University of Warsaw
Faculty of Mathematics, Computer Science and Mechanics

Kornel Jakubczyk
Student no. 236034

Marauder disks detection

Master's thesis
in COMPUTER SCIENCE

Supervisor:
dr Janina Mincer Daszkiewicz
Institute of Computer Science

September 2010

Supervisor's statement

Hereby I confirm that the present thesis was prepared under my supervision and that it fulfils the requirements for the degree of Master of Computer Science.

Date Supervisor's signature

Author's statement

Hereby I declare that the present thesis was prepared by me and none of its contents was obtained by means that are against the law.

The thesis has never before been a subject of any procedure of obtaining an academic degree.

Moreover, I declare that the present version of the thesis is identical to the attached electronic version.

Date Author's signature

Abstract

Marauder disks are devices suffering from degradation of their performance. They can behave in this way because of latent errors, more precisely the ones that can be fixed by the hard drive's internal logic. Presence of marauder disks can lead to performance problems for a storage system. Moreover, it can also be an early symptom of a future fatal failure. The ability of earlier problem spotting would also simplify solving certain problems that the system administrator might possibly face.

This document describes design and implementation of the marauder disk detection component done for the HYDRAstor system, a commercial storage solution provided by the NEC corporation.

Keywords

Hard disk drives, performance, storage systems, latent errors

Thesis domain (Socrates-Erasmus subject area codes)

11.3 Computer Science

Subject classification

D.4.2 Storage Management
C.4 Performance of systems – Measurement techniques

Tytuł pracy w języku polskim

Wykrywanie dysków maruderów

Contents

Introduction 7

1. Hard Disks Reliability 9
1.1. Different Error Causes 9
1.1.1. Operational Failures 9
1.1.2. Latent Failures 10
1.1.3. Source of operation slowdown 10
1.1.4. Importance for RAID 11
1.1.5. Disk scrubbing 11
1.2. Error probability and SMART 11

2. The HYDRAstor system 13

3. Requirements and constraints 15
3.1. Hardware configuration 15
3.1.1. Cache 15
3.1.2. Hardware RAID 15
3.1.3. Physical node layout 16
3.1.4. Controller interface 16
3.2. Business requirements 16
3.2.1. Avoid false-positives 16
3.2.2. Avoid unneeded disturbance to the system's operations 16
3.2.3. Limit human actions 17
3.2.4. Low down-time 17
3.2.5. Test results accessibility 17
3.3. Technical requirements and constraints 17
3.3.1. Integrate well with the HYDRAstor system 17
3.3.2. Adapt easily to a different hardware configuration 18
3.3.3. Detect interference from other software 18
3.4. What is not required 18
3.4.1. Handling totally unresponsive disks 18
3.4.2. Detecting errors in data 18
3.5. Previous work 18

4. Solution 21
4.1. Two-level testing 21
4.2. On-line monitoring algorithm 21
4.2.1. Kernel I/O statistics provided 21
4.2.2. Selected values for monitoring 23

4.2.3. Choosing the right monitoring interval 23
4.2.4. Running average method 24
4.2.5. Additional constraint 24
4.2.6. Reducing false error report probability 24
4.3. Off-line tests 25
4.3.1. Overview of disks and file system testing tools 25
4.3.2. Method of submitting requests 26
4.3.3. Selected disk test method 26
4.4. Impact of DAC 27
4.4.1. Cache 27
4.4.2. Background RAID operations 28
4.4.3. Possibility of a DAC failure 28
4.4.4. Logical devices layout 28

5. Technology 31
5.1. Programming languages used 31
5.2. Libraries used 32
5.3. Important aspects of the code design 32
5.4. Scenario of actions 33
5.5. Getting information about present disks 33

6. Design 35
6.1. Overall module dependency 35
6.2. *DACTool* 35
6.3. *Options* module 36
6.4. *DiskTester* tool 38
6.5. Tool wrappers 40
6.6. *DisksProblemsData* 40
6.7. *DisksConfiguration* 42
6.8. *DiskCheck* tool 42
6.8.1. Invoking the tool 43
6.8.2. Important classes 43
6.8.3. Clean-up mode 45
6.8.4. Read results mode 45
6.9. On-line monitoring 46

7. Parameters' evaluation 49
7.1. On-line part 49
7.1.1. Foundations of the decisions made 49
7.1.2. Sample results of the monitoring 50
7.2. Off-line part 50
7.2.1. Variability of the tests' results 50
7.2.2. Dependency on the space available on disk 55

8. Tests 59
8.1. Unit tests 59
8.1.1. Googlemock 59
8.1.2. *Doctest* 60
8.2. Sample integration test scenario 60

8.3. Ensuring correct functionality 61

9. Summary 63
9.1. Future work 63

Bibligraphy 65

Introduction

The purpose of this thesis is to present a method for detecting certain error behaviour that could happen during hard disks operation, and to provide a way to report it afterwards. A disk that handles its read and write operations in a very slow way will be called marauder even if the requests may be successful.

Hard disks drives are very complex devices now. They are capable of performing quite sophisticated self-healing operations, e.g. relocation of the failed blocks, repetition of a read with slightly modified head position to use a bit different part of the magnetic material. On the one hand such recovery steps increase their reliability, but on the other they can increase the latency of the requests. They can also possibly hide important error symptoms from an application observing only the fatal failures of the operations. Therefore, a marauder disks can happen as a result of recovering from certain latent failures of the disk device.

