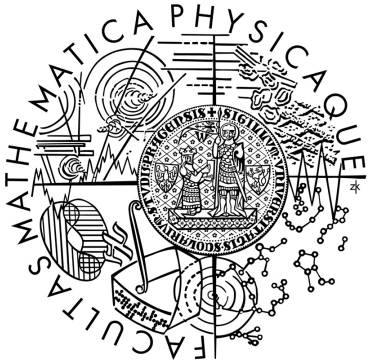
Charles University in Prague
Faculty of Mathematics and Physics

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 diplomové práce: Doc. Ing. Petr Tůma, Dr.
e-mail vedoucího: petr.tuma@mff.cuni.cz

Abstrakt: ZlomekFS je distribuovaný systém souborů určený k transparentnímu sdílení adresářových stromů. Tato práce popisuje tvorbu systému regresního testování pro ZlomekFS.
Práce shrnuje metody používané k testování softwaru podobného ZlomekFS a na jejich základě vytváří samostatný systém regresního testování. Systém sestává z šesti částí: (1) knihovny pro unit testing kódu psaného v jazyce C, která poskytuje automatické vyhledávání testů s minimalistickým rozhraním, (2) logovacího frameworku s rozhraním pro jazyky C a Python, který podporuje filtrování a vzdálené ovládání, (3) generátoru náhodné zátěže pro vytváření náhodných testovacích sekvencí pomocí grafu závislosti s podporou reprodukování a zmenšování zátěže vedoucí k chybě, (4) systému pro kontrolu testování a hlášení výsledků testů, (5) repozitáře pro ukládání výsledků s webovým uživatelským rozhraním, (6) serveru pro automatické sestavování a testování.

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 filesystem for transparent sharing of directory trees. This thesis describes how regression testing for ZlomekFS was built.
The thesis summarizes software testing methods in the context of ZlomekFS and proceeds to build an actual testing system for ZlomekFS, consisting of six parts: (1) a unit testing framework for C code with an automatic test discovery and a minimalistic interface, (2) a logging facility with C and Python interfaces, with filtering and remote control, (3) workload generator for random test sequences generated from an operation dependency graph with support for failure reproduction and trace reduction, (4) test controlling and reporting framework, (5) test result repository with web user interface, (6) continuous integration server for automatic builds and tests.

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 (a 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 this, resulting export hierarchy must be a tree. When two nodes connect, for unique volume one of them must be a master providing content and the other a client. This relationship can be bidirectional (a client for one volume can be the master for another).

A node can cache content of a volume obtained from another node and provide it to further nodes. This caching is not required, a node can use remote volume without any local cache too. On one node, all volumes must be mounted in a tree with one root directory (volume) and other volumes mounted under the root.

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

For mobile nodes, there are two modes of operation (besides full speed connection). If there is no connection, the node operates in disconnected mode (data is served from cache only). Eventually, when the node connects again, data is 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 the master node. When changes are made to files in the disconnected mode, conflicts can arise upon synchronization. In ZlomekFS, a conflict is represented as set of files in a special directory.

Filesystems, same as any other software, can contain bugs. But in case of filesystems, there is a bigger-than-usual need for reliability. This is due to the low-level character of filesystems: applications depend on bug free work of components underneath them, and most applications use filesystems to store

permanent data. To ensure reliability of a filesystem, exhaustive testing should be done to cover as many use-cases of the software as possible. This is not only convenient for one term development, but also essential when further extension and development will be done.

However, ZlomekFS lacks any tests, or testing environment. As described, ZlomekFS is very complex and special filesystem, and there are only few similar filesystems. Because of this, there is need to develop new framework or at least special tools to allow testing of its special functions. 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 submit 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.

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 testing 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 outlined, 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

In general, testing of a filesystem is very similar to testing of any other software. So basic requirements holds, and in addition there are some extra requirements.

To provide best stability of software, tests should cover maximum of its functionality. By executing these tests, bugs can be found before they can affect real usage. The bigger the coverage is, the smaller is the probability that bug would stay undiscovered. For failed tests, testing tools should provide as much information from the time of failure as possible. These data can be later used to reproduce the failure or at least could be useful for tracing.

In reality, the coverage is often the biggest problem - programmers tend to omit writing tests, or refuse to write them at all. The cause of this is that writing tests is uninteresting extra work, sometimes consisting of non-trivial steps.

To reduce this problem, the test format (Section 2.2) has to be simple, readable, and user-friendly. To make it even easier to write tests, there is tendency to allow small unit tests as near to actual code as possible (Section 2.3). If these tests are well written, they can serve as programmer documentation too.

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 (Section 2.4), that may or may not support resuming. When automated test run fails, tester may need to run it again. These reruns may use snapshots, if the method used for creating them makes the resume possible.

For tracing the code execution, there should be some tracing and logging tools (Section 2.5). They have to have minimal footprint, but collect as much information as possible.

During regression testing, tests should be executed automatically in scheduled periods (Section 2.6). Results from these runs should be collected and presented centrally (Subsection 2.7).

When testing unsafe operation, test execution should be separated from

running system. This approach is often called sandboxing (Section 2.9).

For filesystem testing, it is hard to find good testing patterns (Section 2.10) which will cover all cases that can occur. So it is a good idea to have some random workload generator (Section 2.11) that would exercise the filesystem randomly.

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 (Section 2.8).

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

2.1 Test types

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

2.1.1 Specification testing

We can look on a filesystem as on a specification of a way how to store data and associated metadata 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 a correct state to another 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. A well-known group of such filesystems are network filesystems. They assume that some other filesystem does the storage, they specify only the way how data will be accessible remotely, and put some restrictions on the filesystem behavior. In this case, the particular implementation is tested if it conforms to the specification. For example, in case of NFS there are test suites for checking interface stability and protocol conformity [9].

2.1.3 Functional testing

Functional testing aims to test actual code if it works as expected and doesn't contain bugs.

Functional tests can be divided by scope to:

- **Unit tests** are the smallest tests. They don't test functionality of the whole application, but test functionality of small pieces of code, ensuring that the code works as expected. For example, when there is function for writing to file, there should be a unit test that will try to write data through this function and then will check if the data were written correctly. Main purpose of unit tests is to prevent regressions upon code refractoring. They test the internal api stability. For best effect, they should cover 100% of code (all code in an application should be executed upon run of all unit tests). Unit testing should be white box testing - tester should know how the code is working. For unit testing, there is no difference between testing classic application and filesystem testing.

