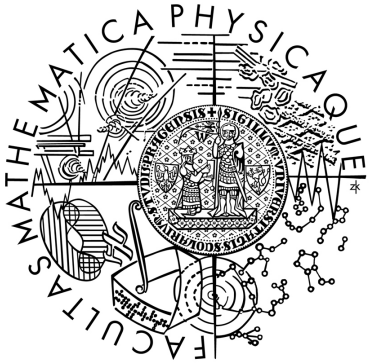
Univerzita Karlova v Praze
Matematicko-fyzikální fakulta

MASTER THESIS



Jiří Zouhar

Regression Testing For zlomekFS

Department of Software Engineering

Supervisor: Doc. Ing. Petr Tůma, Dr.
Study Program: Computer Science

2008

I would like to thank my supervisor, Doc. Ing. Petr Tůma, Dr., for his valuable advice.

I declare that I have written this master thesis on my own and listed all the used sources. I agree with lending of the thesis.

Prague, April 18, 2008

Contents

Název práce: Regression Testing For zlomekFS
Autor: Jiří Zouhar
Katedra (ústav): Katedra softwarového inženýrství
Vedoucí bakalářské práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@mff.cuni.cz

Abstrakt: Jednou třeba přeložím, nejdřív zkrátit...

Klíčová slova: testování software, extrémní programování, ladění programů

Title: Regression Testing for zlomekFS
Author: Jiří Zouhar
Department: Department of Software Engineering
Supervisor: Doc. Ing. Petr Tůma, Dr.
Supervisor's e-mail address: petr.tuma@mff.cuni.cz

Abstract: ZlomekFS is a distributed file system aimed to transparent sharing of directory trees. This thesis describes how regression testing for it was build.
Firstly, it summarizes actual methods used for software testing, debugging, and bug tracing. Where applicable, emphasis is put on specific application of these methods on filesystems. On the basis of this research, actual system for ZlomekFS is considered and build. System consists of six parts.
Unit testing framework for C code providing automatic test discovery with minimalistic interface.
Logging facility with C and python interface using message importance and concerns separation for filtering. The logger is remotely controllable.
Workload generator which can generate random test sequence. Rules for random workload can be defined by tests content and dependency graph describing order constrains. Failed sequences can be saved for later reproduction. The generator can try to find the shortest sequence to reproduce error.
Test controlling and reporting framework. This part controls test execution, prepares sane environment, executes tests, and collects data for further failure investigation.
Test result repository with user interface where test results are stored.
Continuous integration server which triggers events and runs automatic builds.

Keywords: software testing, extreme programming, debugging

Chapter 1

Introduction

ZlomekFS is a special distributed filesystem aimed at transparent sharing of directory trees between any number of computers (nodes).

A unit of sharing (local directory) is called a volume. In ZlomekFS, there are no server and client nodes, the hierarchy is a general graph where every node can export volumes. Despite of this, resulting export hierarchy must be a tree. When two nodes connect, for unique volume one of them must be master providing content, and the other client. This relationship can be bidirectional (client for one volume can be master for another).

Node can cache content of volume obtained from another node and provide it to further nodes. But this caching is not required, node can use remote volume without local cache too. On one node, all volumes must be mounted under tree with one root directory (volume), while other volumes are mounted beneath.

ZlomekFS doesn't require any special layer on disk, arbitrary filesystem is used for storing cached content. To reduce network bandwidth, ZlomekFS doesn't handle files as elementary units, but operates on *chunks* (part of file with predefined size).

For mobile nodes, there are two modes to operate in (beyond full speed connection). If there is no connection, node can operate in disconnected mode (data will be served from cache only). Eventually, when node connects again, data are synchronized with other nodes. Last mode is *slow connection* mode, when, to limit traffic, only directories and data actually read from files are synchronized with master node. When changes are made to files in disconnected mode, conflicts can arise upon synchronization. In ZlomekFS, conflict is represented as files in special directory.

As in any software, in filesystems can be bugs too. But in case of filesystems, there is even bigger need for reliability. This is caused by low-level character of filesystems. Every application depends on proper work of components beneath them, where most of them use filesystem to store permanent

data. To ensure reliability of filesystem, exhaustive testing should be done to cover maximum of use-cases filesystem could be used in. This is convenient for one term development, but essential when further extension and development would be done.

However, ZlomekFS lacks any tests, or testing environment. As described, ZlomekFS is very complex and special filesystem, where there are only few similar filesystems. Because of this, there is need to develop at least special tools to allow testing of it's special functions, or new framework at all. The framework should be able to provide debugging and tracing information, if bugs are found.

1.1 Goals

Extend the existing zlomekFS implementation by introducing a regression testing framework. The framework should be capable of submitting both predefined and random workload to the filesystem and, either by comparing the results with the same operations performed over another filesystem, or by some other appropriate means, identify filesystem errors. The identification of an error should contain both a minimal sequence of steps necessary to reproduce the error, and the debugging protocol excerpt relevant to the error. The framework should include support for generating the debugging protocol and changing the network conditions.

Make all the developer documentation an integral part of the zlomekFS project using appropriate tools such as DoxyGen.

- proc network
- proc random, jake operace
- cleanup
- ?config?
- jak minimal sequence
- proc porovnani

1.2 Structure of the thesis

Chapter 2 describes common techniques used for testing. Basic test types are listed with their usage and aims. Reasons are given for what tools should be used for testing, and how they can help debugging.

Chapter 3 summarizes tools used in actual regression system for ZlomekFS, gives reasons for why they were chosen, and list their main features.

Chapter 4 describes internals of the system: changes made in ZlomekFS are written down, then component interaction is described, and finally each

component function is documented.

Chapter 5 summarizes the work done, how goals are met and what new approaches are used.

Chapter 2

Filesystem testing

The most desired feature on testing framework is ease of use, tightly coupled with automation. To achieve this, the tests have to be written in readable format. There is tendency to place small unit tests as near to actual code as possible to allow easy maintenance. By the mean of regression testing, they have to be executed automatically, in scheduled periods (defined by amount of time or number of changes). The results should be collected and presented centrally.

For tracing the code execution, there may be some tracing tools and logging tools. They have to have minimal footprint, but collect as much information as possible. Their output must be formatted in way compatible with the automation framework.

The output of tests should be accompanied with some state information from the time of failure. This can be achieved by using some tool for creating snapshots, that may or may not support resuming.

For filesystem testing it is hard to find good testing patterns which will cover all cases that can occur. So it is good idea to have some random workload generator that can randomly exercise the filesystem. The problem with this approach is that outputs of such testing tends to be very big and only a small portion of them is related to the error. To allow random testing and avoid this unwanted side effects, some pruning algorithm has to be used. The reruns of tests may use the snapshots, if the method used for creating snapshots make the resume possible.

As the ZlomekFS is multi-threaded, distributed filesystem, the suite should have some support, or at least extensibility to allow control or simulation of distributed environment.

2.1 Test types

Filesystem can be seen as many things, and thus it can be tested from many points of view. Brief description of general test types follows.

2.1.1 Specification testing

We could look on filesystem as on specification of way how to store data and associated metedata on storage media. In this case, we can ask if the structures specified are sufficient for accessing stored data, if the specification covers all eventual operations that should be available, and if the transitions made by operations are sane and leads from correct state to correct state. Specification testing is generally done only once at the beginning, before actual implementation work is done.

2.1.2 Api conformity

Some filesystems don't focus on the way how to store data on media but how to make them accessible. Well known group of such filesystems are network filesystems. They suppose that some other filesystem does the storage, and they specify only the way how data will be accessible remotely, and put some restrictions on the filesystem behavior. In this case, we test the particular implementation if it is conform to the specification. For example, in case of NFS there are test suites for checking interface stability, protocol conformity.

2.1.3 Functional testing

Filesystems have many things in common with normal pieces of code, such as server or desktop applications. But in the means of testing there is big problem in simulating normal environment for filesystems. This is caused by their low level nature and could be one of key reasons why test suites for filesystems tends to use black box testing.

2.1.4 Benchmarking

Benchmarking gives answer the question "how long it will take" for every operation of filesystem. Measurements are done on different filesystem implementations, or filesystems with similar purpose. There should be similar setup for all subjects tested. Most of benchmarking tools assumes that the implementation is sane and doesn't do any invalid shortcuts. Their goal is to compare more implementations or filesystems.

2.2 Test format

When tests are expected to be executed only manually, the format could vary. On the other hand, when they have to be executed automatically, then for every format there must be support in all components of the test suite. Because of this, there is tendency to minimize the number of formats. This apply on both language of tests and interface for tests themselves.

The basic choice is to write tests in native language of the application. Sometimes, embedded tests within normal code is supported with some flags saying, "this is test code, it should be executed when testing". This allows having tests as close to code as possible. It is ideal for short tests of functionality of small parts (functions, objects, etc).

For automated testing, scripting languages are often used to write either control logic, or everything including tests. Scripting languages are ideal for the logic because of their flexibility. The reason for writing tests in the same language as control component is that it makes integration easier.

Another possibility is to have tests in some proper format. This offer possibility of having the format very suitable for the needs of particular software, but brings disadvantage of having to change the format every time new requirement is found.

Some testing tools use configuration (tests) in XML, or XML with embedded code. Main reason for XML is possibility to use external tools for editing tests, or for XML-based transformations. On other hand, XML is very unsuitable format for hand written code, and the DTD of configuration is often hard to understand.

2.3 Filesystem test patterns

In this section, main functional filesystem testing tools used in UNIX-like systems are listed and described.

2.3.1 FSX

Originally written by Apple Computer, Inc. and BSD-style operating systems. Nowadays, there are more versions used, but the main part stays the same [21].

Test consists of single source file written in C, compatible with most Unix based operating systems. Test operates on one file, does loop with random operation in every cycle - one of read, write, truncate, close and open, map read and map write. Memory mapped operations can be disabled. Checks in FSX are made by comparison of write buffer and data read, additional

checks of file size are made too. Failure report from FSX is dump of operation sequence and buffer dump.

