either `return code` or `string` and this solution is only ably to respond with `string`. Simple format `message:X-code:Y`, where X is replace with `string` and Y with `return code`, was implemented which is quickly decoded by `rljCallDaemon()` using Regular expression[7].

5.3.3 Signals

Signals are asynchronous interrupts that are used for inter-process communication. Signals are usually used by the operating system to notify processes that some event occurred[11]. For example when operating system plans to reboot, it sends signal to all running processes to inform them that reboot will take place. `rlJournalEnd()` in daemon-like solution sends signal `SIGTERM` to end *daemon_journalling.py*'s process where it is caught and handled by daemon's `signal handler`.

Signal handlers are functions that are called when program receives signal to handle the event properly. In daemon solution Signal handlers were added to both daemon and Bash side of BeakerLib.

In *daemon_journalling.py* method `signalHandler()` was created. It is set to handle most common signals, that would cause it to exit improperly. When such signal is received, daemon interrupts what is currently doing and through `signalHandler()` calls `saveAndExit()` method which saves current state of XML object to disk and exits.

journal.sh uses `trap` command to catch signals. Upon receiving signal it `kills` daemon to always make sure that daemon will not stay running in the background after test is unexpectedly ended.

5.3.4 Problems with implementation

Using background processes with blocking operations is a rather volatile solution. In a case of some unanticipated event it may happen one side or the other may be hung up on blocking operation with no process to unblock it, even though Signal handlers were implemented to lower the chance of such a situation to happen.

During testing of this solution such behavior was not reproduced. However testing on much larger scale, including different operating systems, CPU architectures and other testing conditions, would have be concluded to confirm it is unlikely such event could occur.

5.4 Verification of implemented solutions

Verification that implemented optimizations didn't cause regression, was directed on *journal.xml* as it is a main focus point of this thesis. Due to its nature automated verification was problematic to implement, because two *journal.xml* files generated from one test may differ in many ways while both still being valid (they may differentiate in such information as time of execution, hardware/software specifications or even in reported results, as test could pass or fail independently on BeakerLib implementation).

Because of this reasons only manual verification took place.

Chapter 6

Performance testing

This chapter explains what performance testing is and why is ti done. Then it describes what tests and in which environments were used to measure performance of BeakerLib before and after optimizations were implemented.

Performance testing is a type of **non-functional testing**, that is testing whose goal is to test quality characteristics of a component, rather than its functionality[10].

For performance testing of BeakerLib two kinds of tests were chosen to run, in two kinds of testing environments. They are described in following sections.

6.1 Tests

6.1.1 Artificial tests

First type of tests are artificial tests created by `beaker-wizard` tool to specifically target and measure performance of Journaling modifications this thesis proposes. They consist mostly of rlCommands that directly work with *journalling.py* (or its variants of implemented modifications), for example commands `rlLog()` or `rlPhaseStart()` and `rlPhaseEnd()`. This way we can observe clear difference in performance without being affected by operations unrelated to Journaling (executing actions that verify functionality of components in real tests).

- Test1 - Test used as and example 3.1. It is a very short test and for which proposed solutions were not aimed therefore increase in performance with this test is not expected. Test contains 17 rlCommands.

- Test2 - Test contains 1000 calls of function `rlLog()` divided into 3 phases. This function requires very little overhead and so its results represent direct performance difference caused by implementations however they do not represent performance difference when running typical BeakerLib test. Test contains 1014 rlCommands.

- Test3 - Test consists of 500 calls functions of `rlPhaseStartTest()` and `rlPhaseEnd()`. Even thought it has the same length as Test2, this test is expected to run longer because phase controlling rlCommands have larger overhead when working with *journal.xml*. Test contains 1014 rlCommands.

- Test4 - Test comprises of 500 phases with a few typical rlComands inside them. This test resembles typical BeakerLib the most out of all artificial ones and is longest as well. Test contains 3013 rlCommands.

These tests are included in Appendix A in directory tests/.

6.1.2   Real tests

Second type are examples of real tests used in Red Hat. Finding such tests was problematic because real tests are often written specifically for particular hardware or software and behave differently under different circumstances. The challenge then was not only to find long running tests but most importantly tests that all have the same behavior on one specific remote machine as well as on local machine used for testing. However at least minimal testing has to be performed to measure whether modifications have an effect outside of controlled environment.