- **Feature tests** are used to check global functionality provided by the software to its user. This type of tests is very similar to api conformity testing (Subsection 2.1.2), but in general, they don't have to use user api. They should cover all use-cases of the software, but not necesarily all code (yet the untested code may be redundant). Tests of this level should ensure that the software will work as it is intended to. When tests are accessing software through general user api without knowledge of internals (or regardless of the knowledge), it is called black box testing. For filesystems, black box testing is often used. One reason for it is that since user api of all filesystems is the same (at least basic part), then these tests can be used for more filesystems. Second reason is, that filesystems are very complex and it is easier to write tests for general problems (such as race conditions) than searching for special paths in code that could fail. Actually, these tests for special paths should be written post-mortem when some bugs are found (to prevent them from appearing again).

2.1.4 Benchmarking

Benchmarking gives an answer to the question "how long it will take" for every use-case of a filesystem. Measurements are done on different filesystem implementations, or filesystems with similar purpose. The setup should be similar for all subjects tested. Most of benchmarking tools assumes that the

implementation is sane and doesn't check results of operations. 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 and the main requirement is ease of use. 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 applies on both language of tests and interface for tests themselves.

The basic choice is to write tests in **native language** of the application. Sometimes, embedding tests within application code is supported with some flags saying, "this is a 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 **proprietary format**. This offers possibility of defining a format that is very suitable for the needs of particular software, but brings disadvantage of having to change the format every time new requirement is found. Proprietary format is also often less compatible with external tools.

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 the other hand, XML is a very unsuitable format for hand-written code and the DTD of configuration is often hard to understand.

2.3 Unit based testing frameworks

Small tests placed near to code tested are often called unit tests. They should be simple and fast. Sometimes, they are used as programmer documentation too. Unit testing is based on Kent Beck's testing pattern [24].

2.3.1 Common principles

Every **test case** is executed separately, test cases 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 it 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 expected 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 their root test suite. Returned value is an aggregated Result object.

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

2.3.2 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 those are run before and after every single test.

Moreover there can be **additional fixture levels** for other code units (depending on programming 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 once before running all tests in the class, and teardown_class is run after all tests have run. Note that this can break the independency requirement. If fixtures are written for more levels, they run in cascade.

2.3.3 Best practices

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

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

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

2.3.4 Implementations

There is at least one unit testing framework for every programming language (see [18] for short list).

This section 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.

Python has **Unittest [43]** as its standard tool (was called *PyUnit* [42] before integration to python standard distribution). The interface is strongly object oriented, tests 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 [13] is an alternative python unit testing framework. It is part of *py.lib* library [34], there is fixture support on all levels, it doesn't need to inherit from any superclass, but fixed naming convention is used instead. Uses the 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. The py.test most interesting feature is ease of use. It is possible to just write a function with *test* in its name and py.test will collect it, run, and if there is failure or error, the output and backtrace will be printed in a 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 a grant project, and after grant expiration there were little of improvement in the project.

Another unit testing framework for python is **Nose [41]**. Offers backward compatibility with standard *unittest* and some compatibility with *py.test* library. It tries 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 a superclass, it has automated test discovery tool, and uses generators. In addition, Nose is very configurable. It has build in 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** [26], which is C based, and consists only from three lines of code (two macros and one definition). It doesn't offer much, just assert - print message block.

CUnit for Dr.Ando [21] 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 offers just test counting beyond *MinUnit.*

Simple C++ Testing framework [36] is written whole as macros, with somehow weird syntax. It 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. It runs equally in pure C and C++ environment.

CxxTest [49] 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). CxxTest may have problems upon linking with pure C based code. It has support for mocking global functions, but this support works on base of calling functions in a separate namespace, so it is not pure mock and code has to be modified to use mocked functions.

CppTest [38] is another C++ based unit test framework with basic features such as assertions, fixtures, and test suites. Beyond this, CppTest is capable of handle and format output, it 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 in CppTest.

CUnit [37] is C based (still C++ compatible), it supports assertions, suites, test counting, results are stored in global registry, more user interfaces (for running tests) are implemented.

GUnit [33] is another unit testing framework with assertions, suites and fixtures. It uses GTK+ libraries (for almost anything). It has gnome and hildon (embedded) GUI, dedicated logging facility. GUnit doesn't offer discovery, however compiles suites as dynamically loadable libraries.

RCUnit [29] (C based) supports assertions, suites (called modules), fixtures. It has own logging facility, tests can be disabled. RCUnit has documented interface for writing output handlers. HTML and plain text handlers are implemented.

Cutee [23] tends to be as simple as possible. Thus it supports only assertions, no fixtures or suites. Tests are collected automatically, yet files with tests must be listed in Makefile.

Check [39] provides assertions, suites, and simple fixtures. It 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 [35] from basic features provides assertions and suites. It has scripted tool for executable generation.

2.4 Checkpointing

Some failures are hard to reproduce thus developers want as much information about the faulty run as possible. Then, state of the application in time of failure may be required. In this case, support for creating snapshots (checkpoints) can be useful.

Moreover, as the sequence to failure can be very long, it can be convenient to repeat just last part of it, or skip some parts to find the shortest run to bring about the bug. For these reruns, it is best when the consequent runs have the same start conditions as the first one. If resuming from stored state is possible, then a snapshot should be used as this common start point.

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 [25], Chpox [45]) 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 suspend / 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 are many virtualization tools with support for snapshoting (for example openVZ [11], Vmware [19], Xen [20], Qemu [15]). However, working with virtualization is fairly complicated and we can't test hardware dependent issues in virtualized environment. 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.

2.5 Logging, tracing

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 developers). In most cases, these features can be provided by one tool.

2.5.1 Models

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 an useful information, and if we want to avoid this, it force us to change 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 can be flat or of arbitrary structure. This allows 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 (a tree). This eases the 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 programming**. In this case, the logging is not present in the code itself, but it is separated as an independent concern to aspect - logical definition what and where has to be logged.

2.5.2 Pitfalls

Even if an adequate logging tool is used to debug the software, problems can arise when the tool is used in 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 a 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 an ordinary application, some measurements were done.