Marauder disk can have a negative impact on the performance of a storage system because this aspect strongly depends on the underlying hardware. After receiving some error notification the user can replace the partially broken disk. This would allow a well-designed storage system to heal itself using the new device.

Moreover, it turns out that probability of fatal disk failure increases drastically after certain first error symptoms. Thus, decreasing the time of using a marauder disk for data operations would also increase the reliability of the storage system as a whole.

It can be time-consuming to diagnose presence of a marauder disk because the entire storage system could be quite complex. However, a hint that the device might be partially broken would directly present the source of the problem.

The reasons given should explain why an automated utility detecting marauder disks has been desired. Therefore, such component was created for the HYDRAstor system, an enterprise distributed storage solution provided by the NEC Corporation. Each storage node of this system uses hard disks as a main place to keep both user data and metadata. To satisfy all the requirements, most notably maintaining the resilience of the data, HYDRAstor needs to perform certain background operations on the informations stored on the disks. Naturally, the user write and read requests also have to use the hard drives. Because of this the disks are the important components of the system, and they affect overall performance a lot. This is why a marauder detection component is important for HYDRAstor. Here its design and implementation will be described.

The organisation of the chapters is as follows:

1. *Hard Disks Reliability* – a description of reasons of hard disks' failures will be provided to cast some light why a device can behave like a marauder, also will try to underline why the problem being dealt with is important.

2. *The HYDRAstor system* – will give a very quick glance on the system that the implementation will become a part of in order to provide some context for the reader.

3. *Requirements and constraints* – will present the most important both technical and business requirements that the marauder detection component has to face.
4. *Solution* – will describe the method that was chosen with some rationale for such decision.
5. *Technology* – will tell the reader about certain general aspects that appeared during the creation of the marauder detection component and about the technology used.
6. *Desing* – will show general module and class layout, also their responsibilities in the implementation.
7. *Parameters evaluation* – some results from the tests of the disks performance done to choose the required parameters for the marauder detection method will be described.
8. *Tests* – will give an introduction to the tests that were implemented to prove the correctness of the implementation.

Chapter 1

Hard Disks Reliability

1.1. Different Error Causes

Nowadays hard disks drives have become quite complex devices. They provide high-level logical block address space that is internally mapped into appropriate blocks. Built-in controllers can track and correct head position, internally queue and cache pending I/O operations, calculate erasure-coding on the data. Thanks to this, many errors can be fixed internally, and they do not bother the user. On the other hand, this means that internal device behaviour is complicated as well. In turn, this can lead to more complex processing in some situations, most notably under error conditions.

To better understand this problem, let us first take a closer look at the reasons why hard drives can fail. If the write operation aborts with an explicit error, the application can take appropriate counter-measures as defined in its high level logic, e.g. fail transaction, drop the operation, or propagate error to some higher layer.The most interesting situation happens when device cannot retrieve data that was considered to be stored correctly on the disk.

As described in [Ele09] such disk failures can be divided into two main groups called operational and latent failures; the categorisation presented there is as follows:

1.1.1. Operational Failures

In this group the error does not affect the data itself, however, the read operation fails so the information stored cannot be retrieved.

Damaged servo track

Servo tracks are stored on the disk's surface by its manufacturer. They cannot be normally rewritten afterwards so their damage is unrecoverable. Because servo tracks are used for the disk's head positioning, such error would cause a read failure, even though the data might be still correct. Servo tracks can be damaged in a similar way as any other ones so the reasons will be described in the parts following.

Head cannot stay on track

This is an interesting category due to its possibly intermittent nature. Depending on the current conditions previously observed error may not happen again; the possible causes include vibration and noise that might have ceased. Because the tracks are not ideally circular,

disks have a system for tracking the head position and correcting it accordingly to the track. Therefore, if the error is repeatable, the disk internal logic may be able to correct it.

SMART limit exceeded

SMART, i.e. Self-Monitoring Analysis Reporting Technology, is an interface provided by disks that allows to access information about internal hard drive's errors, especially the ones that do not cause requests' malfunction. If their analysis claims that the device is broken, the read operations will not proceed.

Electronics and head failure

The electrostatic discharge or other reasons can affect the electronics since they are quite complex in the disk device. Especially, read head can be damaged. Moreover, it can loose its magnetic properties, for example due to high temperature.

1.1.2. Latent Failures

In this group the actual data is damaged, however, the device and software are unaware of this happening. Such failures occur if either the write has failed, or data got destroyed afterwards.

Failed write

If the head is placed too high over the surface, the material may not get magnetised strongly enough, especially the old data may remain partially detectable. The cause of such error can be vibration, curiously as low as coming from other disks' activity.

There is also a non-intermittent version of this problem; happening when head's slider changes its aerodynamic properties, for example because of the lubricant, used in the construction of the device, gathering on it.

Bad media is handled by vendors so the faulty regions should be mapped-out with some padding around. Nevertheless, this process may leave some of the failing blocks in use since it may be hard to calculate the positioning of the bad sectors. Then any writes done to such blocks would silently fail.