- Test1 - This is a regression test for component sos which is a data collection tool. It checks whether all expected data are in fact collected. Test contains 610 rlCommands but big part of it is done without rlCommands with regular commands and only results of those are logged with rlCommands.

- Test2 - This is a install test for `maven` component, which is a project management and comprehension tool. It is used to test whether maven can be installed. Test contains 1026 rlCommands.

These tests are not included in Appendix as they are property of Red Hat.

6.2   Testing Environment

6.2.1   Local

First environment is local laptop for convenience and speed of execution. Tests were run directly, without any harness and with these technical specifications:

Model	Lenovo T460s
CPU	4 cores Intel(R) i7-6600U, 2.60GHz
CPU architecture	x86_64
RAM	19496 MB
Operating System	Fedora release 25

6.2.2   Remote in beaker

Second round of testing was done to emulate real testing conditions and to verify that changes made to BeakerLib do not break functionality outside of controlled environment. Tests were run using the default test harness Beah. Technical specification were following:

CPU	1 core Intel(R) Xeon, 2.10GHz
CPU architecture	x86_64
RAM	2847 MB
Operating System	Red Hat Enterprise Linux Server 7.3

6.3 Measured Value

Goal of this thesis was to optimize performance in regards to time of execution, so that is only the value that was measured. Values were obtained from *journal.xml* which has builtin mechanisms to track how long took the execution of the test, with precision of one second, which should be sufficient for needs of this measurement

Chapter 7

Analysis of results

This chapter compares measured values of executed tests under individual implementations and analyzes why they are that way. Each test described in previous chapter was run 15 times with each implementation in both devised environments. Measured results were averaged and rounded to precision of one decimal place. Complete data set is located in Appendix B.

7.1 Tests

This sections analyzes measured results of individual tests. Each subsection is dedicated to a single test and it shows how the test performed on different implementations. Mostly only results from remote testing are shown as they are more important for this thesis.

7.1.1 Test1

As expected Test1 didn't reveal difference in performance, because drawbacks of original implementation become apparent only when *journal.sh* reach certain size. Results can be seen in figure 7.1.

7.1.2 Test2

Test2 indicates big performance improvement when using queue file or daemon, however as discussed in section 6.1, this test may show big improvement in performance but will not necessarily be reflected in actual test runs. Queue file solution was able to achieve such speed because the test contained minimal number operation that required immediate interaction with *journal.sh* file which would be highly atypical for real test. Results are depicted in figure7.2.