For testing was used a real application - session server from the SUCKS [46] project. This application is threaded and accessible by network. Logging facility was simple centralized logger with predefined log levels. Logger was altered so it has measured 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 facing page. The first column is logging target (medium to which logs are written), the second column is actual log level. The 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 are included. The 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

Log target	Log level	MeanTimePerChar (µs)		MeanTimePercentage (%)	
		ppc	intel	ppc	intel
Discard	0-7	0.00	0.00	0.005	0.007
Memory	0-2	0.00	0.00	0.005	0.007
Memory	3	0.94	0.26	0.016	0.010
Memory	4	0.55	0.12	0.024	0.013
Memory	5	0.43	0.09	0.030	0.014
Memory	6	0.43	0.09	0.030	0.014
Memory	7	0.43	0.09	0.030	0.014
SHM	0-2	0.00	0.00	0.005	0.007
SHM	3	1.10	0.26	0.019	0.011
SHM	4	0.64	0.12	0.028	0.013
SHM	5	0.48	0.09	0.033	0.015
SHM	6	0.54	0.09	0.036	0.015
SHM	7	0.49	0.09	0.033	0.015
File	0-2	0.00	0.00	0.005	0.007
File	3	1.02	0.27	0.018	0.011
File	4	0.60	0.13	0.026	0.014
File	5	0.46	0.11	0.032	0.018
File	6	0.46	0.11	0.032	0.017
File	7	0.46	0.11	0.032	0.017
Console	0-2	0.00	0.00	0.005	0.007
Console	3	8.01	5.42	0.137	0.217
Console	4	6.35	5.79	0.278	0.611
Console	5	6.35	6.04	0.420	0.977
Console	6	6.26	6.06	0.415	0.980
Console	7	6.45	5.95	0.426	0.963

Table 2.1: Mean logging load to application

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, circular buffer (which can be flushed to file only when error occurred) could be used. Requirement is that filtering must be possible (Filtering can be done afterward by user). Another finding is that on non-blocking media big part of logging footprint is 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 a bug could be ireproducible with different logging level.

Other problem, which can arise with logging, is that logging can act as synchronization primitive preventing some race conditions 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 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 (a good overview of this method is in [28]). To achieve regular testing, it is convenient to use some tool (server) to automate the process. There are many proprietary solutions and even more opensource solutions. It is interesting how many organizations deploy their own system (for example Mozilla Foundation uses Tinderbox [16], Redhat has Frysk [5], ThoughWorks has [40]. Apache foundation has even two projects - Continuum [47] and Gump [7]).

2.6.1 Automation

The 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 general, tests are run independently on human interaction.

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 execute tests, or launch separate process to execute them on background. This approach can ensure that no wrong code gets into the repository - a commit that didn'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 passing vital tests and more in-deep checking tests are executed afterward in stand-alone process. Implementations using post commit hooks are often bound to one version control system.

Other option is to not check validity in commit hooks, but use a **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 changes don't break something

2.6.2 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 leave errors undiscovered. In this case distributed testing is the only way to cover specifics of all platforms. Complementary approach (run one separate machine with full build and check service for every platform) is possible too, but managing such system is huge overhead, and collecting results is also non trivial.

Some tests or projects could be **dangerous** to system, or the project itself could be **part of base system**. These should run in separation of production system (to not break it), in some sort of sandbox (a part of system with restricted rights). 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 a 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 was in other network. In general, there are less security barriers between the host machine and the guest machine, but it doesn'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).**commiter**

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

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. In general, data 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 allows execution of code on remote system. There are many remote procedure call tools such as RPC, CORBA, dcom, python twisted, py.execnet, etc.

Older systems tend 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, **proprietary methods** for communication are used. In gen-

eral, they mimic remote procedure call functionality by sending data within special protocol.

2.6.3 Build system

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.

2.6.4 Presentation layer

Pure commit driven environment when commits are delayed after all tests passed doesn't need presentation layer at all, there is just a **message sent to committer**. But as this variant is rare, nearly all automation systems have presentation layer. This layer mostly serves as notifier about the server's status - it shows if the server is actually running anything, and if so, what is being executed. Short overview of past runs is often provided too. 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 outputs than test count and test results, it tends to be plain text.

2.7 Result repository

Usual test run produces outputs of many types, beginning with standard outputs, going on with operation sequence, debug logs, even including state snapshots, core dumps, etc. These is data of very different types. Unless it is written specially to fit the particular application, the continuous integration server is often incapable of storage and presentation of all data from tests. Some continuous integration servers don't have persistent result storage at all. Because of this, separate result repository is very often implemented.

2.7.1 Data storage

When tests produce only small amount of textual data, **log file** can be used. As the simplest variant, this is often part of continuous integration server.

For complex textual data, **databases** are used. They provide way to store both data and relations between them together. The interface of databases is versatile and easy to use, thus they can be used without modifications.

Big binary data such as snapshots are often stored outside databases. Even though modern databases are capable of storage of them, there is no advantage since no searching or indexing is available for arbitrary binary data. Because of this, special **binary data** (or big data) is stored outside database **in file as it is**. Nevertheless, properties of this data (if any) are still stored to database to ease searching.

2.7.2 Presentation layer

The easiest way to present test results is to leave them as **raw data** (as they are produced by application and test suite). When this approach is used, raw data is often made available for downloading through simple protocol such as FTP or remote console. Raw data holds always full information, doesn't suffer by any losses from transformations. On the other hand, raw data is 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.

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 has a 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, binary data should be downloadable for reading through external tools. Interaction with the web is little bit slower than with local application and user comfort is worst too.

Even raw data must be interpreted by an application to be presented in readable form. When there aren't appropriate general tools, they should be written as a 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 consists of the same amount of work as the 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 a special application is that as written specially for the suite it should fit very well the needs.

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, outputs from long and random tests can be huge (and most of them 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 from testing can be divided to log messages, run backtrace, 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 Section 2.5: when logging doesn't slow the application down and doesn't change behavior, the best approach is to log all, store all (or latest), and to provide a 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 the failure.

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

- **On-time pruning** - the test suite changes log level of the 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 a cyclic buffer. When a failure occurs, the test suite will trim the buffer to store just useful information.
- **Re-run** - tests are executed with logging on minimal level. When a failure occur, test suite will rerun the test with more logging *around* the failure, possibly skipping some parts of the test (parts of random generated workload).

2.9 Sandboxing

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

Obviously, there should be possibility to simply create a new sandbox or to restore previous state of the sandbox if it was broken by a test.

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

When testing is distributed (see Subsection 2.6.2), it should be considered to use the remote machine as the sandbox too. Again, there should be method to easily restore state of the remote machine when broken by a test. Note that this would be problematic with real (non-virtualized) hosts.