Data destruction

The first cause is that the data can be damaged by short bursts of heat, that may be caused by a collision between the head and the surface of the disk where small bumps may be present.

Secondly, scratches and corrosion can be the reason of a failure. Particles from several sources can be flying inside the disk causing damage to the surface where a write occurred before.

Finally, in the so–called bit-rotting, the magnetic material may be loosing its properties as the time passes. However, it is not an important factor in practice.

1.1.3. Source of operation slowdown

Some of the errors mentioned have intermittent nature so a repetition of the operation can easily solve this. Moreover, disk devices have error handling procedures, for example in case of a read failure they can retry the attempt with slightly modified head location. Also, if some

failed sectors are found they are remapped to the backup ones, changing the data locality, thus changing performance as well.

The number of such recovery steps taken depends on the model of device. On the one hand, desktop class disks benefit from the increased reliability, but on the other server class drives can report the error earlier to reduce the latency in handling user requests.

Nevertheless, an increase of the operations' duration can happen.

1.1.4. Importance for RAID

Latent failures are a threatening problem for the RAID systems because they can lead to the so-called double failure scenario. This happens when a silent error in the data remains undiscovered until another visible failure occurs. Then normally RAID starts reconstruction. However, as there has been the initial error, the process may not finish successfully, and part of the data may be lost.

This is why the problem of latent disk failures has to be addressed in the implementation of the RAID controllers. For example, as described in [RAIDmonitor], it can be done by monitoring artificially generated disks requests. They are directed to the areas dedicated solely for this task, and their latency is measured.

1.1.5. Disk scrubbing

A well-known solution used to prevent latent errors is the disk scrubbing technique. It suggests to periodically read the stored data, even if the user does not ask about them. For example, such protection mechanism is a part of the HYDRAstor system (see chapter 2).

There is some research being conducted on how to do this efficiently, and how much protection can be offered. Note that some errors are caused by the operations of the disk so scrubbing would increase their probability. Nevertheless, the benefits are greater than the risk of failure introduction, as it is described in [LatentErrors].

For another example of such research, authors of [StaggeredScrubbing] prove in their paper that due to error locality characteristic scrubbing can be made more efficient by dividing the disk into small enough sections; then scrubbing can read only part of them unless some operation indicates a problem.

Also the vendors of the disk hardware do a lot to provide a high quality product. They design the racks for disks so that heat is reduced by ensuring good ventilation. Another issue here is the reduction of the vibrations. The shipment and installation procedure should also concentrate on avoiding early damage to the device.

1.2. Error probability and SMART

The paper [FailureTrends] contains analysis of disks failures and possibilities of making predictions about it using SMART. They make some observations about distinguishing faulty devices with SMART data:

- 56% of the failed devices showed no errors of the kinds considered the most significant.

- 36% of the failed devices showed no SMART errors at all. However, a major part of the disk population showed a non-zero number of such errors, making the possibly weakest condition of having any SMART failure ineffective.

The conclusion that was reached is that SMART is not enough to provide good failure monitoring.

However, four kinds of SMART errors do cause a big increase in the chances of future failure. Such analysis allows to infer that monitoring of disk's non-fatal failures is important because they may indicate upcoming problems. Therefore, the marauder detection component should indirectly help to avoid some read failures.

Chapter 2

The HYDRAstor system

The HYDRAstor is a distributed storage system featuring global duplicate elimination, provided by the NEC corporation. To give the reader a general overview, some of its features will be mentioned here, for a more detailed description refer to [HYDRA09].

The system is designed to support writing backup data with both high throughput and low latency. By applying proper cutting operation, a stream of data is divided into a stream of chunks. Now the parts that are already stored in the system can be reused in saving the current stream. This process is known as the duplicate elimination. The use of the content-addressing paradigm allows to efficiently implement this, increasing the logical capacity for the backup data.

Features

A secondary storage solution faces different user requirements than other kinds of storage. The time window when the data can be saved should be as short as possible because backup might limit the access to the user's system. This makes the performance of the write operation the most critical one as it is the function executed most often. Read throughput is important for the recovery procedure. HYDRAstor also handles data deletion.

The concept of the distributed hash table is employed to offer a high level of scalability and error recovery. The first comes from the fact that users can gradually increase the capacity and performance by adding new servers for storage and handling data access, the back-end is designed to potentially contain even thousands of nodes. The second means that the system automatically handles failures of disks, network, entire machines, and even some software errors. The marauder detection component adds another functionality that would detect an early disk failure so it is aimed to further improve this particular aspect.

The system monitors resilience of the data stored in it. Therefore, failures are not hidden until the read operation. It reports a global state with information about how many disks and machines can be lost before a data loss would occur.

HYDRAstor supports various resilience levels for the data stored in it. This allows the user to select the right balance between the space taken and the amount of safety. The Reed-Solomon erasure codes are applied in order to greatly increase the safety of the data.

High availability is achieved by the distributed and decentralized organisation of the HYDRAstor system; on-line upgrading of software and hardware is another step in this direction. Extending the system does not require any down-time. A new node can be added to the system while write operations are in progress. Then their performance increases as the new machine begins to get utilized. New protocol drivers can be added on-line allowing to handle a new format of the input data.