2.3.2 LTP

Linux testing project [8] is collection of test suites for Linux operating system internals. Tests are compiled programs or scripts, driving is done by control script. The part dedicated to filesystems contains previously mentioned FSX (Subsection 2.3.1) and own filesystem testing binaries. Checking is mostly done by comparison between results and expected values. There are both types of tests: predefined loops with random arguments (file sizes, etc) and stress tests consisting of random sequences of operations. No cleanup is provided. Failure log contains only arguments to last tests. There is vast amount of operations implemented.

2.3.3 OpenSolaris ZFS / NFSv4 Test Suite

Very exhaustive test suite for filesystems on OpenSolaris [9]. There is two sets of tests: one for NFSv4 and one for ZFS, but the techniques used in them are the same. Only difference is that ZFS test suite have extra stress tests.

System is Makefile driven, tests are generally shell scripts (ksh) with few compiled support programs and tests. Testing is deterministic, cleanup is done after set of tests (directory). Errors are printed to stderr.

What should be noted is the coverage of these tests, there are tests for nearly every operation possible, even for zones, ACL or redundancy. In ZFS part, there are simple stress tests too - predefined loops with configurable length.

2.3.4 Mongo

Mongo benchmark [43] is test program written in Perl, aimed to Linux filesystem performance and functionality testing (developed by Hans Reiser for reiserfs). Is very tightly coupled with standard tools and filesystem usage. Does everything from mkfs, through mount to classic operations. The version examined was mainly benchmarking tool, there were no checks if things goes well, just if they goes.

2.4 Unit based testing frameworks

Unit testing is based on Kent Beck's testing pattern [23].

2.4.1 Common principles

Every **test case** is executed separately, test case have common interface (in object based languages presented as common super class). The run of single *test case* should be independent on other test cases.

Test case may have **fixtures** - methods to set up environment before test and clean up after test. These methods are very often called *setup* and *teardown*. Teardown method is executed regardless of result of test.

Expected problem in test is called **failure**, non anticipated problem is called **error.**

There is **common method of testing** if expectations hold. In smalltalk by using *should* and *shouldn't* blocks, in modern languages by using *asserts*. When assertion doesn't hold it is called failure. Errors are mostly represented by exceptions.

Result of test is a *result* object.

*T*est cases are aggregated to **test suites** that can be aggregated too.

All test cases in test suite are **run recursively** by calling *run* on root test suite. Returned value is aggregated Result object.

Unit testing should be **automated**, independent on human interaction.

2.4.2 Other features

Many unit testing frameworks offer more **elaborated tests structuring and state handling** (fixtures). Very often, tests can be aggregated to classes with common setup and teardown methods that are run before and after every test.

Moreover there can be **additional fixture levels** for all code units (depending on programing language these can be class level fixtures, module level fixtures, package level fixtures, etc.). *Setup_* code is run before entering particular block, and *teardown_* code is run after leaving particular block of tests. For example *setup_ class* is run before running tests in class, and teardown_class is run after all tests have run. Note that this could break the independency requirement. There may be *setup_ method* and *teardown_ method* fixtures to execute around every test method.

2.4.3 Best practices

Unit testing is mostly used for **testing small pieces of code** and thus use cases are mostly very simple and execution of test fast.

Unit testing is very often used to watch for regressions, so all tests should be **executed periodically** by some automation tool.

There should be **100% code coverage** done by unit tests. Every function (method) should have at least one test case, class should have test suite as

counterpart.

2.4.4 Implementations

There is at least one unit testing framework for every programming language (see [17] for short list).

This chapter will focus mainly to these written in C (language ZlomekFS is written in) or python. Aim on python is given by choice of language for driving component, for reasoning see 2.2 and 3.1.

JUnit [35] is unit testing for java. This framework is interesting just by it's author: *Kent Beck*, guru of unit testing. Has standard features as test level fixtures, assert methods and aggregation. In last version (JUnit 4) there were added class level fixtures, timeouts for tests, expect exception annotation, and requirement of inheriting from base class (TestCase) was removed. It is not part of standard language distribution.

Python has **Unittest [42]** as it's standard tool (was called *PyUnit* [41] before integration to python standard distribution). The interface is strongly object oriented, test must inherit from TestCase class and override specific methods. Doesn't offer more levels of fixtures. It is very pure reimplementation of Kent Beck's original smalltalk framework.

Py.test [12] is alternative python unit testing framework. It is part of *py.lib* library [32], has fixture support on all levels, doesn't need to inherit from superclass, but has fixed naming convention instead. Uses standard python assert clause to test for failures, handle exceptions as errors. Moreover, py.test has automated test discovery tool for searching for tests in directory trees. Test classes can be marked as conditionally *disabled* depending on generic boolean expressions. This library has support for generator methods which *allows to yield more tests* easily. Py.test most interesting feature is ease of use. It is possible to just write function with *test* in name and py.test will collect it, run, and if there is failure or error, the output and backtrace will be printed in readable format. Py.test can also take big advantage from py.lib which offers distributed execution through *py.execnet,* etc. Whole py.lib is written to be easy to use, but in current implementation with trade off configurability. Note should be taken, that py.lib was developed as grant project, and after grant expiration there were little of improvement in the project.

Another unit testing framework for python is **Nose [40]**. Offers backward compatibility with standard *unittest*, some compatibility with *py.test* library, and try to mimic *py.test* without magic. Nose provides all features of unittest, moreover it implements py.test's all level fixtures, tests doesn't need to inherit from superclass, has automated test discovery tool, and uses generators. In addition, Nose is very configurable. It has in build support for changing naming

conventions. Tests can have flags, and it is possible to define expressions which tests should be run according to these flags. Nose has extensible api with plugin support. There are for example plugins for profiling, doctest, code coverage, etc.

Curiosity among unit testing frameworks is **MinUnit** [25], which is C based, and consist only from three lines of code (two macros and one definition). Doesn't offer much, just assert - print message block.

CUnit for Dr.Ando [20]claims to be easy to use C based unit testing framework inspired by *cppunit-x* (Interesting piece of code documented in Japanese). In fact, it is just another framework which lacks fixtures and offer just test counting beyond *MinUnit.*

Simple C++ Testing framework [34] is written whole as macros, and have somehow weird syntax. Offers basic assertions and test suites. There is no need to write main function listing all tests, but this is achieved by wrapping all tests to macros START_TESTS and END_TESTS. Because of this, tests must be written in one big chunk. Again, tests files must be compiled and run by hand. Runs equally in pure C and C++ environment.

CxxTest [47] is C++ based, all tests have to be wrapped to test suite Classes. This framework has assertions, fixtures, and handle exceptions. Automated collection is done by python script (simplified C++ grammar is used). May have problems upon linking with pure C based code. CxxTest have support for mocking global functions, but this support works on base of calling functions in separate namespace, so it is not pure mock, and code have to be modified to use mocked functions.

CppTest [37] is another C++ based unit test framework. Has basic features such as assertions, fixtures, and test suites. Beyond this, CppTest is capable of handling and formatting output, offers api for writing output formatters (TextOutput, CompilerOutput, and HtmlOutput formatters are implemented). As for CxxTest, pure C sources must be modified (headers wrapped in extern C block) to run under CppTest.

CUnit [36] is C based (still C++ compatible), supports assertions, suites, test counting, has global registry, more user interfaces (for running tests), but has no automatic collection and output handling.

GUnit [31] is another unit testing framework. Uses GTK+ libraries (for almost anything). Supports assertions, suites and fixtures. Has gnome and hildon (embedded) GUI, dedicated logging facility. Doesn't offer discovery, however compiles suites as dynamically loadable libraries.

RCUnit [28] (C based) supports assertions, suites (called modules), fixtures, has own logging facility, tests can be disabled. RCUnit has documented interface for writing output handlers. HTML and plain text handlers are implemented.

Cutee [22] tends to be as simple as possible. Thus supports only assertions, no fixtures or suites. Tests are collected automatically, yet files with tests must be listed in Makefile.

Check [38] provides assertions, suites, and simple fixtures. Forks every test in separate process, can handle timeouts, output can be printed in plain text or XML. Has no build or collect helpers, adding test is very annoying.

CuTest [33] from basic features provides assertions and suites. Has scripted tool for executable generation.

2.5 Logging, tracing

2.5.1 Models

When an error is detected in software, developer needs to have as much information about the failure as possible. What occurred is nice to know, but in most cases useless without more details about circumstances. Therefore, developers use logging and tracing to get some useful information about the particular run.

By tracing, we mean storing information about call sequence in the program. By logging we mean saving information about data changes, or notes about states of system (inserted by developer). In most cases, these features are provided by one tool.

The simplest logging tool used is insertion of **direct message prints**. Messages may provide the information needed, but this approach suffers by not having centralized control of what has to be printed. This leads to excessive logging, in which is hard to find useful information, and if we want to avoid this, it force us to changing the code on many places.

So next logical step is to send logging messages (accompanied by importance level) to some **centralized facility**. The importance level list is in most cases directly given in advance. Providing this, it is possible to change the amount of output centrally and even redirect messages to distinct places.

When simple distinction by importance is not enough, more advanced logging facilities come with **tagging of messages**. Tags could be flat or of arbitrary structure. This allow better filtering of messages of special types.

Other approach to logging is to have more than one logger. In this case, the tool has frequently **producer - consumer** based architecture and loggers are organized to dynamically created hierarchy. This ease goal of having different output locations for different types of messages. On the other hand, the architecture is not so easy to understand for anybody who might contribute to the code. Moreover, with more people participating on development, it is nearly impossible to keep the hierarchy of producers and consumers used in

application sane.

The last approach to logging and tracing is called **aspect oriented programing**. In this case the logging is not present in code itself, but it is separated as independent concern to aspect - logical definition what and where has to be logged.