In Section 2.4 the problem of providing information about test state to developer was analyzed. If a sandbox is represented by full operating system then checkpointing of the 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 tests depend heavily on system state or change system state. Then without system snapshot some information to track the failure may be missing. Other case when full snapshots may be generated is when a test has caused system failure (therefore normal snapshot can't be created).

2.10 Filesystem test patterns

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

2.10.1 FSX

Originally written by Apple Computer, Inc. for MacOS and BSD-style operating systems. Nowadays, there are more versions used, but the main part stays the same [22].

The test consists of a single source file written in C, compatible with most Unix-based operating systems. The test operates on one file, does a loop with a 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. A failure report from FSX is a dump of operation sequence and a buffer dump.

2.10.2 LTP

Linux testing project [8] is a 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 2.10.1) and LTP specific 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 a vast amount of operations implemented.

2.10.3 OpenSolaris ZFS / NFSv4 Test Suite

Very exhaustive test suite for filesystems on OpenSolaris [10]. There are two sets of tests: one for NFSv4 and one for ZFS, but techniques used in them are the same. The only difference is that ZFS test suite has extra stress tests.

The system is Makefile driven, tests are generally shell scripts (ksh) with few support programs and tests that are compiled. Testing is deterministic, cleanup is done after a 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 the ZFS part, there are simple stress tests too - predefined loops with configurable length.

2.10.4 Mongo

Mongo benchmark [44] is a test program written in Perl, aimed at Linux filesystem performance and functionality testing (developed by Hans Reiser for reiserfs). It is very tightly coupled with linux 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 operations are correct, just that they goes.

2.11 Random workload generation

As described in Subsection 2.1.3, for complex applications (which filesystems indeed are) there is problem with defining and creating tests that would cover all use-cases and their combinations. One possibility how to deal with this is to generate a random workload to the filesystem that will exercise all operations

available in random order. By this (at least statically), all combinations can be tested.

The 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 (for filesystem that will be open, read, write, etc.), or small, well defined tests. It is sometimes wanted to group sequence of atoms to create new (bigger) atom.

Random workload generation has little usage on **stateless systems**. The only thing random workload can test on a stateless system is, if it is really stateless. In general, all possible pairs of operations should cover stateless system functionality.

In case of **stateful systems** (and meta tests), not all operations can be used in any state. So there should be method how to define and check states and legal transitions. Simple method to allow this is to give to tester a way to define **pre and post run hooks** that can initialize state, check transition, and possibly do cleanup after a test. While approach with pre and post run hooks is simple yet powerful, there is one issue connected to it. In this system, tester trades the possibility of using stateful tests for a potential waste of system resources. When a test expects some state different from actual, it must either made a state change (non trivial operation out of its scope) or silently pass without testing and let the system run another test. More sophisticated system for resolving statefulness is to give to tester a tool for defining **allowed transitions between tests**. Transitions are often given by a graph (edges can be allowed transitions or tests).

Sometimes it may be desirable to give some **preferences** (what should be tested). For the random workload this means either switching meta tests on and off, or giving preferences to tests. When statefulness 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 random workload can be restricted by:

- 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 stateful graph.

Chapter 3

The test suite architecture

We want to create a testing framework suitable for testing of userspace filesystem (which ZlomekFS is). This framework should be able to setup and control ZlomekFS so that every test run could have the same conditions (or at least as similar as possible). It should be able to run predefined tests and at the same time allows generation of random workload. For failures, the framework should collect as much useful information as possible. As for any software, the architecture of the test suite should be flexible enough to allow further extensions.

3.1 Programming language

Since ZlomekFS tends to be multiplatform and support more operating systems (currently only Linux is supported), language which has support for as many operating systems as possible is needed (or at least for UNIX-like operating systems). For testing, the language should be flexible enough, but on the 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 the existing code.

We decided to test ZlomekFS through general user api (access to filesystem) rather than calling directly functions of ZlomekFS code. There were two reasons for it: to allow internal refractorization of ZlomekFS code without modifications of tests (refractoring could be needed to allow integration with new operating system), and because of complexity of ZlomekFS code (tests would have to reflect this and thus they would incline to be tricky or incorrect).

The first option is to use pure C, because ZlomekFS is written in it. The main reason for C is possibility of integration. This is for sure required for unit tests. But for random workload generation, this requirement is rather

weak. As noted before, random workload will be generated through general user api (filesystem access). Moreover, it would be good to use some existing testing framework as base solution and write just ZlomekFS testing specific functionality. But we have not found any suitable framework written in C.

On the other hand, there are several testing frameworks written in **python** [14][1], that meet our needs. Compared to C, python is also more flexible language. Because of this, python was chosen as main programming 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 still written in ZlomekFS native language - **C**.

3.2 Used tools

List of tools finally used in the regression testing framework follows. For many parts, there is tendency to reuse existing tools. But for every one, there should be considered if the work saved by the use of existing tool worth problems with integration (it is unlike that existing tool would fit exactly into different, non-trivial project). For external tools with active comunity not only the work of writing the tool would be saved, but the maintenance of it can be shared too. Therefore mainly opensource projects with reliable community were considered.

3.2.1 Testing environment

For very specific purpose of testing distributed file system, no existing suitable solution was found. Before writing completely new framework, existing tool capable of extension was looked for. From general purpose testing tools written in python, **Nose** [41] 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

To provide best results, tests should be executed automatically in predefined periods or for every change in code. For this purpose, continuous integration servers (see Section 2.6) are used.

Buildbot [50] was chosen as continuous integration server.

[1] Author of this thesis is aware of other scripting languages that could be used. Python was favored because of personal preferences.

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 other tools 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 distributed filesystem, the support for distributed testing is essential. Moreover, ZlomekFS is intended to run on multiple architectures (and in future possibly on multiple operating systems) which requires distributed testing too.
- 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

As noted in Subsection 2.7, continuous integration servers in general can't store and present wide spectrum of data that can be generated from testing. This holds for buildbot too (test results are presented as textual output from commands). This is not enough for two reasons: when failure or error is found, the textual output may be messy (yet useful in some cases). Bigger problem is that textual output 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 is **Django** [4]. It is written in python, offers object oriented database abstraction layer and web pages generating tools. Its configuration format is pure python, so it is easy to integrate it with other parts of project.

3.2.4 Logging

Original approach used for logging in ZlomekFS was direct message prints (see Section 2.5). This approach is inappropriate for regression testing: there is no possibility of controlling the output and defining what should be logged. In general, direct message prints are the worst approach too and could be used only for very small projects where classic logger would be bigger than the project itself. This is for sure not the case of ZlomekFS, thus we need to provide new logger.