HYDRAstor does global duplicate elimination, featuring blocks of variable length, what allows for content-based stream chunking. In this method the fragments generated can be better adjusted to the patterns in the data by allowing a more intelligent cutting operation. In effect better ratio of the duplicates found is gained. The elimination process is done in-line, during the write operation, to increase the performance as duplicated blocks do not have to be stored again.

The system is composed of server-grade machines that offer high reliability. Each node is responsible for the portion of the data that is assigned to it proportionally to its abilities. The load-balancing spreads the components, the elements that contain the data, so that resiliency and performance are maximized.

This is just a quick enumeration of some of the important features, more details can be found in other papers([HYDRA09]). The important thing for this work is the fact that the HYDRAstor system is a commercial high-quality storage solution that utilizes the hard disks to provide its services, and therefore, it depends on their performance in servicing user requests.

Location of the marauder detection component in the system

The marauder detection component, which is the subject of this thesis, is located on the back-end part of the system, where hard disks drives that store the data and handle user requests are present.

Each machine, forming the grid of the back-end infrastructure, is called a physical node. It serves as a host for more than one server instantiation, known as a logical node. Each logical node handles its own hard disks that are going to be monitored to detect possible marauders. Other logical nodes have much less influence on those drives, and they may be using different hardware components. Therefore, the main area of interest will be the scope of a single logical server. However, due to the reasons described in the following chapter, some actions require knowledge and operations on the level of the physical node.

Location of the elements forming the marauder detection component inside a single physical node will be presented in chapter four.

Chapter 3

Requirements and constraints

The most important requirements that affect the design of the marauder detection component will be described in this chapter. They can be divided into two main categories: business and technical ones. Both groups are strongly influenced by the fact that the component will be part of the HYDRAstor system, described in the previous chapter.

The main requirement of marauder disk detection is to recognise a decrease in disk performance that is not caused by an activity of hardware RAIDs or the operating system, but indicates an error in the hardware. The main task is to spot a failure of a single device, however, hinting that the disk controller may be broken would also be useful.

3.1. Hardware configuration

One of the key factors that affect the design of the marauder detection component is the underlying hardware configuration. In the basic one HYDRAstor logical server uses 6 SATA hard-disks drives that are connected through a SATA controller, which will be called shortly DAC. This controller provides vital services for the system, however, its presence causes some complications to the design of this component.

3.1.1. Cache

Read and write operations normally are cached in the controller in order to increase the performance. This cache is shared among all the disks. If we now consider a situation when one device has a significantly worse performance than the rest, the cache would be filled with requests going to this device. That could slow down requests directed to the other disks because now they cannot use caching efficiently. This implies that noticing a slowdown of a particular device does not mean that this is a marauder detected.

3.1.2. Hardware RAID

DAC provides hardware RAID implementation built on top of the physical disks connected. Such entities are exported as logical devices, similar to the ones representing physical disks. Thanks to that, the operating system sees them as normal SATA devices so they are very easy to use by the applications. However, for such controller load on the RAID partition gets ultimately converted into load on some physical disks, and the operating system does not have the right knowledge to report it correctly. This introduces a necessity to identify which logical disks are actually hardware RAIDs and possibly what they consist of.

Still, if disk devices are configured into so-called software RAID, implemented by the operating system, the load on each disk is reported correctly.

To maintain proper data protection RAID's implementation has to perform certain background operations, most notably reconstruction of data in case of an error detection. Such actions generate heavy load on the underlying disks. All these operations are done internally so the operating system again knows nothing about them.

3.1.3. Physical node layout

A typical physical node layout consists of two disks controllers, each having its own disks and similar configuration.

Depending on the HYDRAstor system configuration, one DAC can be used exclusively by one logical server instantiation or shared between more of them. Here arises a need for information which controller is the given server using, and whether it is shared with any other logical node.

3.1.4. Controller interface

Hardware vendor provides a command line utility to access disk controller's configuration. It is utilized in marauder detection component to read the DAC set-up and current state, also to perform any required operations.

Different controllers may come with different interfaces and slightly different tools.

3.2. Business requirements

3.2.1. Avoid false-positives

One of the key aspects of the system design is to require as little human supervision as possible. This reduces the actual cost of using the system, therefore, is a big benefit for the client. However, as marauder detection component deals with hardware failure, it is justified to expect some physical actions. The purpose of the design is to suggest some recovery to the maintenance personnel of the client's system installation. False alarms would then increase the cost, and perhaps even worse, they could lead to overlooking the actual correct report.

Please notice that even without disk replacement, the HYDRAstor system as a whole will function, and it will provide services for the clients, yet perhaps with reduced capacity or performance. This component should quicken restoration of its full potential. The system is well and carefully designed to handle replacement of a broken disk with a new working one. Another benefit of a reaction to the earlier error symptoms is reducing the time period when a nearly broken device is used, in effect this reduces the possibility of a failure due to hardware problems.

3.2.2. Avoid unneeded disturbance to the system's operations

Although detecting marauder disks is, as we can see, desired for the system, it is still an action performed in addition to the more important ones. During normal operation HYDRAstor system is busy with handling user requests, and even if there are no such activities, this does not mean that the system is idle. It may be performing internal tasks that involve checking and moving data to maintain its resilience, namely also disk scrubbing. As the HYDRAstor is a distributed system, a machine may be required to take part in actions being initiated on other servers. With highly utilized system, it would be difficult to find a convenient free time

window to perform some additional actions required by the marauder detection. However, this component is mainly focused on dealing with atypical conditions, during which some extraordinary actions may be necessary.