7.1.3 Test3

Test3 behaved in surprising manner. While performance of first three implementation were awaited, daemon one was expected to perform much better than is shown in figure 7.3. Even stranger is that when tested locally it performed as expected which can be seen in figure 7.4. Explanation of this behavior was not fully understood at the time of writing this thesis.
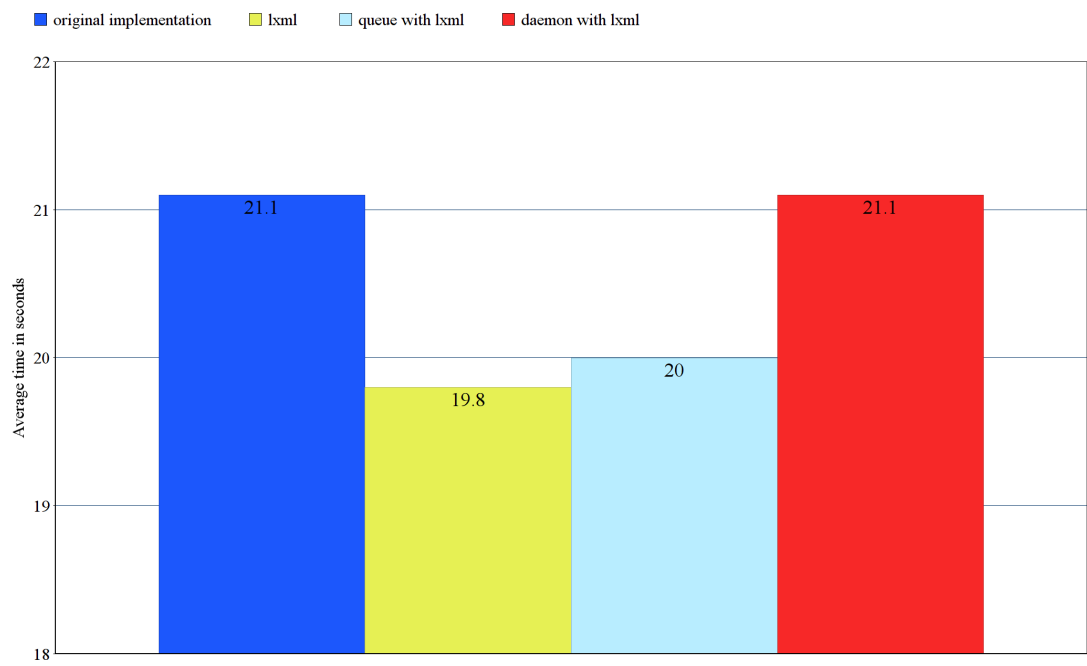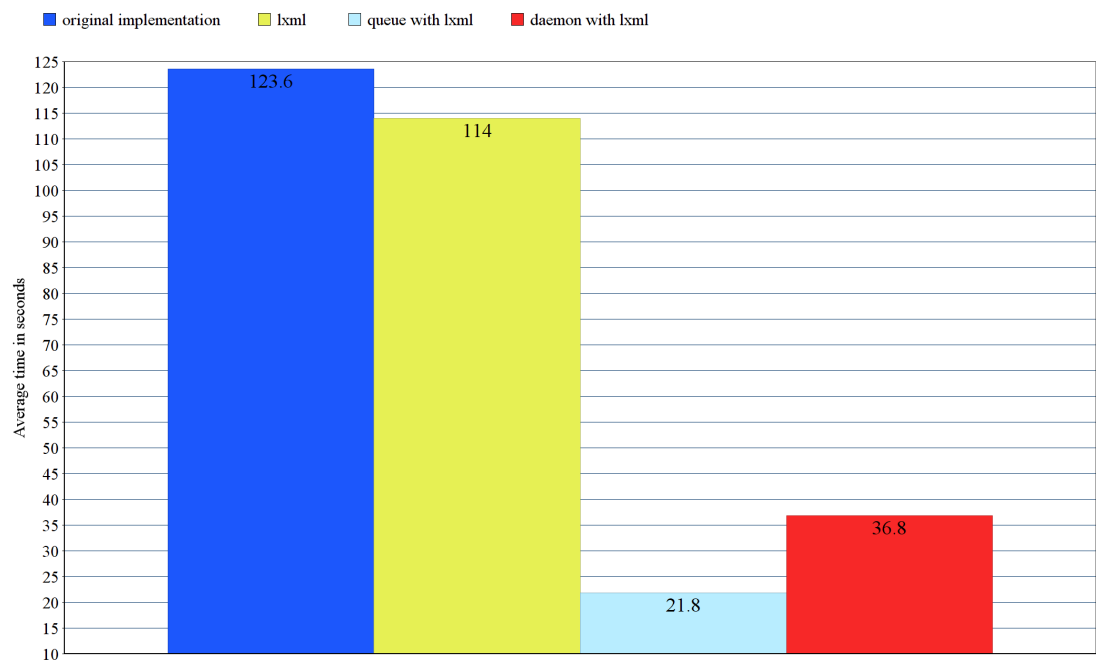
Figure 7.1: Test 1 results from remote testing