We need logger with these features:

- It can be controlled externally
- It has simple still full featured interface
- It has to have implementation (interface) for both languages (python and C)
- The output has to be in parseable and user readable format

This enforces us to write **new logger**, which will suit 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 tags 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

ZlomekFS is very complex and huge project with active development. Thus there is need to check internal api for stability too. As described in Section 2.3, unit tests can be used to check internal api stability. Moreover, unit tests have better traceability of bugs than random workload testing (yet they can't test complex functionality). Therefore it is better to find as many bugs as possible by unit testing.

Unit tests must be written in the language of code tested, thus C based unit testing tool is needed.

All existing C based unit testing tools found 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 **lack automatic test discovery**. Thus tests must be listed somewhere and collected manually (even in case of hierarchies). The best solution found is heuristic search by grep.

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

3.2.6 Documentation

For the programmer documentation, the best approach is to write it as close to the documented code as possible. This provides better maintainability, but with the documentation only within code, there could be problem to find particular information. Because of this, export to more user friendly format should be provided. To allow export of the documentation to external format (that should be more readable and may support searching), it must be written in a format recognized by some tool. Consequently, the choice of actual tool gives the format for documenting the code.

For documenting C code, **DoxyGen** [48] 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 import filters such as Doxypy [30], and even then results are not ideal. There is standard docstring format for python [27], 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 [31]). Their problems are mainly enumerated in the list.

Finally, **reStructuredText Docstring Format** [**?**, 32] was chosen. Main reason was that Nose [41] uses this format, secondary reStructuredText is easy readable in **Pydoc [12]** output and there are HTML formatting tools too.

3.3 Architecture

There are four applications used in testing: ZlomekFS daemon, Django (TestResultStorage), Nose, and Buildbot .

ZlomekFS daemon uses **Syplog** library to log debug messages, **D-bus** listener to notify about its status and Fuse library to integrate with linux kernel. Upon start, Syplog and zfsd reads configuration from program arguments. While zfsd is running, D-bus listener waits for connections through D-bus. If message or request is received, it is served either by Syplog (change of logging settings) or by zfsd status notifier (returns status of daemon).

Inside of any C code, unit tests can be written using **Zen-unit library** interface. When zfsd is executed with libzenunit.so library loaded (given by LD_PRELOAD), all Zen-unit based tests are found and executed. Results are printed to stdout, error messages to stderr.

All C code should be documented using **Doxygen**. Html documentation can be generated by calling *make doc* in root directory of particular project.

TestResultStorage should run on master machine, it provides web based user interface for result repository. Results are committed into repository directly (without usage of TestResultStorage process). Programmer documentation for TestResultStorage is again written directly to code, user should read it through *pydoc.*

When **Nose** is run, it loads plugins, searches for tests and executes all tests found. It is used to execute tests of all types (ZlomekFS stress and normal tests, Zen-unit based C tests, and unittest based tests of python components).

There are five plugins used in Nose. **ZfsStressGenerator** plugin is used to find meta-tests and generate random workload from them. This random workload is then given as test to nose to execute. **ZenPlugin** is used to search for Zen-unit tests. It uses Zen-unit library to search for tests in all passed binaries and libraries. Then these tests are executed and results reported through Nose (triggers ZfsReportPlugin). **ZfsConfig** plugin is used to load user defined configuration files. It loads given files into objects and passes these objects to tests. In specified moments (for example upon failure) **SnapshotPlugin** creates snapshots of test state. Plugin only triggers events and stores generated snapshots - the snapshoting intelligence must be provided by the implementation of test. When a test finishes, **ZfsReportPlugin** collects all information about the test run (snapshots, result, time elapsed) and commits them into TestResultStorage.

When a **test** for ZlomekFS is run by Nose, it forks zfsd instance. During execution, it can communicate with the daemon through D-bus retrieving status of daemon or changing logging settings. When test finishes, the zfsd instance is destroyed.

On all machines involved to testing, there must be **Buildbot** running. On master machine, it waits for changes in source repository. When a change is detected, master sends commands to slave machines, waits for end of their execution and retrieves results (only return code and textual outputs from tests,

snapshots and other status data are handled by plugins and TestResultStorage. On slave machines, the Buildbot process is waiting for master commands, executes them and sends results back to master. Main commands are fetching of code, building of binaries (rpms), execution of Nose (tests), and cleaning up. Simple results (return code and textual outputs) are then provided through web interface by Buildbot process on master machine. The interface shows both results of finished commands and actual status of all slaves.

All python code is documented in **reStructuredText** format, *pydoc* should be used to read this programmer documentation.

Chapter 4

Implementation details

This chapter describes key implementation details of the regression testing suite. It is not intended as programmer documentation. Programmer documentation is present directly in code (with possibility of export). Only general features and concepts are written up here.

4.1 ZlomekFS changes

4.1.1 Logging

Original message printing system of ZlomekFS (two verbosity levels, hard-coded) was replaced with **new logger** (*Syplog*) developed specially for ZlomekFS.

The logger allows programmer to print formatted messages into log. To distinguish severity of messages (and filter by it), Syplog (the new logger) supports eleven **log levels**. A log level is 32bit unsigned integer (typedef) which represents severity of message. Log levels are defined by macro constants (from LOG_EMERG to LOG_LOOPS). For example, message logging data input from user should be logged on log level LOG_DATA, message informing about system failure should be logged on log level LOG_ERROR. There is conversion function available to convert log levels to user-readable strings.

Similary to log levels, filtering by concerns can be done by **facilities**. A facility is either a part of application (d-bus service, logger, cache) or a logical domain going through all components (threading). Facilities are distinct one-bit flags that can be OR-ed. Log message can be labeled as belonging to any number of facilities. For example message notifying about acquirement of mutex for d-bus socket should be logged as belonging to facility FACILITY_THREADING and to facility FACILITY_DBUS. Then, this message will be logged if at least one of these facilities is set as to be logged. Again, there is conversion function to convert facilities to user-readable strings.

When message is send to logger, its log level and facility set are compared with current settings of logger. If severity of message (log level) is greater than logger's and at least one facility of message is set to be logged then message is written into log. Otherwise it is discarded. Both log level and facility set used for filtering can be altered in run-time.

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

Initial logger **settings** are read from program arguments (an array of strings with given array size). The logger ignores unknown options, so direct configuration from command line is possible (and currently used).