2.5.2 Pitfalls

Even if a adequate logging tool is used to debug the software, there can arise problems when the tool is used under some automated stress testing. The amount of output **logs would eventually grow too big** for storage capacity, or at least for the potential reader to deal with. So the automation tool should be able to communicate with the logging facility and dynamically change the amount of output according to actual needs. This must be tuned to throw away the biggest possible portion of unrelated logs, but to preserve the crucial information for debugging the failure as the failure could be hard to repeat. Other possibility (commonly used) is to have circular buffer which holds only latest logs.

There is one more reason that may be considered for muting logging output. The reason is that **logging could slow the application down**. To check how much logging slows down a ordinary application, some measurements were done.

For testing was used real application - session server from the SUCKS [45]project. This application is threaded and accessible by network. Logging facility was simple centralized logger with predefined log levels. Logger was alternated so it measures time spent by logging. Tests consisted of predefined workload, output was time spent by whole application, time spent in logging and characters printed. Test cycle was composed of one run of all tests for every log level and log target. After finishing, the cycle starts again. This had been running for approximately thirty hours on two platforms:

1. Intel centrino with core2duo CPU, set to static frequency of 1Ghz with 2GB memory (most unused), running kernel 2.6.20.1 x86_64.

2. Motorola ppc MPC8241 (266Mhz) with 128MB memory running kernel 2.4.32

Results are summarized in Table 2.1 on the following page. First column is logging target (medium to which logs are written), second column is actual log level. First column of platform dependent part is how many microseconds are (in average) spend for printing of one character of log output. Aliquot part of function calls, conditions evaluation and auxiliary operations done by logger

Log target	Log level	MeanTimeToChar (µs)		MeanTimePercentage	
		ppc	intel	ppc	intel
Discard	0-7	0.000000	0.000000	0.004898	0.007112
Memory	0-2	0.000000	0.000000	0.004845	0.007102
Memory	3	0.937382	0.259296	0.016400	0.010398
Memory	4	0.551059	0.120269	0.024276	0.012704
Memory	5	0.429199	0.087316	0.029717	0.014125
Memory	6	0.429590	0.088023	0.029792	0.014239
Memory	7	0.429620	0.087709	0.029718	0.014188
SHM	0-2	0.000000	0.000000	0.004946	0.007323
SHM	3	1.101189	0.264383	0.018914	0.010602
SHM	4	0.636369	0.124098	0.027536	0.013109
SHM	5	0.484440	0.091811	0.032791	0.014852
SHM	6	0.535813	0.091781	0.036307	0.014847
SHM	7	0.485906	0.094485	0.032755	0.015284
File	0-2	0.000000	0.000000	0.004820	0.007141
File	3	1.018307	0.272918	0.017675	0.010944
File	4	0.599782	0.129446	0.026131	0.013674
File	5	0.463753	0.109439	0.031663	0.017704
File	6	0.464099	0.105559	0.031701	0.017076
File	7	0.464440	0.105147	0.031581	0.017009
Console	0-2	0.000000	0.000000	0.004870	0.007182
Console	3	8.009316	5.422122	0.137280	0.217427
Console	4	6.347917	5.787681	0.277779	0.611372
Console	5	6.346556	6.042242	0.420388	0.977433
Console	6	6.259989	6.055290	0.414675	0.979544
Console	7	6.448019	5.952686	0.426063	0.962946

Table 2.1: Mean logging load to application

are included. Second column is percentage from running time of application, that were spent by logging (in average).

Both platforms behaved equally, the only difference was in speed (and for console prints, we must consider, that Motorola was connected by network and all console prints must went through ssh). From results we can see that all logging targets had the same footprint and the only *slow* target was console write. For non-blocking targets, the slowdown was in hundredths of per cent for all log levels.

Conclusion of this (regarding to speed) is that when we don't need to read the output of application online we can log everything. In case of small storage capacity could be used circular buffer which can be flushed to file only when error occurred. Requirement is that filtering must be possible (Filtering can be done afterward by user). Another finding is that on non-blocking media logging footprint is in half made by checks if something has to be logged or not. Thus if minimum slowdown is required, logging should be entirely removed from binary in compile time. Problem with this is that it makes changes to binary image, and these changes can lead to different behavior of erroneous code and

the bug could be unreproducible with different logging level.

Other problem, which can arise with logging is that logging can act as synchronization primitive, which could prevent some collisions to appear. Problem with synchronization can be solved by design of logging facility. The logger must be designed in way that creates separated resources for every concurrently running entity in advance, and then the only effect done by the logger is slowdown upon creation of new *threads*.

2.6 Presentation layer

Usual test run produces outputs of many types, beginning with standard outputs, going on with filesystem changes, debug logs, even including state snapshots, core dumps, etc. These are data of very different types. The goal is to present them to user in usable way with structure that can be easily understood, and through ways, which are accessible from as much environments as possible.

2.6.1 Raw data

The easiest way to present results is to leave them as they are produced by application and test suite. When this approach is used, raw data are often made available for downloading through simple protocol such as FTP or remote console. Raw data hold always full information, don't suffer by any losses from transformations. On the other hand, raw data are often platform dependent and interpretation on the system that have produced them may be required. Raw data should be in standard format to allow reading with external tools.

2.6.2 Web interface

Dynamic web pages are nowadays very used way of presentation, as they are relatively easy to write. Web pages have big advantage in accessibility - nearly every computer have web browser installed and people are used to get information through these. On the other hand, web pages can hardly display some debug outputs, such as core dumps, and other binary data. In this case, data should be downloadable for reading through external tools. Interaction with web is little bit slower than with local application and user comfort is worst too.

2.6.3 Special application

Even raw data must be interpreted by application to be presented in readable form. When there aren't appropriate general tools, they should be written as part of test suite. The fact that they must be written is one big disadvantage by itself. Full featured interpreter of debug data with presentation layer may consist of the same amount of work as test suite itself. Moreover, requirements and dependencies of such application could be non-trivial and platform independency is hardly to achieve with this approach. The big advantage of special application is that as written specially for a suite it should fits very well the needs.

2.7 Random workload generation

Random workload for stress testing must be generated from small tests (operations, meta tests, atoms). Depending on subject tested, atoms are either defined by tradition (basic operations that can be made on subject tested), or small, well defined tests. It is sometimes wanted to group some atoms to create new (bigger) atom.

In ideal case, run of meta test may not change state of subject tested. But this type of atoms can test only stateless systems and operations which are less erroneous thus not so big candidates for regression testing as state-full systems are.

For state-full systems (and meta tests) there should be method how to define and check states and legal transitions. Simple method to allow this is to give to tester way to define pre and post run hooks that can initialize state, check transition, and possibly do cleanup after test. While approach with pre and post run hooks is simple yet powerful, there is one issue connected to it. In this system, tester can use state-full tests, but in trade of possible waste of resources. When test expected some state different that in which system is, it must either made a state change (non trivial operation out of test scope) or silently pass without testing and let the system run other test. More sophisticated system for resolving state-fullness is to give tester tool to define allowed transitions between tests. Transitions are often given by graph (edges can be allowed transitions or tests).

Sometimes it may be desirable to give some preferences (what should be tested). For random workload this means either switching meta tests on and off, or giving test preferences. When state-fullness is not solved or solved by pre and post hooks, percentage is connected to tests. When transitions are used, percentage can be either for tests (implicit edges) or for transitions.

Length of continuous random workload can be simple defined as:

- number of meta tests (minimum, maximum, mean)
- time (resources) used
- by transitions to end point

When user preferences are given, system itself can be simple automata running on state-full graph.

2.8 Pruning output

When long test (or random test) fails it could not be clear which step has caused the failure. So test outputs (debugging info) are needed to locate the bug. On the other hand, output from long and random tests can be huge (and most of it useless). The goal of pruning output is to provide enough information to find the bug, and at the same time hide useless ballast.

Basically, outputs to developer can be divided to log messages, run back-trace, and system state snapshots (memory dumps, filesystem state, etc).

Data automatically generated is more resources consuming, but as storage capacity is cheap, we can simply leave all snapshots, or limit space used by constant and delete old snapshots. On the other hand, we must avoid excessive slowdown (which in case of core dumps is non-trivial).

As for logs, the problem was described in Subsection 2.5: when logging doesn't slow application down and doesn't change behavior, the best approach is to log all, store all (or latest), and provide tool for filtering and searching logs. With this approach, no crucial information is lost by heuristic pruning. In special cases, like low resource platforms (without storage, extremely slow, etc), where wasting can't be afforded, some heuristic must be used. For system state this can be the state in time of failure.

For logs there can be more approaches which can be divided to:

- **on-time pruning** - test suite changes log level of application according to probability of failure. The question is, how it should know.
- **afterward** - log level is constant for test run, logs are stored to cyclic buffer. When failure occurs, test suite will trim the buffer to store just useful information.
- **re-run** - tests are executed with logging on minimal level. When failure occur, test suite will rerun the test with more logging *around* the failure, possibly skipping some parts of test (as for random generated workload).

2.9 Checkpointing

Some failures are hard to reproduce, thus developer wants as much information about the faulty run as possible. Beyond logs, state of application in time of failure can be required. That is why some snapshot creation support is useful.

Moreover, as the sequence to failure can be very long, the test suite may try to repeat just short parts of them, or skip some steps to find the shortest possible run to bring about the bug. For reruns it is best when the second run has the same start conditions. The snapshot (checkpoint) can help with it by resuming from stored state (if possible).

There are many projects trying to create **full featured checkpoint / resume** support for applications. They can be divided in two groups: user-space only tools and kernel-based tools. The main problem among them is that none of them have full support for every resource an application could use. The most frequently missing features are suspend / resume support for: networking, devices, threads, signal handlers, shared memory, shared objects. Some of them (BCLR [1], CryptoPID [24], Chpox [44]) seems to have everything needed, but for the price of many constraints and dependencies. In general, these projects are useful only in cases when the snapshot / resume is vital for application or system itself.