In conclusion a combination of unintrusive operations in a normal situation with the possibility of performing more costly actions when it is required would create a good solution for the problem of marauder detection.

3.2.3. Limit human actions

Due to reasons described in the previous sections, the marauder detection component should not require any special operations in case of the system working correctly. However, we can expect that the user would take some steps in case of a possible disk failure.

3.2.4. Low down-time

High-availability and low-latency are among the key clients' requirements. This component cannot be allowed to cause unnecessary drops in handling user requests. Therefore, even if some errors are suspected, the inactivity time of the machine should be as short as reasonably possible.

3.2.5. Test results accessibility

The tests' results should be readable regardless of whether HYDRAstor logical server is currently running because that logical node may be stopped due to a marauder disk presence. Also, this data should survive systems restarts and if possible some disk errors. This requirement implies that the result access operation cannot be done only in the HYDRAstor logical server.

3.3. Technical requirements and constraints

3.3.1. Integrate well with the HYDRAstor system

The purpose of the marauder detection component implementation is to provide an additional useful functionality for the HYDRAstor system. This has a huge impact on the technology chosen to implement the solution, the libraries used, and on the design.

High level of integration brings some benefits, especially it allows using already implemented elements. Among the most important ones are:

- the build system,
- the framework for unit testing and integration testing,
- the HYDRAstor configuration support.

Actually, this component must use all the facilities mentioned in order to be accepted as part of the HYDRAstor system.

User interface

All of the user interaction in the marauder detection component is handled in a uniform way as any other components of the HYDRAstor system. Therefore, the design of user interface is not presented here. If we would think about classic Model View Controller paradigm, the

entire implementation of the marauder detection component, presented in chapter 6, lays in the Model layer. This allows to limit the communication with the outside components to only exporting well-defined data objects and allowing to trigger required actions.

Communication is organised in a way similar to the RPC model, requests from the higher layers are received, and a response may be provided. For the output part, the marauder detection component only has to decide what part of data will be made available and export that in form of C++ objects. Then appropriate higher layers will handle the task of presenting them. The input part is actually reduced to receiving a request to trigger a required action that can possibly contain some defined parameters.

3.3.2. Adapt easily to a different hardware configuration

With the technology constantly progressing some changes to the hardware are to be expected, especially using faster and bigger disk devices. However, models of devices that will be installed in the system should be known beforehand. This will allow to prepare some configuration for the new components.

3.3.3. Detect interference from other software

Many things can cause disturbance of a single application's performance. Most notably, the reasons can be other programs' activities and some operating system's actions. Those are hard to be distinguished from hardware errors if observations are based only on data provided by one application. This is why a source of information closer to the operating system is needed.

3.4. What is not required

Sometimes, it can be a good idea to state what a component should not do in order to gain better understanding of its scope.

3.4.1. Handling totally unresponsive disks

This situation is easy to handle, as the failure is a definite one. However, it is important to remember that it should not cause failures in the operations of the marauder detection component that can be avoided.

3.4.2. Detecting errors in data

As we already know, before actual failure occurs, there may be a slowdown of performance, caused by the disk trying to recover from non-fatal errors. Therefore, detecting marauders is somehow connected to detecting any disk errors. Apart from that, the HYDRAstor system already contains appropriate countermeasures for detecting and handling read errors so it makes little sense to duplicate this functionality.

3.5. Previous work

Before the solution proposed will be described in the next chapter, an overview of the previous work will be provided. It turns out that the topic of disk performance and reliability receives significant attention. Some aspects have already been presented in chapter 1. The areas of work connected with marauder detection are:

- Analysing latent disk errors, and ways to protect against them, e.g. [IntraDisk], or [LatentErrors], and other works about disk scrubbing. There is also research on constructing better devices done by the hardware vendors.
- Means for predicting disk failures using various artificial intelligence methods, and the accuracy they can achieve, e.g. [FailureTrends].
- Preventing degradation of performance in RAID systems, also preventing the double error scenario, an example is given in chapter 1 ([RAIDmonitor]).
- Disk requests' latency is an important measure in constructing high-performance systems, i.e. database servers. There are tools for profiling it (e.g. see [ORION]), and also for monitoring it used by the system administrators.
- Disk performance benchmarks are developed in order to compare various new device models and file systems. However, both these tasks are not among the requirements of the marauder detection component.

Chapter 4

Solution

4.1. Two-level testing

To meet the requirements stated in the previous chapter a two-level solution has been proposed. The key idea here is non-intrusive monitoring that watches over the disks' performance. It can suggest more expensive actions to be taken in order to confirm that observed anomalies are actually errors.

First part is done using operating system statistics. It is the on-line monitoring that is being performed while the HYDRAstor logical server is working. It is able to notice symptoms of a marauder disk presence. However, it is not expected that on-line monitoring suggestions can be always trusted.

Second part executes short disks' performance test. It is quite difficult to be reliably performed together with normal logical node operations so it is done while the required servers are stopped. Because of this, the off-line test takes more time to execute and effectively uses up significant resources. In exchange it should provide a result that would be reliable.