To allow **integration** with external driving component (Nose testing environment) there is **D-bus** [3] control api implemented. This allows adjustment of both log level and facility set from either control component or tests. For further integration with driving component, 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 added by implementing given function set and providing description structure. For further reference, see DoxyGen documentation of the *Syplog* library.

4.1.2 Zfsd status notifier

The driving component of tests needs to know in which state ZlomekFS daemon is (starting, running or terminating). Without this information, some heuristic assumptions may be done leading to false failures or invalid reports. To accommodate this need, **D-bus** [3] listener was added to zfsd.

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

- D-bus **provider** - stateful 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 ZlomekFS daemon (zfsd) 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.

When ZlomekFS D-bus service is running, any other application (with adequate rights) can ask for ZlomekFS daemon status.

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 the current zfsd message handler.

As the test driving component is written in python, there is **python client module** for this api too. The client module is automatically generated by swig, so there should be no problems upon extending the service.

4.2 Testing environment

Main component driving tests and controlling ZlomekFS daemon is implemented as **Nose** [41] plugins. Nose is a python unit testing framework, but it has very powerful plugin system so that it was possible to extend it to allow testing of filesystem operations, even random workload generation.

To Nose, **plugins** must be provided as python classes, listed as setuptools entry points under *nose.plugins.0.10*. They should inherit from *nose.plugins.Plugin* class, but it is not mandatory. Each plugin must implement some basic methods (such as *configure*, *options*, *help*). Then a plugin can define hooks for specific conditions. For ZlomekFS testing, we use mainly *want* and *loadTestsFrom* hooks to load tests from special sources (for example saved failure sequences), *startTest, handleFailure, handleError, addFailure, addError and addSuccess* hooks to control execution of tests, and few other miscellaneous hooks for minor fixtures. Nose itself and its plugins read configuration from environment variables and command line options. Command line options have superior priority.

Plugins mostly provide general functionality needed for testing and global state handling. Test specific configuration and setup must be implemented in test classes (ZlomekFS state handling is provided by base test class).

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 (plugins) too. This way is used to ensure that all tests within one batch 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.

4.2.1 ZfsTest

Where Nose plugins provide general functionality needed for testing, there control of *zfsd* (ZlomekFS userspace daemon) from python is implemented in *ZfsProxy* class (which is then used by tests to interact with the daemon). This class provides methods for starting zfsd, status queries, *Syplog* control, sane zfsd terminating and locked daemon killing. Tests are methods, grouped to classes with common setup and teardown methods.

Nose provide test fixtures (see Section 2.3) on all levels. For ZlomekFS testing purposes, all tests are for simplicity wrapped to classes. Then for normal tests, the setup of ZlomekFS daemon is done in test class *setup* method (run before each test) and the teardown (shutdown and cleanup) is done in *teardown* method (run after each test method). For stress testing, the setup is done in *setupClass* method (run once before all tests in class) and the teardown is done in *teardownClass* method (run once after all tests in class). How random workload is actually generated is described later in Subsection 4.2.5.

If a special configuration is needed by test, it should be written into a file in format recognized by python ConfigParser. Then, name of the file should be listed in used profile in the ZFS_CONFIG_FILE environment variable. The *ZfsConfig* plugin will then load all these files and pass them as object to tests. By this approach, it is possible to have multiple configurations for a test and specify by profile, which has to be used.

When writing new tests, **normal** ZlomekFS **tests** should inherit from class *ZfsTest* (which implements basic fixtures for zfsd). New filesystem daemon is started for every test. **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 remote *ZfsProxy* and *File* objects (to ease communication with remote system and performing actions on it). Communication is done through *twisted perspective broker*. Twisted [17] is 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 *se-*

tupClass method. Configuration for local and remote *ZfsProxy* is stored in file *zfs_client_server_config*, configurations for ZlomekFS daemons are in files *localZfsMeta.tar* and *remoteZfsMeta.tar*. These options are provided to test by *ZfsConfig* plugin.

The local *ZfsProxy* instance (local zfsd) is initialized in the same way as it would be in case of non-distributed testing. The remote *ZfsProxy* instance (remote zfsd) is initialized through calls performed on wrapper object. Tests then open local files as usual and remote files by calls to *RemoteControlWrapper* object (returning remote file instance wrapper).

In this manner, any topology of ZlomekFS providers and clients can be made. There should be *remoteZfs.py* running on every machine used in testing except master. On the machine where *Nose* will run, there should be configuration file for *ZfsConfig* plugin and on every peer there should be configuration tarball. Test setup then connects to all machines, configures their zfsd and following testing operates on local and remote file wrappers.

Since there are multiple machines involved in distributed testing, synchronization of versions of libraries and scripts must be handled. Currently *buildbot* is used for this.

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

Infinite testing loops are handled by insecticide in the same way as other tests. The 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

In order to find and fix bug causing faulty behavior, developer need data describing the failure. In general, the trickier it is to cause the failure, the more data is needed. In this aspect, random workload testing is one of the most problematic approaches. Sometimes, it is not possible to reproduce the failure. So exhaustive tracing and state information for the failure is needed.

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

When a test fails or an error is detected, **snapshotPlugin** creates a snapshot of predefined components. Snapshot plugin can be configured to create a snapshot before test and after successful test 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 consist of:

- Snapshot of ZlomekFS 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 a test class. Method gets *SnapshotDescription* instance as argument. *SnapshotDescriptio*n class has methods for appending primitive types, python objects, files and directories. Every entry in a snapshot has unique name, type and description. Primitive types are stored in memory, bigger data on disk. For purpose of reporting, a 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

As noted in Subsection 2.7, results of tests (especially failed) should be accompanied by (failure) state information and backtrace. For these data, there is result repository implemented.

Basic data about test result and relations are stored in **MySQL database**. Only failure state snapshot (content is listed in Subsection 4.2.2) is stored as a file in separate directory. For the snapshot, only its filename and relation to a test is stored to the database.

Since result repository and database are running on master, there must be method how to send snapshot to master too. Currently, **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).

Access to the database is provided through **Django** api. Schema of the database is given by classes (descendants django.models.Model), one class represents one table. Table columns are given by attributes of corresponding class, each attribute defining one column. Queries to database can be done by calling *filter* method of special class attribute named *objects*. Results are represented as (lazy) sets of objects, where for each column in given table the object has corresponding attribute.