Other possibility is to use some **virtualization** tool and run application (not necessarily test suite) inside virtual machine. Nowadays, there is many virtualization tools with support for snapshoting (for example openVZ [10], Vmware [18], Xen [19], Qemu [14]). However, working with virtualization is fairly complicated and we can't test hardware dependent issues on them. Moreover, creating snapshots of whole system can be very resource-consuming.

The last and easiest possibility is to use just **snapshoting without resume** and save the snapshots in some easy to read format. For example GDB gcore [6] command (which creates gdb core dumps) can be used to snapshot the application.

2.10 Continuous integration

Existence of tests is not enough to provide stable development cycle. For stability of project it is vital to test it for errors (run tests) as often as possible, preferably after each change (commit). The approach when **changes are often merged** into *mainline* is called Continuous integration (good overview of this method is in [27]). To achieve regular testing, it is convenient to use some tool (server) to automate the process. There are many proprietary solution and even more opensource solutions. It is interesting how many organizations deploy their own systems (for example Mozilla Foundation uses Tinderbox [16],

Redhat has Frysk [5], ThoughWorks has [39]. Apache foundation has even two projects - Continuum [3] and Gump [7]).

2.10.1 Automation

Best approach to achieve stable build and check cycle is to automate it. Some projects use manually driven systems, but there is hazard of human failure (developer could omit tests, forgot about them or ignore them at all).

In common there are two approaches used to achieve automated build and check:

First is to use some **post commit hooks** (we assume that version control system is used) which executes tests, or launch separate process to execute them on background. This approach can ensure that no wrong code gets into repository - commit that won't pass all tests can be rejected.

Running all tests can be relatively long lasting task and therefore environment is rarely set to execute all tests before commit. Commonly, commit is delayed only after vital tests pass and more in-deep checking tests are executed afterward in stand-alone process. Implementations of this approach are often bound to one version control system.

Other option is to not check validity in commit hooks, but use **independent service** which monitors state of repository and runs tests either for every change (commit), or on regular basis (night builds). Benefits of this approach are that it doesn't slow down commits, and tools using this approach have frequently more features and better configurability. These solutions are generally independent on version control system (support for distinct version control systems is provided by plugins), but sometimes use hooks to get alert upon change.

Often this combination is used:

- pre-commit hooks are used to enforce repository rules and coding conventions
- separate service builds project (upon commit or nightly), and executes all tests checking if change doesn't break something

Because one of build steps can be building the source, there should be support for **build system** used by project. Again, many tools are plugin based and have plugins for most common build systems. Particular set depends on aim of the tool.

Pure commit driven environment when commits are delayed after all tests passed doesn't need presentation layer at all, there is just the message send to commiter. But as this variant is rare, nearly all automation systems have **presentation** layer. The complexity vary from simple text (HTML) file statically served by web server to rich database backed GUI. Standard tools offer HTML overview and detail pages, tools bound to specific environment usually provide GUI for that environment. Very often, there are email or instant messaging notifiers too. If there are no other output than test count and results, it tends to be plain text.

2.10.2 Distributed testing

Testing doesn't need to be run on the same machine as control service. When tests doesn't run on the same machine as main continuous server, we call it distributed testing.

There can be many reasons for distributing:

It is crucial do distribute testing for **multiplatform applications**. Wrong code can behave badly only on one of target platforms, and therefore testing on only one platform can left errors undiscovered. In this case distributed testing is the only way to cover specifics of all platforms. It is possible to run one separate machine with full build and check service for every platform, but managing such system is huge overhead, and collecting results is non trivial too.

Some tests or projects could be **dangerous** to system itself, or the project is **part of base system**. These should run in separation of production system (to not break it), in some sort of sandbox. One possibility is to run dangerous tests on separated physical machines. But this would lead to non-trivial problems with recovery. As virtualization is nowadays easy to deploy, the best way (and in case of base system parts the only viable way) is to sandbox project testing in virtual machine. This must be considered as distributed testing too because the communication with virtual machine must be done in the same way as if the virtual machine were in other network. There are in general less security barriers between the host machine and guest machine but it don't affect the connection method itself. For some special cases there can be one more option how to separate dangerous tests without distributing: to use operating system provided tools to restring their privileges and resources (for example chroot).

Another reason for running tests on other machine can be **resource consumption**. As the control system should be visible to wide network (at least the presentation layer), it is frequently run on production server which hosts

other applications too. In this case it is not good idea to slow down or even block whole system by testing. Again, tests are given to another machine to execute.

Sometimes tests takes long **time** to complete. Then it is convenient to spread tests over more machines, each running only part of tests. By this we achieve shorter build and check cycle.

Distribution can be achieved by:

Sending commands through **remote terminal**. For example on UNIX, data could be copied to target machine by scp, tests executed through ssh, and results again retrieved through scp. This is the easiest way used in simple cases where no synchronization or overview is needed. Data in general doesn't need to be delivered through the same way as commands, clients can fetch them themselves upon test command, or there can be shared network storage where control server should put data for tests.

Most common way to connect control server with machines executing tests (sometimes called slaves or bots) is to use **remote procedure call**. As remote procedure call not only RPC is considered, but any method that allow calling procedures on remote system. There are many RPC tools such as RPC, CORBA, dcom, python twisted, py.execnet, etc.

Older systems tends to use **e-mail** communication. It consists of specially formatted messages sent between master and clients. This approach has many drawbacks as security problems, frangibility and non-deterministic behavior.

Sometimes, **own methods** for communication is used that mimic remote procedure call by sending data within special protocol.

2.11 Sandboxing

When tests need more privileges over hosting environment, or the tested component itself is part of operating system, there is big probability that **running tests can broke something**. In this case, tests must be executed in separation (so called sandbox). It is either part of system with restricted access to some resources, or whole separate system.

Obviously, there should be possibility to simply create new sandbox or to restore previous state of sandbox after breakage from test.

As we described in Subsection 2.5, execution logs are crucial to track failure (bug) found by automated testing. Therefore it should be possible to get logs (and other data) from sandbox at least after failure, but preferably to send them back to master straight upon generation. This is mostly feature of logger, but to use it, the sandbox must allow communication with outside.

When testing is distributed (see Subsection 2.10.2), it should be considered to use the remote machine as sandbox too. Again, there should be method to easily restore state of remote machine after breakage from test. Note that this would be problematic with real (non-virtualized) hosts.

In Subsection 2.9 we have analyzed problem of providing information about test state to developer. If sandbox is represented by full operating system then checkpointing of whole sandbox would be in most cases big overhead. Still it should be taken into account in some cases. One case where sandbox snapshots may be convenient is when test depends heavily on system state or changes system state. Then without system snapshot, some information to track failure may be missing. Other case when full snapshots may be generated is when test has caused system failure (therefore normal snapshots can't be created).

Chapter 3

The test suite architecture

3.1 Programming language

Since ZlomekFS tends to be multiplatform and support more operating systems (currently only Linux is supported) language which runs on any platform (or at least under UNIX-like operating systems) is needed. For testing, the language should be flexible enough, but on other hand, since file system will be tested, speed must be considered too. Last but not least need is that the language should allow integration with existing code.

Because of this criteria **python** [13] was chosen as main programing language for driving component. Python can integrate with most compiled languages and thus not all components must be written in python. Performance critical parts and integration libraries can be written in ZlomekFS native language - **C**.

3.2 Used tools

3.2.1 Testing environment

For very specific purpose of testing distributed file system, no suitable existing solution was found. Instead of development of new system, it was decided to tweak existing tool (if possible) to fit the needs. From general purpose testing tools written in python, **Nose** [40] was considered as the most suitable. Moreover, every feature needed which Nose doesn't implement can be delivered as plugin without modifying Nose core code. Nose plugin architecture was found as flexible enough for further extending of the test suite too.

3.2.2 Continuous integration

When choosing tools for continuous integration, only opensource projects with reliable community around it were considered.

Buildbot [48] was chosen as continuous integration server.

It was chosen for these reasons:

- Compatibility with ZlomekFS build system. Buildbot can use shell commands and python code as build steps. It can do output parsing for configure, make and gcc output too.
- It is small and easy to deploy. There is no need to write big XML configuration files to run *hello world*, buildbot code is relatively small and easy to read. In comparison with others, buildbot has around 500K, where build of CruiseControl can take more than 300M.
- Written in python. Since whole Buildbot is written in python, we can easily integrate it with other tools used in project (as they are written in python too).
- Extendable architecture. Buildbot parts are written as objects, so it is possible to inherit from them and tweak behavior according our needs.
- Support for distributed testing. As ZlomekFS is intended to run on multiple architectures (and in future possibly on multiple operating systems) the support for distributed testing is essential.
- Active development. Buildbot is often used and has active community which ensures that it will be maintained in future too.

3.2.3 Web result presentation and result repository

Test results in buildbot are presented as textual output from commands. This is not enough from two reasons: when failure or error is found, the textual output may be messy (yet useful in some cases). Bigger problem is, that it can't provide enough information, and binary data can't be provided in this way at all.

So there was need for another way to represent test results. Since test outputs can contain binary data (snapshots, core dumps, filesystem state, etc), the presentation layer should be able to distinguish several types of data and present them according to type. As data and information for test run are related to each other, database driven storage was preferred.

The solution used was **Django** [4]. It is written in python, offers object oriented database abstraction layer, web pages generating tools. It's configuration format is pure python, so it is easy to integrate it with other parts of project.

3.2.4 Logging

We need logger with this features:

- it can be controlled externally
- it has simple still full featured interface
- it has to have implementations in both languages (python and C)
- the output has to be in parseable and user readable format

This enforces us to write **new logger**, which suits best the needs. From models, we can't use aspect oriented logging, as there is no implementation of aspects for both languages (and moreover it will be big requirement for developers to learn aspects). The producer-consumer model seems to be too complicated as there will be large and non-homogeneous group of developers working on ZlomekFS.