Figure 4.1 presents an overview of the physical node's organisation, where the marauder detection component is situated; the actual number of elements can differ depending on the configuration of the system. DAC is a hardware disks controller, described in the previous chapter.

The figure shows the marauder detection component consisting of the two main parts: on-line monitoring and off-line tests implementation. The first one is included in the logical node while the second one is a separate entity. In general, off-line part does not work together with the logical nodes using the DAC controlling the disks to be tested. The test tool can have access to all the DACs, also it is possible that two tools would be running at the same time – each one using a different DAC.

4.2. On-line monitoring algorithm

4.2.1. Kernel I/O statistics provided

In order to avoid pitfalls mentioned in section 3.3.3, a more reliable source of input data is required. It would have to include the impact of the system activity but also interference from other applications running on the same machine.

Luckily, the Linux kernel provides some information about performance of disk devices (for more detailed description see [IOstats]). They are most useful for selecting and tuning I/O scheduler for the system. Let us take a look at what kind of data they provide:

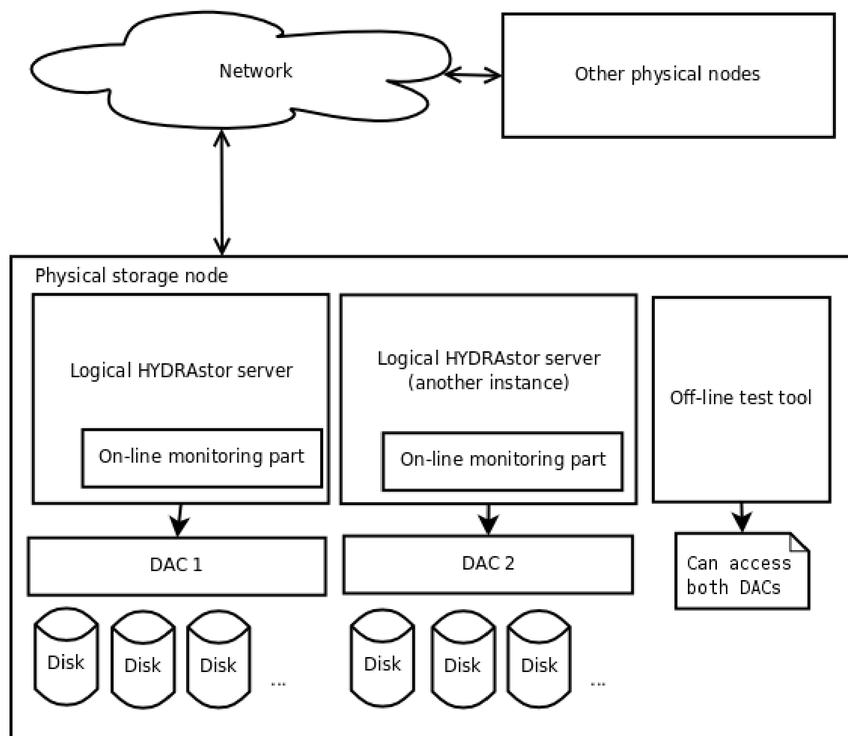

Figure 4.1: Location of the marauder detection component in the HYDRAstor system; it consists of on-line and off-line parts

1. Number of read/write requests completed – quite interesting, as it shows whether disk is doing any useful work.
2. Number of requests merged – perhaps useful from the I/O scheduler point of view, however we do not really care about operating system's internal actions.
3. Number of sectors read/written – this gives the amount of data that was actually processed.
4. Time spent reading/writing.
5. Number of requests in progress – not really useful for us, as it has typically low values, and it depends mostly on the point in time when the statistics have been read.
6. Time spent doing I/O – interesting as it gives us an idea about how active the device has been.

This list applies to the whole devices, for individual partitions reported data can contain less positions depending on the kernel version. However, we are interested in summarised statistics for the entire device.

Statistics data can be obtained by reading the "/proc/diskstats" file, conveniently for all the disk devices at once. Values are returned in form of 32 bit counters. To use the statistics the following steps are needed:

- read the proc file,
- wait and measure the real time elapsed,

- read the proc file again,
- subtract the counters' values.

Note that there is a source of some inadequacy between the time measurement and the moment of doing a read from the file. Implementation has to be aware of the possibility of an overflow in the counter. It turns out that due to the properties of the integer value representation, a simple subtraction of 32 bit numbers does this correctly.

During the experiments with these statistics it turned out that they may not be updated in an uniform way in the environment that the system is running. Regular picks in the number of requests completed were observed, as if a group of requests was reported as completed in short amount of time, and after that a new group was started. This implies that the interval between measurements cannot be too short.

Libraries used by the HYDRAstor system allow to handle statistics gathering almost in the way that was desired for marauder detection component, they required only an addition of exporting some values in a raw format.

4.2.2. Selected values for monitoring

It has been decided to concentrate on the number of requests completed by a single disk. They provide simple to analyse information about whether useful work is being performed by the hardware. Obviously time has to be taken into account as well. Fortunately, the kernel I/O statistics include the counter of time during which requests were being processed, and the actual value to be monitored is the time spent working in relation to the number of operation successes, this is an approximation of the average request service time. This measure should significantly increase when a device would become a marauder.