We use *TestRun* table to store information about executions of single tests, *TestRunData* table to hold auxiliary information about tests executions - backtraces, exceptions, file names of snapshots. Set of TestRuns that were executed together are connected by *BatchRun*. BatchRun is represented as table, where one row holds common attributes for one set of tests such as name of the machine that executed these tests, profile name, repository branch and revision. Project and profile details connected to BatchRun are represented as foreign keys to tables *Project* and *ProfileInfo* (because they repeat a lot).

Settings of result repository are stored in *TestResultStorage/settings.py*. On master, access should be configured to use local database. On slaves, developer should alter their *TestResultStorage* settings to use master's database.

The testing environment (Section 4.2) is driven by **Nose**. We use its plugin interface to hook particular events during testing. To report results of tests, we use *addFailure, addError* and *addSuccess* hooks. Reporting to TestResultStorage is done by **zfsReportPlugin** which uses **ReportProxy** class as wrapper around BatchRun instance.

When there is an unhandled system error (python Exception), it is caught by outer try-except block and reported to the repository too.

The result repository has dynamic **web interface** which consists of listing pages for tests and batches (with simple filtering options), detail pages for test run, batch run, and project list page. If a snapshot is available for the test run, it can be downloaded from test run detail page. Older results can be deleted from administration interface (and there is script for automatic cleanup of obsolete entries provided).

To use central result repository, all tests must be executed under Nose and **zfsReportPlugin** must be enabled.

4.2.4 Options

Test set for execution 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 attributes of a test. For example expression '*not disabled*'

will discard tests where *test.disabled* exists and evaluates to True. Tests loaded from saved path ignore 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 a python object, and pass this object to all tests.

To handle **deadlocks** and infinite loops in both 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

For filesystem, it is very hard to define all possible use-cases that it could be used in. Moreover, the ZlomekFS daemon is stateful, using non-trivial caching mechanism. Because of this, some failures could appear only in case of very specific workload to filesystem. Since it is nearly impossible to write (all) such workloads by hand, the best approach how to test as many use-cases as possible is to generate random workload.

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

Format for meta-tests 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 be no problem).

If there are **dependencies between meta tests** (such as that open file test should run before read from file test), they can be defined by a graph. The graph format is python dictionary where key to dictionary is meta-test name and value stored is list of oriented edges originating in the meta-test. Edge is defined by the 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 terminating probability defined. Hard **length limit** of meta-test chain can be defined by plugin option.

If stress test fails, the path which has caused the 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.

A failed test is represented by path through the graph of allowed transitions from starting operation to operation where failure has been detected. In most cases, the failure was not caused by the last operation, but by the previous sequence. As the sequence was randomly generated, we can assume, that there are redundant operations in it and only part of it is actually needed to cause the failure. Thus it is logical to **strip the failed sequence** to see, if a shorter test sequence would cause the failure too. After a failure, zfsStressGenerator plugin try to do so either by finding shortest path through the graph, or by disabling operations (meta tests), or by skipping parts of failed walk.

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, just the ordering of their execution should be preserved. Execution order is given by plugin's class attribute *score* ascending. See [41] for further information.

4.3 C unit test

As described in Section 2.3, unit tests are small tests written often directly to code of application to test functionality of small parts. They can be useful as programmer documentation too.

Unit tests must be written in the language application is written in, so Nose can't be used here. We have not found satisfactory C based unit testing library, thus new unit testing library (**Zen-unit**) was developed.

The 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. *FAIL* is recommended return value but any test returning value different from *PASS* is considered failed.

To make the unit testing framework easier to use, we want to avoid the need for manual registration or listing of individual tests. The framework therefore scans all binaries for symbols exported by the ZEN_TEST macro and execute associated test functions. 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 the main file. This removes the need of tests listing, 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. Generation of test lists or test libraries (through swig) was considered, but found as redundant overhead.

4.4 Build system

To ease the task of building code into binary format and the task of installation, nearly all projects use so called build systems. Wihthout build system, programmer must write every single step to build binary manually. Build system is generally a tool, that automates these steps. Still programmer must define dependencies and describe steps in some meta language that build system understands. Actual binary is then build by calling build system's binary defining which step (target) should be build. Where basic tools provide just possibility of defining steps and dependencies there more advanced tools provide integration with actual operating system, installation and packaging beyond basic functionality.

Original build system of ZlomekFS was make. For just building, it was adequate, but there were issues with supporting installation on different platforms. Thus it was replaced with **automake** which can handle these platform dependent problems (for example difference between library directory on 32bit

system and 64bit system). Hence for C based components **autoconf**, **automake**, and **libtool** are used. For python based components **setuptools** are used. Setuptools are extension of python's standard build system. Setuptools are used mainly because of Nose, which requires usage of *entry-points* (registry-like tool of setuptools) for listing of plugins. For better compatibility with other tools, setuptools were wrapped into make system (actual work is done by setuptools, make only redirects calls).

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

To provide stable development cycle, tests should be executed automatically, on regular basis. As described in Section 2.6, continuous integration servers are used to fulfill this requirement. They either watch for changes in source code and build and test the software for every change, or build and test it periodically (for example every night - so called night builds). We have chosen buildbot [50] as continuous integration server for ZlomekFS (for reasoning see Subsection 3.2.2).

In buildbot, basic unit is a *build step* (typically one shell command, for example *make all*). Build steps are grouped to sequences. A sequence is called *builder*. Typical builder represents sequence of building a single application, testing it and cleanup for the application (for example builder for Zen-unit library is following sequence of steps: checkout source, configure automake, make rpm (contains make all), install rpm, run tests and upload rpm to server).

Watching for source code changes is done by a *change source*, which in case of ZlomekFS polls its svn repository for changes every minute. To define what should be done (which builders should be run) when actual change in code is detected, *schedulers* are used. A scheduler defines set of paths inside repository and set of builders that should be run when these paths change (again, scheduler for Zen-unit watches for changes in zen-unit subdirectory of actual branch and execute Zen-unit's builder upon change).

For ZlomekFS, buildbot is configured to create builder for every component on every host. Thank to build system unification, the build step sequence is equal for all components except ZlomekFS: it goes update - build - make rpm - install - test - upload. For ZlomekFS, the test step is separated since it should run only on one bot (others are used as peers for distributed testing).

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

Test driving (Nose) was included in way, that doesn't need any external configuration, only Django (result repository) 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 was needed. Thus in every ZlomekFS build, there is a 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 them to TestResultRepository. Then, control is passed to Nose (environment and command line options are preserved).