Thus the logger will be centralized, supporting tagging with fast evaluation. The output will be redirectable to shared resource (shared memory, network socket) which can be used and controlled by test suite. The format of written log for failure may be preferably readable by some GUI or web based reader such as Chainsaw [2], if not possible, user readable output should be used.

3.2.5 C based unit testing

All existing C based unit testing tools have one of two **major problems** (or both).

- Many tools are **very complicated** and writing simple test for "a + b" could take five minutes.
- C based unit testing tools **lacks automatic test discovery**. Thus tests must be listed somewhere and collected manually (even in case of hierarchies). The best is heuristic search by grep.

Since external are non-trivial for integration to project and writing of simple C based unit testing is relatively easy, **new C based unit testing library** was implemented.

3.2.6 Documentation

For documenting C code **DoxyGen** [46] was chosen. Main argument was that ZlomekFS is documented in DoxyGen. Secondary, DoxyGen is nowadays nearly standard tool for documenting C code.

For Python code, DoxyGen has some support, but the support is problematic, needs usage of impute filters such as Doxypy [29], and even then results are not ideal.

There is standard docstring format for python [26], but it doesn't support more than plain-text formatting.

Another option is to use some non-standard documentation tools for python (most of them are listed in PEP256 [30]). Their problems are mainly listed in mentioned PEP256.

Finally, **reStructuredText Docstring Format** [**?**, 15] was chosen. Main reason was that Nose [40] uses this format, secondary reStructuredText is easy readable in **Pydoc [11]** output and there are HTML formatting tools too.

Chapter 4

Implementation details

4.1 ZlomekFS changes

4.1.1 Logging

Original message printing system (two levels, hardcoded values) was replaced with **new logger** (*Syplog*), developed specially for ZlomekFS.

Logger supports eleven **log levels**. Log level is 32bit unsigned integer (typedef), log levels are defined by macro constants (from LOG_EMERG to LOG_LOOPS). There is conversion function available to convert log levels to user-readable strings.

For concern separation, logger distinguishes **facilities**. Facilities are distinct one-bit flags which can be OR-ed. Log message can be labeled as belonging to any number of facilities. Again, there is conversion function to convert facilities to user-readable strings.

Log levels and facilities can be used to **filter** what types of messages (of which importance) should be recorded and what should be discarded. Both log level and facility filtering set can be altered in run-time.

Output from logger can be written to file or shared memory, api is open for extension such as socket write. Format of output files can be user readable strings, or raw memory dumps. For all formats and output targets, there is both writer and reader support, so transformation between formats is trivial.

Settings are read from program arguments (array of string with given array size). Logger ignores unknown options, so direct configuration from command line is possible (and currently used).

Integration with external driving component (Nose testing environment) is done through **D-bus** control api. Both log level and facility set is adjustable through D-bus. For further integration there is full-featured python wrapper for syplog generated by swig.

Log level and facility sets can be **extended** by listing constants for new log levels and facilities in header files. Output formats and targets are defined by static structures holding pointers to functions with specified behavior. New formats and targets can be provided by implementing given function set and providing description structure. For further reference, DoxyGen documentation of *Syplog* library should be used.

4.1.2 Control component

The driving component needs to know in which state ZlomekFS daemon is (starting, running, terminating). To accommodate this need, **D-bus** listener was added to zfsd.

The D-bus component of ZlomekFS consists of two parts:

- D-bus **provider** - state-full service which manages initialization of D-bus, listening loop and termination of D-bus connection. Doesn't serve messages.

- D-bus **message handlers**. Set of functions describing zfsd specifics - naming and signal handlers.

Integration of these two components with zfsd daemon is following:
On beginning, zfsd creates and initializes D-bus provider handler (structure). Then, it registers zfsd D-bus message handlers by calling *dbus_provider_add_listener*. Currently, the syplog D-bus service is implemented in way compatible with zfsd D-bus provider, thus another *dbus_provider_add_listener* is called for syplog service. After all needed listeners are registered, zfsd calls *dbus_provider_start*. This starts new thread which listens for messages and forwards them to registered handlers. Finally, when zfsd is terminating, *dbus_provider_end* is called.

For future, there are plans for **remote zfsd control** mechanism. Main intention of it should be to allow user initiated synchronizations, mode changes (slow connection, fast connection), etc. These should be implemented either as another set of message handlers or by extending current zfsd message handler.

As the test driving component is written in python, there is **python client module** for this api too. Client module is automatically generated by swig, so there should not be problems upon extending the service.

4.2 Testing environment

To allow central result repository, all tests are executed under **Nose**. Settings for plugins can be given by command line options, but preferred way is to store them as **environment variables**. Environment variables are used to pass values between components too. This way is used to ensure all tests within one BatchRun will be reported properly. To ease repeated runs with the same configuration, support for **profiles** is implemented. If environment variable *PROFILE_NAME* is found, system will try to load module with given name and read environment variables from it.

If **zfsReportPlugin** is enabled, results are reported to Django driven **TestResultStorage**. Settings for TestResultStorage are read from module which name is in environment variable *DJANGO_SETTINGS_MODULE*. Default place for it is *TestResultStorage.settings* (on most systems in */usr/lib/python/site-packages/TestResultStorage/settings.py*).

4.2.1 ZfsTest

Zfsd control from python is implemented in *ZfsProxy*. This class provides methods for starting zfsd, status queries, *syplog* control, sane zfsd terminating and locked daemon killing.

Normal **ZlomekFS tests** should inherit from class *ZfsTest* which implements basic fixtures for zfs. New filesystem daemon is started for every test to run.

Stress test classes listing meta tests should inherit from class *ZfsStressTest* which runs all tests on single instance of zfsd.

For **distributed testing**, there are wrappers for *ZfsProxy* and *remote file* objects. Communication is done through *twisted perspective broker*. Twisted were used as RPC because there is direct support for it in *Nose*.

Example distributed test is represented by *TestClientServer*. On remote system used in test there should be *remoteZfs.py* script running. It runs twisted reactor and listens for incoming connections.

TestClientServer is stress testing meta class, thus all setup is done in *setupClass* method. Configuration for local and remote *ZfsProxy* is stored in *zfs_client_server_config*, configuration for ZlomekFS daemons is in *localZfsMeta.tar* and *remoteZfsMeta.tar*. These options are provided to test by *ZfsConfig* plugin.

Local *ZfsProxy* is initialized in the same way as it would be in case of non-distributed testing. Remote *ZfsProxy* is initialized through calls performed on wrapper objects. Tests then opens local files as usual and remote files by calls to *RemoteControlWrapper* object.

In this manner, any topology of zfs providers and clients could be made. There should be *remoteZfs.py* running on every machine used in testing. On machine where *Nose* will run, there should be configuration tarball for every client machine and configuration file for *ZfsConfig* plugin. Test then connects to all machines, configures their zfsd and operates on local and remote files.

Synchronization of versions of libraries and scripts must be handled externally, currently *buildbot* is used.

When failure is detected, state of both local and remote ZlomekFS daemon is provided by snapshot method.

Infinite testing loops are handled by insecticide in the same way as other tests. Only difference is, that there should be special profile for infinite testing, since normal testing should not run for such a long time.

4.2.2 Failure state data

ZfsProxy class sets environment for zfsd to create core dump upon crash. If crash is recognized, this core dump is collected and appended to snapshot. If zfsd is running while snapshot is created, core dump of running process is created instead.

When test fails or error is detected, **snapshotPlugin** creates snapshot of predefined components. Snapshot plugin can be configured to create snapshots before tests and after successful tests too, but while zfsd core dump has more than 150Mb, it takes non-trivial amount of time to create it (and thus it is disabled by default).

By default snapshot will consists of:

- snapshot of zfs cache (filesystem)
- snapshot of filesystem to which comparisons are made
- zfsd log output
- nose log output
- test instance
- ZfsProxy instance
- zfsd core dump
- zfsd stdout and stderr
- python exception and backtrace (if any)

Developer can specify any **further data** to include to snapshots by overriding *snapshot* method of test class. Method gets *SnapshotDescription* instance as argument. *SnapshotDescriptio*n has methods for appending primitive types, python objects, files and directories. Every entry in snapshot have unique name, type and description. Primitive types are stored in memory, bigger data on disk. For purpose of reporting, snapshot can be packed into single file that will contain both data and their descriptions. For further reference see python documentation for *insecticide/snapshot.py*

4.2.3 Reporting and result repository

Test results are stored in **MySQL database** to which access is provided through **django** api. Settings are stored in *TestResultStorage/settings.py* and should be synchronized between master and any slave reporting to master.

Small data (textual information, return codes, etc) are stored in database. Big data such as snapshots are stored on disk. Every file has to have entry in database where it's relation to test run is stored. **File transfer** between slave and master must be handled externally. Preferred method how to handle this is to map storage directory (for example /var/lib/TestResultStorage/data) between master and slave by NFS (in case of separate hosts) or by method provided by virtualization software (if master and slave are virtualized on the same machine).

Django stores objects as rows in **tables**, one table per class (plus some index and many-to-many relation tables). We use *TestRun* class for information about one run of single test, *TestRunData* class to hold auxiliary information for test - backtrace, exception, file name of snapshot. Set of TestRuns that were executed together are connected by *BatchRun* object which holds common attributes such as machine that executes tests, branch, revision and profile. Projects and profile information are represented as foreign key to tables *Project* and *ProfileInfo* because they repeats a lot.

In **Nose,** reporting of test results is wrapped in **ReportProxy** class. It holds BatchRun information, generates and commits TestRun objects into database. Nose integration is done by **zfsReportPlugin**. This plugin implements *startTest* hook for test duration measurements, *addFailure*, *addError* and *addSuccess* hooks to catch and report test results. When there is a unhandled system error (python Exception), it is caught by outer try-except block and reported to database too.