After analysing the statistics provided by the kernel, some observations about possibility of false positives can be made. The first one is that when a device has a very small number of requests to process, comparing to its abilities, they can take significantly longer. This is why utilization of the drive has to be monitored as well. It can be calculated by taking the active time counter and dividing it by the real time elapsed.

When utilization is too low the measure of average service time is invalidated. As a marauder disk would take long to process request, the active time would be high, and utilization should increase. This reasoning proves that such additional check should not influence the ability to spot the actual error.

4.2.3. Choosing the right monitoring interval

As described before, the method of accessing the kernel I/O statistics requires selecting some interval between the counters reads. There are some benefits of choosing a longer period:

- The proc file will be read less frequently so it would cause less interference to system's operations.
- More requests will be taken into account, in effect an average of more samples will be calculated giving better result. We expect marauder disks to be showing its behaviour for some significant period of time.
- Only an approximation of average request service time is available that is calculated from activity time. However, this counter gets increased also for uncompleted requests. Too short measure time could lead to an increase in the ratio between request actually

completed in the measurement window and the requests only started in it. This could result in an undesired rise in the average service time.

On the other hand, HYDRAstor system's behaviour may have phases of performing some disks activity followed by periods of idle time. In such situation very long monitoring interval would render the phases indistinguishable. Then the low utilization detection could classify entire sample as idle time, effectively disabling the monitoring mechanisms.

4.2.4. Running average method

To conquer the problem mentioned before, and still keep most of the benefits of the longer period selection, taking advantage of the running average properties was proposed. This involves using a shorter monitoring interval that allows for fine-grained removal of samples without high enough utilization.

A selected number of non-idle measurements is summarised to form a longer one, giving the benefit of getting better statistical properties. Samples are organised in a queue; when a new one arrives, the oldest one is removed. The queue length is determined by the time span of all the measurements in it, and a decision whether to remove any sample is based on the length of the real time window that we would like to normally observe. This guarantees that even if many samples would be rejected, the algorithm would not allow grouping too distant ones together. Also, such method produces results that are more similar to having a long monitoring interval from which parts of low disk activity are removed, what is actually desired here.

4.2.5. Additional constraint

The previous sections describe a reasonable way of monitoring disk activity. However, during both the artificial and real system tests, it turned out that there is a situation with significant increase of the average request service time (such monitoring results are presented in figure 7.5). It could happen when after some period of high activity disk quickly becomes idle. The problem lays in the border sample between the busy and idle time. The utilization may be high enough to pass the non-idle test described before, and some final requests can take much longer than expected.

Increasing the sample duration could help a bit, but only by reducing the probability of such failure. The running average method allows for a simple extension that would handle this exception. If it detects a few samples with high utilization followed by a sample below the idle threshold then the last high activity sample is disposed of. Notice that the possible anomaly can occur in either the last valid sample of the busy period or in the first sample of the idle time following. Therefore, this additional check together with the low utilization filtering will invalidate both possibly wrong measurements.

4.2.6. Reducing false error report probability

To give control over the possibility that some rare and unrepeatable error condition would keep causing monitoring failure while the system is able to work normally one more check is added. Marauder possibility notification will be generated if error conditions occur frequently enough. Both the warning number and length of the time window used here are configurable, as described in chapter 6. This also allows the monitoring system to be closer to what a human would do when given similar disk performance data.

4.3. Off-line tests

The most important factor in implementing the off-line part is providing correct environment where load on the disk being checked is coming almost exclusively from the test being performed.

The other important task is finding some balance between the duration and reliability of the tests. A hard disk contains big amount of data scattered across its surface. As the errors can affect the device only locally, they may be noticeable only during long test using big parts of the disk. However due to the requirements stated such approach cannot be taken. As we will see, write tests are designed to test the space that is most likely to be used in future requests. The read tests' role will be to get an overall picture of the entire device.

4.3.1. Overview of disks and file system testing tools

There already exist many tools for performing disk tests. We will shortly describe two for a quick overview.

hdperf

Hard Drive Performance benchmark is a quite simple cross-platform tool created in an open source project (see [HDPerf]). However, it provides all the basic functionality. It features simple random and sequential tests, only reads in the current version. The disk can be divided into a few (64) zones, for which results are reported separately. This allows to address the issue of performance being dependent on the physical location on the drive. Another important feature is the ability to test different sizes of the requests. There have been plans to support testing disk devices with bypassing the operating system's cache (by using the *O_DIRECT* flag in the *open* system call's arguments).

Bonnie and Bonnie++

The first one was a very well known disks benchmark application for the Linux system. Later one is a new branch, created in C++, that delivers support for testing bigger modern disk devices (see [Bonnie++]). It allows to perform a few kinds of tests, like:

- sequential output, also in mode where data is rewritten,
- sequential input,
- random seek,
- file creation and deletion.

First two kinds support two operating modes – one block or one character at a time. These benchmarks also measure CPU utilization.

Conclusion

There are even more complicated benchmarks that feature sophisticated checks. Some of them are designed to prevent certain simple optimizations in the file systems, solely dedicated to improving performance in artificial scenarios, from affecting the results of the test. This is important because otherwise benchmark would give unrighteous advantage to one file system over another. However, the marauder detection component is not really concerned

about testing the file system's performance. This layer will impose a constant impact on the underlying disks' performance because it is standardized in the HYDRAstor system.