Nose parses environment variables and command line options and **configures enabled plugins** according to them. At this time, initialization phases of *SnapshotPlugin*, *ZfsConfig* plugin, *ZfsStressGenerator*, *ZfsReportPlugin* are executed. ZfsConfig plugin tries to load configuration files for tests (that contain for example paths for zfsd and zfsd configuration). SnapshotPlugin ensures that required directories for snapshots exists. ZfsReportPlugin calls TestResultRepository fetching previously created 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 them 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 the execution of tests begins, ZfsStressGenerator will append ContextSuite containing **stress tests** into *main ContextSuite* of Nose.

Then, execution phase is reached. For every ContextSuite (TestCase) its **context is initialized**. In initialization, *ZfsConfig* plugin passes **ConfigParser** object (representation of config files) to test. In case of classic (non stress) test, initialization of zfsd is done in setup and teardown methods in scope of method. In case of stress test, initialization of zfsd is done 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, exception 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 has 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 those related to threading only).

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

Reporting of classic test is 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** failure, *ZfsStressGenerator* 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 and reported.

When all tests have run, control is passed to **nose wrapper**, which **finalizes BatchRun** - sets its 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

Goal of this thesis was to extend existing ZlomekFS implementation by providing regression testing framework which would fit its special needs. After exploration of ZlomekFS, there was 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 make 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 were 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 remote objects were provided as wrappers for python twisted perspective broker objects.

By ZfsStressGenerator plugin, possibility to generate random workload to filesystem was 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. Failed sequences are saved for further usage.

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

Currently, changes of network conditions are possible only by inserting special rules into iptables. The changes of network conditions are mainly aimed

to check behavior in disconnected or slowly connected state. For future, there is d-bus interface prepared to allow direct changes to 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 usage 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 a userspace based filesystems, the modification needed would be only change of daemon binary and filesystem settings (which would be obviously different). If filesystem would have kernel component (especially in case of full kernel based filesystems) *kdump* integration for snapshoting state of filesystem in case of failure will be needed.

For unit testing of small parts of code (whitebox testing), small library for C was 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 is build target for generating HTML documentation available (moreover RPM packages with documentation can be build too). For python, language provided `__doc__` attributes with predefined syntax was used.

Beyond scope of this thesis, short installation guide for ZlomekFS and example configuration were made (there were need to extract these information from previous thesis text and create functional configuration by experiments before). To ease installation and management, project was splitted to smaller parts and packaging was provided.

Upon beginning of work on this thesis, ZlomekFS was running on very problematic kernel module and the basic functionality wasn't bug-free. Since then, ZlomekFS was modified (as part of other thesis) to use fuse based integration with linux kernel and some bugs were fixed. Nowadays, the basic functionality seems to be bug-free (among other things, author of this thesis has tried to run all filesystem testing tools available alongside with this thesis framework simultaneously on single ZlomekFS instance to bring about a failure with no success). Upon testing on inserted bugs, the output of framework were as expected, all bugs were found, pruning of random workload were producing very short sequences preserving faulty behavior (for example on bug that has caused failure upon sixth write, sequence of about hunderedth tests generated originally were shortened in about twenty cycles to eight steps - generate file name, open file and six writes). Note should be taken that the framework is mostly responsible for support facilities and random workload handling. The detection of failures depends always on implementation of particular test.

5.1 Further work

ZlomekFS network communication and request handling is multi-threaded. This is fine for real usage, but in case of testing it makes it hard to reproduce error since the way dispatch of request is done depends on switching of threads. Number of threads used is now hardcoded in source code. Thus, for further testing it would be better to change ZlomekFS to allow configuration of the number of threads in runtime (restring it to smallest number possible).

Currently, ZlomekFS sets connection mode (full, slow connection, or disconnected) on startup by measuring connection speed by ping messages. This is inconvenient for real usage (user can't set up mode by himself). For testing, connection mode can be forced by firewall rules, but it is superfluously tricky. Thus external control of connection speed (for example through d-bus) should be added.

For saved paths, there is only the sequence of tests saved. In most cases, this is sufficient to reproduce failure. But in case of testing behavior of conflict solving, the results depends on data written to actual files. Thus in future there should be support for saving testing data alongside with test sequences. This feature must have support in suite, but main work should be done by tests, because only tests themselves understands their data (and they may differ a lot between tests). Preferred method how to implement this is to provide another hook method to store data (like snapshot method of test class) and pair function to load them.

Main goal of this thesis was to provide regression testing framework capable of testing special features of ZlomekFS. This goal was met and example tests were implemented too. Despite this, for full coverage of ZlomekFS functionality there should be more tests implemented (for example there are no tests for capabilities, conflict resolution, slow connection mode, synchronization in complex hierarchies, etc). The framework supports testing of these features, the problem is that there is no specification of how ZlomekFS should behave in complex situations. Thus full analysis and specification of ZlomekFS behavior should be made before testing complex use-cases of ZlomekFS.

Appendix A

Coding conventions

C based code

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

Identifiers 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 [41] 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 clean Fedora 8 installation, installation to other systems may be different. In example, buildmaster has IP 192.168.16.253, buildslave has IP 192.168.16.252 and development system has IP 192.168.16.241. We assume that buildslave has basic development packages (such as gcc) installed. Note 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.

```
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/buildmaster \
 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).

```
...
@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, use 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/buildslave \
 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 its testing configuration.

```
cd /var/buildbot
svn checkout \
 https://shiva.ms.mff.cuni.cz/svn/zzzzzfs/\
 branches/zouhar/zlomekfs/tests/nose-tests \
 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, open 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.

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 in automatic builds.

To not spoil central TestResultRepository with your builds, create 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

Data on enclosed CD is structured to directories as follows:

- dist - distribution packages
- doc - programmer documentation generated by doxygen
- rpms - non standard rpm packages needed to install thesis and thesis rpms
- src - sources exported from the repository
- thesis - text of this thesis

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] D-bus. http://www.freedesktop.org/software/dbus/.

[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] Nfs testing tools. http://wiki.linux-nfs.org/wiki/index.php/Testing_tools.

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

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

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

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

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

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

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

[17] Twisted project. http://www.twistedmatrix.com.

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

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

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

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

[22] Jr. Avadis Tevanian. Fsx (file system excerciser). http://www.freebsd.org/cgi/cvsweb.cgi/src/tools/regression/fsx/.

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

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

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

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

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

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

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

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

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

[32] David Goodger. restructuredtext markup specification. http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[47] The Continuum team. Continuum. http://maven.apache.org/continuum/.

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

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

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