Result repository has dynamic **web interface** which consists of tests and batches listing pages (with simple filtering options), detail pages for test run and batch run and project list page. If snapshot is available for test run, it can be downloaded from test run detail page. Older results can be deleted from

administration interface.

4.2.4 Options

Test sets can be **filtered** in three ways:

- by passing **list of files** (modules, classes, tests), that should be run (disables search)
- by **nose.attrib** plugin. User can define expression, that must evaluate to True for given test attributes. For example expression '*not disabled*' will discard tests where *test.disabled* evaluates to True. Test loaded from saved path bypasses this filtering.
- by **name regular expressions**. By default, test name must match regular expression *(?:^|[\\b_\\./-])[Tt]est)* to be executed. But this expression is configurable through *NOSE_TESTMATCH* environment variable.

ZfsConfig plugin provides user-definable configuration files straight to tests. List of configuration files (in defined format) can be passed to plugin. Plugin will read them, convert to python object, and pass this object to all tests.

To handle **deadlocks** and infinite loops both in zfsd and malformed tests, there is ***timed* decorator**, by which timeout for test and handler function to execute when time runs out can be set.Current implementation of handler will send SIGABRT to zfsd causing termination with core dump generated.

4.2.5 Random workload generation

Generation of random workload to file system (stress tests) is done by **zfsStressGenerator** plugin. User must define so called *meta-tests*, basic operations, from which workload will consists. Random workload is generated by graph walk. By default, full graph with even edge scores will be used, but user can define dependency graph by himself.

Format for meta-test is identical with normal test, meta-tests intended to be used together must be listed in one test class (as python has multiple inheritance allowed, this should not be problem).

If there are **dependencies between tests** (such as that open test should run before read test), they can be defined by graph. Graph format is python dictionary, where key is meta-test name and value is list of oriented edges originating in the meta-test. Edge is defined by target meta-test name and by edge score. Score is arbitrary positive number, bigger number means bigger probability to use that edge.

Meta-test chain will terminate, if meta-test with no successor is hit, or there can be defined terminating probability. Hard **length limit** of meta-test chain can be defined by plugin option.

If stress test fails, the path which causes failure is saved into file for further usage. By default, **saved paths** aren't committed into repository, but it can be specified to do so. ZfsStressGenerator plugin can try to **strip failed tests** to see, if shorter test sequence would cause failure too. Actual algorithms to strip sequences is shortest path, disable function, and skip part.

4.2.6 Extendability

Extending tests should be pretty straightforward. As python is object oriented language, inheritance should be used.

If new features are needed on the level of driving component (Nose), they should be delivered as new plugins. Plugins are nearly independent, but the ordering of their execution should be preserved. Execution order is given by *score* attribute ascending. See [40] for further information.

4.3 C test

For unit testing of C source code **Zen-unit** library was developed.

Library has very minimalistic api consisting of single file with four defines:

1. **ZEN_TEST** macro used to declare test header
2. **ZEN_ASSERT** to test conditions in tests
3. **PASS** which is value that should be returned from passing tests.
4. **FAIL** which is value that should be returned from failing tests. Where *FAIL* is recommended return value, but any test returning value different from *PASS* is considered as failed.

Search for tests is done using libelf, tests are looked for in dyntab and symtab of binary and all libraries linked to it. Shared libraries can be tested by linking through *LD_PRELOAD* to *zentest* binary.

There were more options to use for test collection:

- To use user listed tests in some type of #ifdef declared main. This doesn't remove the need of hand written list of tests, and moreover creates some difficulties in main source file.

- To use full C grammar to search for tests in source files and to generate main file. This remove need of listing tests, but requires full C parser.

- To use some C preprocessor to generate simply parseable overview (XML) and generate test list from them. This possibility was not fully explored, but was considered far more complicated that binary format based discovery.

For integration with nose, we use parsing of test output. We have considered generation of test lists or test libraries (through swig), but we found it as redundant overhead.

4.4 Build system

Original ZlomekFS build system (make) was switched to **automake** due to problems with library directories (lib vs lib64) on 64bit systems. Hence for C based components **autoconf**, **automake** and **libtool** are used, for python based components **setuptools** are used (with make wrapper for better compatibility with other tools).

Target audience of both ZlomekFS and regression testing framework uses mainly **Redhat** or **Fedora** based systems. To ease installation and upgrades, it was decided to provide automatic build targets for **RPM** packages. For other systems, .tar.gz source packages can be generated.

4.4.1 Standard targets

All components understand following make targets:

- all - build all binaries and libraries

- doc - build documentation

- dist - build .tar.gz source package

- rpm - build all available rpm packages (source, doc, binary)

- clean - remove generated data

- test - run available tests

4.5 Buildbot configuration

Buildbot is configured to create builder for every component on every host. Thanks to build system unification, build step sequence is equal for all components: it goes update - build - test - make rpm - install - upload.

Change source is SVN polling, schedulers are configured to wait some time after change before builder is run.

Test driving (Nose) was included in way, that doesn't need external configuration, only django needs to have *DJANGO_SETTINGS_MODULE* present in environment, so it is exported in start time of buildbot.

For infinite testing loops it was decided to run them outside of buildbot. Running them inside buildbot would generate long progress bars displacing other build results from view. Since there can be only one ZlomekFS daemon running at the time, synchronization to avoid collision were needed. Thus in every ZlomekFS build, there is step signaling infinite loop controller to pause run (before buildbot cycle) and step signaling to unpause run (after buildbot cycle).

4.6 Typical call sequence

When testing is invoked (through buildbot or manually by *make test*), the cooperation and calls between components are as follows.

First component used is **nose wrapper**, which loads **profile**, creates **BatchRun** object and commits it to TestResultRepository. Then, control is passed to nose (environment and command line options are preserved).

Nose parses environment and command line options and **configure enabled plugins** according to them. Here is initialization phase of *SnapshotPlugin*, *ZfsConfig* plugin, *ZfsStressGenerator*, *ZfsReportPlugin* executed. ZfsConfig plugin tries to load configuration files for tests that contains for example paths for zfsd and zfsd configuration. SnapshotPlugin ensure that required directories for snapshots exists. ZfsReportPlugin calls TestResultRepository fetching previously created object of BatchRun.

After initialization phase, nose will **search for tests**. Standard tests are handled directly by nose, plugins are not involved in this process. When file with meta class is found (or directly passed), *ZfsStressGenerator* will load all tests from it to cache and block their normal execution. The same is done for saved path files. If binary file (or library) is found, *ZenPlugin* will try to execute it as *zen test suite* - that means executing it with LD_PRELOAD of libzenunit.so. Then output is parsed and if there were test run, ZenPlugin will create report for it.

Before execution of tests begins, ZfsStressGenerator will appends ContextSuite containing **stress tests** into *main ContextSuite* of Nose.

Then, execution phase is reached. For every ContextSuite (TestCase) it's **context is initialized**. In initialization, *ZfsConfig* plugin passes **ConfigParser** object (representation of config files) to test. In case of classic (non stress) tests, initialization of zfsd is made in setup and teardown methods in scope of method, in case of stress tests initialization of zfsd is made in setupClass and teardownClass - methods in scope of class fixtures. In setup and teardown methods of stress test, there is only check, if zfsd is still running and if not, execution is raised.

Initialization of zfsd is encapsulated into *ZfsProxy* object, consists of: unpacking configuration, reading configuration and fork of actual zfsd (passing given parameters). After fork, there is wait loop where proxy object tries to connect to zfsd through d-bus, and check if it have started correctly. Eventually, when something goes wrong, exception is raised.

In next phase, **tests are executed**. Each test is considered as passing if there is no exception raised. There are two types of exceptions distinguished. If the exception is of type *AssertionException* (raised by assert clause), test is considered as failed. *Other exceptions* are handled as not-expected, thus error is reported. Tests can modify zfsd behavior by calls on *ZfsProxy* instance, for example there is possibility to change log level or facility set (this can be convenient in tests of special scope - if test is aimed to locking problems, it can constrain log messages to threading related only).

If failure or error is detected, *SnapshotPlugin* will **create snapshot** of failed test where arbitrary data defined by developer are appended in test class *snapshot* method. In current implementation of *ZfsTest* class, it means test object, test data, *ZfsProxy* object, zfsd *log file*, nose log file, zfsd *core dump*, zfs *cache directory*, zfsd stdout and stderr, and in case of comparing tests the directory on compare filesystem. In case of stress test, *call sequence* is stored too.

Reporting of classic tests are handled by *ZfsReportPlugin* which creates *TestRun* object with appropriate parameters and commits it to *TestResultStorage*. If test has failed, ZfsReportPlugin will append failure data (*snapshot, backtrace, exception*) to *TestRunData*. Stress tests are handled and reported by *ZfsStressGenerator* plugin. This special case is separated to prevent multiple reports of the same call sequence.

After **stress test** fails, *ZfsStressGenrator* can try to **prune** the call sequence and put it back to test queue. This is done only given number of times, then the last failure (some pruned sequences may not fail) is stored by ZfsStressGenerator to **saved path** file.

When all tests have run, control is passed to **nose wrapper**, which **final-**

izes BatchRun - sets it's duration and result. If there is exception, that is not handled in nose, it is caught by wrapper and reported as system failure in current BatchRun.

Chapter 5

Conclusion

5.1 Work done

Goal of this thesis was to extend existing ZlomekFS implementation and to implement regression testing framework which would fit it's special needs. After exploration of ZlomekFS, there were clear need of logging facility and remote state discovery to allow reliable testing. From research done on related projects, it was decided to create new logger and made status information available through d-bus.

As basic regression testing framework, existing solution (Nose) was used. It was extended by plugins to support new types of tests and to provide required features.