Figure 7.2: Test 2 results from remote testing
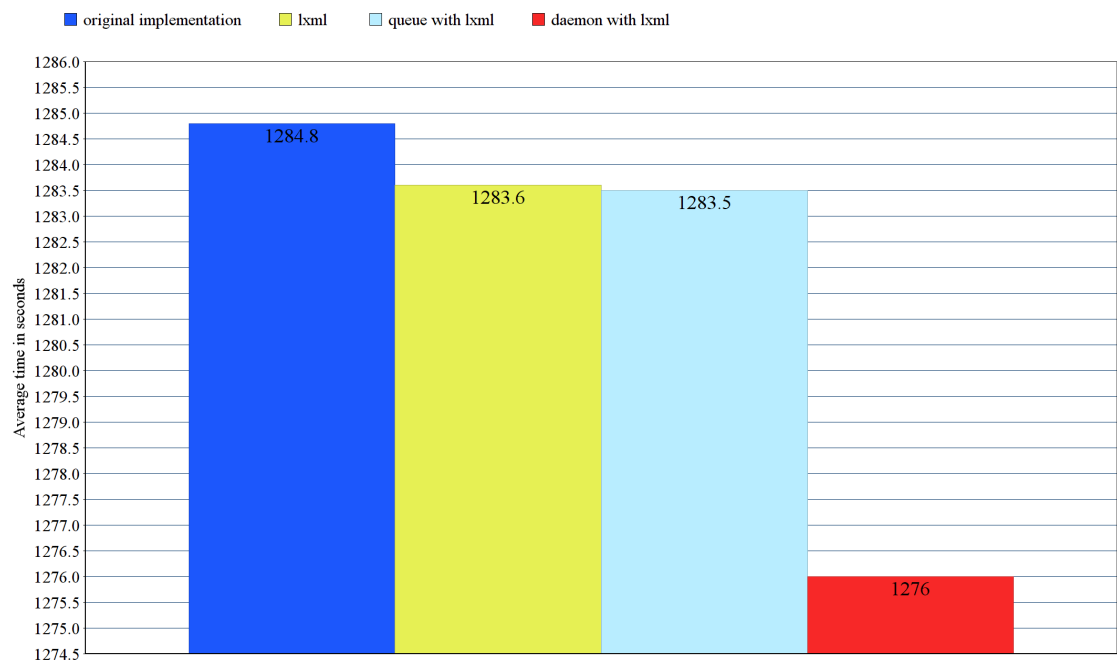
Figure 7.3: Test 3 results from remote testing
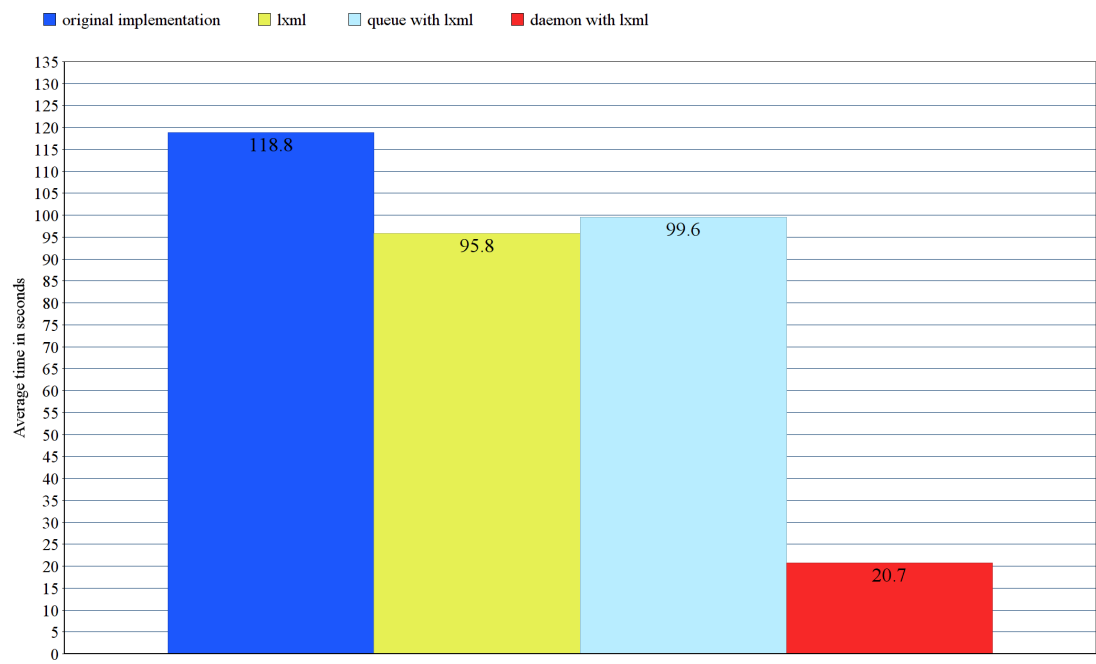
Figure 7.4: Test 3 results from local testing

Chapter 8

Conclusion

Recap of results
Future work

Bibliography

[1] Manual page bash_builtins(1) General Commands Manual. April 2004.