Under it, prototypes of tests are created. First type of tests identifies problems of filesystem by performing operations on second, reliable, filesystem and by comparing results. Second test type identifies errors by checking, if behavior of filesystem is as expected (data read is the same as written, ...). For distributed testing, basic objects are provided as wrappers for python twisted perspective broker objects.

By plugin, possibility to generate random workload to filesystem is provided. This plugin can be constrained to generate valid sequences of operations only. Plugin can prune sequence to find minimal sequence needed to reproduce the error.

Snapshoting plugin offer chance to have as much state information and trace protocol as possible to ease debugging of the problem which have caused the problem. Current implementation provides core dump of daemon, log files, sequence which have caused the error, cache content, comparison file system (if used) snapshot, and python component state. Snapshot plugin can include most of data types that can be required when debugging, thus tester can easily define other data to include to snapshots.

Currently, change of network conditions is possible only by inserting special rules into iptables. The change of network conditions is mainly aimed to check behavior in disconnected or slowly connected state. For future, there is d-bus interface prepared to allow direct change of ZlomekFS state without changing network conditions (which could be useful in real usage too). When there will be need to provide network protocol robustness testing, it will need use of sophisticated external tool which is beyond scope of this thesis.

It is possible to use this system for testing of any other filesystem. Especially in case of userspace based filesystem the modification needed will be only change of daemon binary and settings (which will be obviously different). If filesystem would have kernel component (especially in case of full kernel based filesystems) there will be need of *kdump* integration for snapshoting state of filesystem in case of failure.

For unit testing of small parts of code (whitebox testing), small library for C were written. This library called Zen-unit has very intuitive interface, but what is special about it is, that it has automatic test discovery. This allows thing common in scripting languages, but rare in compiled ones: to write tests anywhere in code without listing them in some central block.

Developer documentation for C code was written in DoxyGen and there are build target for generating HTML documentation available (moreover RPM packages with documentation can be build too). For python we use language provided __doc__ attributes with predefined syntax.

5.2 further work

- threading of zfs (single thread?)
  - mode control
  - full system reproduction

Appendix A

Coding conventions

C based code

For code in C, original formatting from ZlomekFS was adopted.

Identifiers ah are in lower case, words separated by underscore.

```
uint32_t log_level;
```

Defines (macros) are in upper case, words separated by underscore.

```
#define MY_MACRO_CONSTANT 5
```

Typedefs are in lower case with suffix *_t*.

```
typedef uint32_t fibheapkey_t;
```

Braces around code block should be on new lines, indentation level as previous code.

```
syp_error set_log_level (logger target, log_level_t level)
{
  target->log_level = level;
  return NOERR;
}
```

Braces around function arguments should be separated from function name by one space, if argument list is multiline, ending brace should be right after last argument (on same line).

```
syp_error send_uint32_by_function (uint32_t data,
  syp_error (*function)
  (int, uint32_t, const struct sockaddr *, socklen_t),
  const char * ip, uint16_t port);
```

I**ndentation** should be two spaces per level.

```
syp_error dbus_disconnect(DBusConnection ** connection)
{
  if (connection == NULL)
    return ERR_BAD_PARAMS;
  if (*connection == NULL)
    return ERR_NOT_INITIALIZED;
  dbus_bus_release_name (*connection,
    SYPLOG_DEFAULT_DBUS_SOURCE, NULL);
  dbus_connection_unref(*connection);
  *connection = NULL;
  return NOERR;
}
```

Operators should be separated from arguments by one space on both sides.

```
file_position += bytes_written;
```

Comments have one space between comment mark and comment text. They are on line before code they are describing.

```
/*! Structure holding logger state and configuration. */
typedef struct logger_def
{
  /// input - output medium definition struct
  struct medium_def printer;
```

File names consisting from more words should have dash between words.

```
control-protocol.h
```

Python code

For code in python, formatting from Nose [40] was adopted.

Identifiers ah are in CamelCase, class names with first letter in upper case, instance names with first letter in lower case.

I**ndentation** should be four spaces per level.

```
class DependencyGraph(object):
    graph = None
    currentNode = None
    randomGenerator = SystemRandom()

    def equals(self, graph):
        return self.graph == graph.graph and \
            self.currentNode == graph.currentNode

    def initRandomStartNode(self):
        self.restart(self.randomGenerator.choice(self.graph.keys()))
```

Braces around function arguments should be right after function name.

Arguments should be separated by one space.

```
def testLocal(self, empty):
    assert self.buildGraphsAndCompare(reference = self.nonUniformGraph,
        buildMethod = GraphBuilder.USE_LOCAL,
        methods = self.nonUniformMethods)
```

Operators should be separated from arguments by one space on both sides.

```
file_position += bytes_written;
```

Documentation comments should have one space between comment mark and comment text. They should be on line after element they are describing.

```
def isMetaClass (self, cls):
    """ Tests if class is meta class (should contain meta
tests)

        :Parameters:
            cls:  class object to check

        :Return:
            True if is metaclass, False otherwise
    """
```

Code comments should have one space between comment mark and comment text. They should be on line before code they are describing.

```
def countNiceElements(list):
    """ Count elements which are nice :)  """
    count = 0
    for element in list:
        if isNice(element):
            count += 1

    return count
```

Appendix B

Installation

This guide is based on new Fedora 8 installation, installation to other systems may be different. In example, we have buildmaster on IP 192.168.16.253, buildslave on IP 192.168.16.252 and development system on IP 192.168.16.241. We assume that buildslave has basic development packages (such as gcc) installed. Not that routing description is not included. Routing should be set if domain names are used (in build system configuration or remote testing).

Development system may not be part of buildbot network, but the experience is much better if it is, because then package versions on it will be the same as on buildslave.

Buildmaster

Create user account under which buildbot will run on buildmaster.

```
useradd -d /home/buildmaster -s /bin/bash buildmaster
```

Then, install external packages and tools.

```
yum install buildbot python-sqlite2 mysql-server \
 MySQL-python screen
```

Install TestResultStorage. Django in version version 0.97 (pre) is required.

```
rpm -ivh python-django-snapshot-*.rpm TestResultStorage-*.rpm
```

Set mysql to start on boot and start it.

```
/sbin/chkconfig mysqld on
/etc/init.d/mysqld start
```

Change root password for mysql.

```
/usr/bin/mysqladmin -u root password 'secret'
/usr/bin/mysqladmin -u root -h 192.168.16.253 password 'secret'
```

Create database for TestResultStorage.

```
echo '
CREATE DATABASE trs character set utf8;
GRANT all ON trs.* TO nose@localhost IDENTIFIED BY 'secret';
GRANT all ON trs.* TO nose@192.168.16.252 IDENTIFIED BY 'secret';
GRANT all ON trs.* TO nose@192.168.16.241 IDENTIFIED BY 'secret';
FLUSH PRIVILEGES;
'| mysql -user=root -password=secret
```

Setup TestResultStorage to use local mysql database with right credentials (Figure B).

```
...
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'trs'
DATABASE_USER = 'nose'
DATABASE_PASSWORD = 'secret'
DATABASE_HOST = 'localhost'
DATABASE_PORT = ''
...
```

Figure B.1: TestResultStorage/settings.py (buildmaster)

Create TestResultStorage tables.

```
cd /usr/lib/python2.5/site-packages/TestResultStorage
python manage.py syncdb
```

Checkout (export) buildmaster's configuration

```
su buildmaster
cd /home/buildmaster
svn export \
 http://shiva.ms.mff.cuni.cz/svn/zzzzzfs/branches/zouhar/buildbot/buildmaste
 zlomekfs
```

Setup buildmaster to allow client connections and to start www server on right port (Figure B, Figure B). Note that if svn url or branching schema changes, they should be tweaked too.

```
WAIT_BEFORE_BUILD = 1

c['slaves'] = [BuildSlave("misc", "secret"),
    BuildSlave("zen", "secret")]
c['slavePortnum'] = "tcp:9989"

c['projectName'] = 'ZlomekFS'
c['projectURL'] = 'http://dsrg.mff.cuni.cz/~ceres/prj/zlomekFS'

c['buildbotURL'] = 'http://192.168.16.253:8010'

svnurl = 'https://shiva.ms.mff.cuni.cz/svn/zzzzzfs'
...
```

Figure B.2: master.cfg

```
...
basedir = r'/home/buildmaster/zlomekfs'
configfile = r'master.cfg'
...
```

Figure B.3: buildbot.tac (master)

Set buildbot to start on boot, for example by adding crontab entry. (first line in Figure B on the following page).

```
...
@reboot make start -C /home/buildmaster/zlomekfs
@reboot screen -d -m -S \
 TestResultStorage python \
 /usr/lib/python2.5/site-packages/TestResultStorage/manage.py \
 runserver 192.168.16.253:8020
0 2 * * * /home/buildmaster/cleanup.py
```

Figure B.4: buildmaster.cron

Setup automatic cleanup of old data. It can be done by cleanup.sh located in *misc* directory in repository.

```
svn cat \
 https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/branches/zouhar/misc/cleanup.py \
 > /home/buildmaster/cleanup.py
chmod +x /home/buildmaster/cleanup.py
```

Set cron to execute this script every day on 2 a.m. (second line in Figure B)
Start buildmaster.

```
make start -C /home/buildmaster/zlomekfs
```

Open ports 3306, 8010, 8020, 9989 (or other, if setting in master.cfg is different) on firewall. Rules below are only examples, they should be pernament (for example written in /etc/sysconfig/iptables).

```
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 8010 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 8020 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 9989 -source 192.168.16.0/24 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 3306 -source 192.168.16.0/24 -j ACCEPT
```

This is all except for file transfers. If you want to use nfs for file transfers, read nfs configuration below.