[2] Behnel, S.: *lxml documentation*. [Online; visited 15.05.2017].
Retrieved from: http://lxml.de/index.html

[3] Behnel, S.: *lxml performance*. [Online; visited 15.05.2017].
Retrieved from: http://lxml.de/performance.html

[4] Brownell, D.: *Sax2*. O'Reilly Media. 2002. ISBN 0596002378.

[5] Cooper, M.: *Command Substitution*. [Online; visited 16.05.2017].
Retrieved from: http://www.tldp.org/LDP/abs/html/commandsub.html

[6] Goldt, S.: *Blocking Actions on a FIFO*. [Online; visited 16.05.2017].
Retrieved from:
http://www.tldp.org/LDP/lpg/node19.html#SECTION00734000000000000000

[7] Goyvaerts, J.: *Regular Expressions Tutorial*. [Online; visited 16.05.2017].
Retrieved from: http://www.regular-expressions.info/tutorial.html

[8] Goyvaerts, J.; Zak, K.: Manual page kill(1) User Commands. July 2014.

[9] Indiana University: *In Unix, what is a daemon?* [Online; visited 16.05.2017].
Retrieved from: https://kb.iu.edu/d/aiau

[10] ISTQB Exam Certification: *What is Non-functional testing?* [Online; visited 17.05.2017].
Retrieved from: http://istqbexamcertification.com/what-is-non-functional-testing-testing-of-software-product-characteristics/

[11] Little Unix Programmers Group: *Introduction To Unix Signals Programming*. [Online; visited 16.05.2017].
Retrieved from: https://web.archive.org/web/20130926005901/http://users.actcom.co.il/~choo/lupg/tutorials/signals/signals-programming.html

[12] Peck, B.; Callaghan, D.; Bastian, J.; et al.: *Restraint documentation*. [Online; visited 15.05.2017].
Retrieved from: http://restraint.readthedocs.io/en/latest/

[13] Python Software Foundation: *Sax*. [Online; visited 15.05.2017].
Retrieved from: https://wiki.python.org/moin/Sax

[14] Python Software Foundation: *xml.dom.minidom documentation.* [Online; visited 15.05.2017].
Retrieved from: https://docs.python.org/2/library/xml.dom.minidom.html

[15] Python Software Foundation: *xml.sax documentation.* [Online; visited 15.05.2017].
Retrieved from: https://docs.python.org/2/library/xml.sax.html

[16] „Red Hat Inc.“: *Beah documentation.* [Online; visited 14.05.2017].
Retrieved from: https://beah.readthedocs.io/en/latest/

[17] „Red Hat Inc.“: *Beaker documentation.* [Online; visited 14.05.2017].
Retrieved from: https://beaker-project.org/

[18] „Red Hat Inc.“: *BeakerLib GitHub wiki man page.* [Online; visited 15.05.2017].
Retrieved from: https://github.com/beakerlib/beakerlib/wiki/man

[19] W3Schools: *Introduction to XML.* [Online; visited 15.05.2017].
Retrieved from: https://www.w3schools.com/xml/xml_whatis.asp

[20] W3Schools: *XPath Tutorial.* [Online; visited 15.05.2017].
Retrieved from: https://www.w3schools.com/xml/xpath_intro.asp

Appendices

List of Appendices

Appendix A

Content of enclosed CD

Files:

- **pdf** - PDF version of the thesis

Appendix B

Measured values

B.1 Baseline measurements

Local

Test	Measured time is seconds															Average
Test1	1	1	1	1	1	1	0	1	1	1	1	0	1	1	1	0.9
Test2	62	63	63	64	67	66	63	63	66	65	65	63	64	64	61	63.9
Test3	109	109	110	110	112	110	119	112	113	111	114	111	118	110	109	111.8
Test4	393	396	397	408	398	395	400	421	429	407	395	398	399	395	403	402.3
Test5	45	46	43	47	42	42	43	42	42	42	42	42	42	43	43	43.1
Test6	4	5	4	3	3	3	4	4	5	3	4	4	3	3	2	3.6

Remote

Test	Measured time is seconds															Average
Test1	27	21	19	22	19	19	19	20	20	20	19	19	19	19	20	21.1
Test 2	132	125	127	122	122	122	122	121	121	125	120	123	124	122	126	123.6
Test 3	1239	1303	1241	1258	1274	1282	1295	1289	1307	1285	1293	1296	1295	1303	1312	1284.8
Test 4	1417	1400	1390	1383	1535	1501	1548	1413	1490	1531	1541	1455	1437	1487	1402	1462
Test 5	313	189	184	180	182	178	181	182	182	181	180	184	182	178	180	190.4
Test 6	85	40	40	41	38	38	40	46	39	39	39	39	38	39	40	42.7