Set data directory to be exported (B).

```
/var/lib/TestResultStorage/data \
 192.168.16.252(fsid=0,rw,root_squash,sync) \
 192.168.16.241(fsid=0,rw,root_squash,sync)
```

Figure B.5: /etc/exports

Tell portmap to allow connections to services (B).

```
portmap:  192.168.16.241 , 192.168.16.252
lockd:  192.168.16.241 , 192.168.16.252
rquotad:  192.168.16.241 , 192.168.16.252
mountd:  192.168.16.241 , 192.168.16.252
statd:  192.168.16.241, 192.168.16.252
```

Figure B.6: /etc/hosts.allow

Set mount daemon to use specific port - needed for firewall settings (B).

```
...
MOUNTD_PORT=32773
...
```

Figure B.7: /etc/sysconfig/nfs

Open ports on firewall. Note that you must make this rules pernament for example through *system-config-firewall*.

```
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 2049 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m udp -p udp -dport 2049 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
```

```
 -m tcp -p tcp -dport 111 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m udp -p udp -dport 111 -j ACCEPT
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m udp -p udp -dport 32773 -j ACCEPT
```

Run nfs and make it start upon boot.

```
/sbin/service nfs start
/sbin/chkconfig nfs on
```

Buildslave

First, install required packages. Note that not all are available from Fedora repositories. For i386 and x86_64 architecture they can be found on thesis cd.

```
yum install buildbot python-sqlite2 MySQL-python kernel-devel \
 dbus dbus-devel libtool autoconf automake gettext gettext-devel \
 python-setuptools python-nose pyflakes screen
rpm -ivh python-django-snapshot-*.rpm \
 libelf0-0.8.10-*.rpm libelf0-devel-0.8.10-*.rpm \
 TestResultStorage-*.rpm py25_pysvn_svn144-*.rpm
```

Install packages from all components. This can be skipped, but when further builds will go in wrong order, dependency problems could arrive.

```
rpm -ivh zen-unit-*.rpm syplog-*.rpm pysyplog-*.rpm \
 zlomekfs-*.rpm zfsd-status-*.rpm TestResultStorage-*.rpm \
 insecticide-*.rpm
```

Note that installation of packages may require removal of previously installed ones (for example fuse).

Restart D-bus to use new configuration (allow syplog and zfsd communication).

```
/etc/init.d/messagebus restart
```

Change TestResultStorage settings to store results on buildmaster (B).

```
...
DATABASE_ENGINE = 'mysql'
DATABASE_NAME = 'trs'
DATABASE_USER = 'nose'
DATABASE_PASSWORD = 'secret'
DATABASE_HOST = '192.168.16.253'
DATABASE_PORT = ''3306'
...
```

Figure B.8: TestResultStorage/settings.py (buildslave)

If you use nfs for file transfers, set nfs mount (B).

```
...
192.168.16.253:/var/lib/TestResultStorage/data \
 /var/lib/TestResultStorage/data nfs defaults 0 0
...
```

Figure B.9: /etc/fstab

Create directory for builds and fetch config.

```
mkdir -p /var/buildbot
cd /var/buildbot
svn export \
 https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/branches/zouhar/buildbot/buildslav
 zlomekfs
```

Change buildbot configuration to connect to master and use actual credentials (B).

```
...
basedir = r'/var/buildbot/zlomekfs'
buildmaster_host = '192.168.16.253'
port = 9989
slavename = 'zen'
passwd = 'secret'
...
```

Figure B.10: buildbot.tac (slave)

If infinite testing loop should run on host, checkout it's testing configuration.

```
cd /var/buildbot
svn checkout \
 https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/branches/zouhar/zlomekfs/tests/nos
 zfsTests
```

Open zfsd port on firewall to allow comunication between nodes.

```
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 12323 -j ACCEPT
```

On remote zfs provider, remote listening port on firewall.

```
/sbin/iptables -A RH-Firewall-1-INPUT -m state -state NEW \
 -m tcp -p tcp -dport 8007 -j ACCEPT
```

Configure buildbot to start on boot, and if host should do infinite testing loop, configure start on boot too (for example via crontab B).

```
...
@reboot buildbot start /var/buildbot/zlomekfs
@reboot screen -d -m cd /var/buildbot/zfsTests \
 && ./infiniteControl.sh run
```

Figure B.11: buildslave.cron

Start buildbot and infinite testing loop on.

```
buildbot start /var/buildbot/zlomekfs
cd /var/buildbot/zfsTests
screen -d -m ./infiniteControl.sh run
```

On Slaves acting in remote testing as slave ZlomkeFS providers, lines

```
screen -d -m ./infiniteControl.sh run
```

should be replaced with

```
screen -d -m ./remoteZfs.py
```

(we want to run control on one slave and remote zfs on others).

Development system

When installing on development system without need of automatic builds, just install required packages and build projects in correct order.

For testing all components except ZlomekFS can be tested without install (make test). The reason why this is not possible for ZlomekFS is need of fuse build, which is integrated and needs to create device links, install kernel modules etc.

If you want environment as close to buildslave as possible, you can install your system in the same way as is described in Subsection B. But even in this case development should be done without buildbot checkouts. Changing buildbot checkouts could lead into conflicts upon automatic builds.

To not spoil central TestResultRepository with your builds you enable out your own TestResultRepository on local machine (create mysql database) and report into it. This should be done in case of slow connection to master too. Running automatic tests without ZfsReportPlugin is discouraged - there would be little backtrace provided in that case.

Appendix C

Enclosed CD

Bibliography

[1] Berkeley lab checkpoint/restart (blcr). http://ftg.lbl.gov/CheckpointRestart/CheckpointRestart.shtml.

[2] Chainsaw. http://logging.apache.org/log4j/docs/chainsaw.html.

[3] Continuum. http://maven.apache.org/continuum/.

[4] Django. http://www.djangoproject.com.

[5] Frysk. http://sourceware.org/frysk/.

[6] gcore. http://sourceware.org/gdb/.

[7] Gump. http://gump.apache.org/index.html.

[8] Linux test project. http://ltp.sourceforge.net/.

[9] Opensolaris zfs test suite. http://opensolaris.org/os/community/zfs/zfstestsuite/.

[10] Openvz. http://openvz.org.

[11] Pydoc. http://docs.python.org/lib/module-pydoc.html.

[12] The py.test tool and library. http://codespeak.net/py/dist/test.html.

[13] Python. http://www.python.org.

[14] Qemu. http://fabrice.bellard.free.fr/qemu/.

[15] restructuredtext markup specification. http://docutils.sourceforge.net/docs/ref/rst/restructu

[16] Tinderbox. http://www.mozilla.org/tinderbox.html.

[17] Unit testing frameworks list. http://www.testingfaqs.org/t-unit.html.

[18] Vmware server. http://www.vmware.com/products/server/.

[19] Xen. http://www.xensource.com/.

[20] Toshikazu Ando. Cunit for mr.ando. http://park.ruru.ne.jp/ando/work/CUnitForAndo/html/.

[21] Jr. Avadis Tevanian. Fsx. http://www.freebsd.org/cgi/cvsweb.cgi/src/tools/regression/fsx/.

[22] Stefano Barbato. C++ unit testing easy environment. http://codesink.org/cutee_unit_testing.html.

[23] Kent Beck. Simple smalltalk testing:with patterns. http://www.xprogramming.com/testfram.htm.

[24] Bernard Blackham. Cryopid - a process freezer for linux. http://cryopid.berlios.de/.

[25] John Brewer. Minunit - a minimal unit testing framework for c. http://www.jera.com/techinfo/jtns/jtn002.html.

[26] Guido van Rossum David Goodger. Pep-0257, docstring conventions. http://www.python.org/dev/peps/pep-0257/.

[27] Martin Flower. Continuous integration. *http://www.martinfowler.com*, 2006. http://www.martinfowler.com/articles/continuousIntegration.html.

[28] Jerrico L. Gamis. Robust c unit. http://rcunit.sourceforge.net.

[29] Philippe Neumann Gina Haussge. Doxypy. http://code.foosel.org/doxypy.

[30] David Goodger. Pep-0256, docstring processing system framework. http://www.python.org/dev/peps/pep-0256/.

[31] Peter Hagg. Gunit. https://garage.maemo.org/projects/gunit.

[32] Philipp von Weiterhausen Holger Krekel, Jens-Uwe Mager. py.lib library. http://codespeak.net/py/dist/index.html.

[33] Asim Jalis. Cutest: C unit testing framework. http://cutest.sourceforge.net/.

[34] JevonWright. Simple c++ testing framework. http://simplectest.sourceforge.net/.

[35] Erich Gamma Kent Beck and comunity. Junit - java unit testing framework. http://www.junit.org.

[36] Anil Kumar. Cunit - a unit testing framework for c. http://cunit.sourceforge.net/index.html.

[37] Niklas Lundell. Cpp test. http://cpptest.sourceforge.net.

[38] Arien Malec. Check: A unit testing framework for c. http://check.sourceforge.net/.

[39] Alden Almagro Paul Julius and col. Cruise control. http://cruisecontrol.sourceforge.net/index.html.

[40] Jason Pellerin and col. Nose - alternative unit testing for python. http://python-nose.googlecode.com.

[41] Steve Purcell. Pyunit - the standard unit testing framework for python. http://pyunit.sourceforge.net/.

[42] Steve Purcell. Unittest api. http://docs.python.org/lib/doctest-unittest-api.html.

[43] Hans Reiser. Mongo. http://namesys.com/benchmarks/mongo_readme.html.

[44] Olexander Sudakov. Chpox: transparent checkpointing and restarting of processes on linux clusters. http://freshmeat.net/projects/chpox/.

[45] SUCKS team. Somewhat usefull camcorder system, 2007. http://urtax.ms.mff.cuni.cz/prk/.

[46] Dimitri van Heesch. Doxygen. http://www.stack.nl/ dimitri/doxygen/.

[47] Erez Volk. Cxxtest. http://cxxtest.sourceforge.net/.

[48] Brian Warner. The buildbot. http://buildbot.sourceforge.net/.