B.2 Implemented optimizations

B.2.1 lxml parser

Local

Test	Measured time is seconds															Average
Test1	1	1	2	1	1	1	1	2	1	1	2	1	1	1	1	1.2
Test2	61	60	65	63	62	61	61	61	63	60	60	62	63	61	59	61.5
Test3	100	88	88	88	96	96	96	97	98	101	98	96	97	98	100	95.8
Test4	228	228	236	236	241	246	236	219	214	214	211	208	206	222	231	225.1
Test5	46	46	45	46	46	46	47	45	47	46	47	47	46	48	46	46.3
Test6	6	6	6	5	5	5	5	5	5	6	5	5	5	3	6	4.9

Remote

Test	Measured time is seconds															Average
Test1	25	19	19	19	19	20	20	19	20	20	20	19	19	19	20	19.8
Test2	121	113	113	113	113	113	113	115	114	115	114	114	113	113	113	114
Test3	1322	1223	1247	1264	1261	1281	1290	1292	1301	1286	1293	1293	1295	1302	1304	1283.6
Test4	1042	1035	1036	1109	1097	1049	1039	1033	1037	1039	1040	1037	1036	1035	1037	1046.7
Test5	302	180	175	175	175	176	176	176	177	178	175	176	180	172	175	184.5
Test6	75	37	36	36	36	37	37	37	38	36	36	43	37	37	36	39.6

B.2.2 Queue file solution with lxml parser

Local

Test	Measured time is seconds															Average
Test1	0	0	0	0	1	1	1	1	1	0	0	0	0	0	0	0.3
Test2	4	4	3	4	5	4	4	4	4	4	4	4	4	4	4	4
Test3	101	102	105	97	98	99	101	97	100	99	98	98	97	103	99	99.6
Test4	119	119	116	117	121	117	116	117	118	124	116	115	116	116	107	116.9
Test5	12	11	12	12	11	12	12	12	12	12	12	12	12	12	12	11.9
Test6	2	2	2	2	2	1	2	1	1	2	2	1	1	2	1	1.6

Remote

Test	Measured time is seconds															Average
Test1	27	19	19	20	20	19	19	20	19	19	20	20	19	20	20	20
Test2	27	22	22	22	22	21	21	21	21	21	21	21	21	22	22	21.8
Test3	1316	1227	1254	1259	1276	1283	1288	1292	1281	1291	1289	1293	1292	1307	1305	1283.5
Test4	898	890	881	875	876	877	871	910	994	957	977	935	880	913	935	911.3
Test5	249	130	133	134	130	134	131	136	133	133	133	133	132	129	131	140.1
Test6	86	45	39	36	36	36	36	37	36	43	36	38	37	35	36	40.8

B.2.3 Daemon solution with lxml parser

Local

Test	Measured time is seconds															Average
Test1	1	0	0	1	0	0	1	0	0	1	0	0	1	0	0	0.3
Test2	5	6	6	6	6	6	5	5	5	6	5	6	6	7	7	5.8
Test3	20	20	21	21	20	20	19	22	21	21	20	21	20	24	20	20.7
Test4	44	44	44	44	44	44	45	42	42	44	46	45	45	46	44	44.2
Test5	13	13	13	13	13	13	13	13	13	12	13	13	13	13	13	12.9
Test6	14	14	14	14	14	14	14	14	14	14	14	14	15	14	14	14.1

Remote

Test	Measured time is seconds															Average
Test1	27	21	19	22	19	19	19	20	20	20	19	19	19	19	20	21.1
Test2	40	41	32	37	43	30	32	28	29	55	55	33	42	29	27	36.9
Test3	1249	1226	1243	1256	1271	1294	1278	1292	1287	1286	1283	1283	1290	1304	1308	1276.7
Test4	773	780	787	751	814	745	731	736	783	757	759	787	741	732	736	760.8
Test5	238	129	125	124	125	122	121	118	118	119	119	123	120	119	123	129.5
Test6	124	50	51	48	47	46	49	48	52	49	50	48	49	48	46	53.7