From quick comparison of the both benchmarks above, it is clear that most of the functionality that is necessary is already included in the hdperf. We are not interested in CPU usage. Reading one character at a time is not a real world scenario, it tests things like implementation of the standard library. Rewrite test would actually measure benefits coming from operating system's cache – but we are not interested in testing the kernel layer. This convinces us that the test method does not have to be very complicated.

Moreover, due to the low down-time requirement, some further simplifications are desired. The size of a single disk request can be fixed. It is chosen to approximate a typical operation of the HYDRAstore system. By using just one length, a greater number of similar requests can be issued in a limited time. In effect, as an average from a bigger sample is calculated, the result value is more reliable in terms of representing a chosen type of disk operation.

4.3.2. Method of submitting requests

For performing the test scenarios an appropriate method of submitting disks requests has to be chosen. It has been decided to use asynchronous input-output because it has good support in the proprietary library used in the HYDRAstor system.

An important aspect here is the decision to use the mode with the *O_DIRECT* flag set. It allows to remove the impact of the operating system's cache on the disk requests. Therefore, it makes the test more dependent on the hardware and less affected by the upper layers. The marauder detection component aims to test the disk directly so if by accident some data would be cached by the operating system then request would finish very quickly, perhaps causing error in problem detection.

This decision can be further justified by the fact that there are only two disadvantages in the implementation:

1. Requests have to be aligned to the block size of the operating system and need to have the length being a multiplicity of this block size. This is a rather unimportant constraint because the test's requests can have whatever size and align we want, as their results are not used. It is only a simple thing to ensure in the implementation. It is not desirable to test smaller or unaligned requests as they are rarely performed by the HYDRAstor system, also the operating system can merge and align them appropriately using its I/O scheduler's algorithm.

2. Direct mode does not work for remote file systems. This issue affects only some of the test environments, and it does not occur in the production configuration. With slightly more careful design of selecting the write path in tests this problem can be simply avoided.

4.3.3. Selected disk test method

Four kinds of tests were selected, those are sequential and random both reads and writes. With sequential tests chosen to measure raw throughput and random tests to measure seek performance it is expected to cover most of the disk functionality.

Write tests

The problem with doing writes on a working system is to avoid accidental corruption of its data. With a mounted file system direct modifications on the level of block device seems to be

really dangerous, especially that there may be some other programs doing their operations, also a flush of the operating system's cache may be happening. Because of that, it has been decided to issue writes through the file system layer. The main advantage of this is safety and simplicity of reverting all the changes done.

There are two main issues that arise here:

- Necessity of clean-up action – it is vital to ensure removal of the temporary write file despite any errors that may happen during the test because the HYDRAstor system monitors free disk space on each device. Some errors are quite likely to happen with a marauder disk present, for example time-out conditions.
- It would be hard to ensure a fixed location of the write file on a disk, and troublesome to check its placement afterwards; also the file may get fragmented during write.

The first one has to be dealt accordingly by the design and implementation.

Surprisingly, with the second one comes a related benefit. The location of the write file should be more or less similar to the location where new data would be placed. This becomes quite significant because of the requirement of using a short test length. This way we focus on the disk's parts whose failure would be most critical for the performance of the system.

Read tests

Testing reads through the file system layer would be quite difficult and would impose many problems, like:

- finding some files to read,
- dealing with files too short for test,
- finding placement of file on disk because it influences the result.

Here we can take advantage of the Linux's block devices support. By reading directly from them, we bypass troubles with checking many files. Please notice that it is not a problem whether we read from a block that contains some valid data or not because we are not really interested in checking the read result. Only the performance is important.

A small disadvantage is that super user permissions are required to access such device, but for safety reasons it is desirable to limit the number of actions and the amount of code performed with higher privileges.

4.4. Impact of DAC

For both on-line and off-line parts there are important exceptional situations when their work cannot be performed due to the possibility of generating false positives. As described in chapter three, DAC provides some level of abstraction internally that we want to take into account to allow correct marauder detection component's operations.

4.4.1. Cache

The disks requests are cached in the DAC layer so this strongly affects their performance in a hard to predict manner. For tests focused on the disks only, it would be beneficial to disable this functionality. That will be done in off-line testing.

However, as the on-line part is functioning together with the HYDRAstor logical server, the cache has to be enabled. This factor will have to be included in tuning the expected performance thresholds for the monitoring. This is a key reason why a degradation of a single disk's performance could also imply a degradation for some other ones. Because of that on-line monitoring is not expected to tell exactly which disk is the faulty one. This shows why a combination with off-line test is designed to solve this problem.

4.4.2. Background RAID operations

To maintain safety of the data stored on a hardware RAID device in case of a disk failure DAC has to perform a rebuild operation. As this task is done internally, the operating system will not notice it, and a marauder possibility could be wrongly reported by the on-line monitoring. Moreover, marauder presence could be incorrectly indicated by the off-line test. Therefore, it is vital to detect periods of time when the RAID background operations are being performed, as they may lead to false-positives in marauder detection.

Periodical checking if background RAID tasks are running will be done using a tool provided by the DAC's vendor. The information obtained states that either nothing is currently going on or that some actions are being performed now. With checks frequent enough, we should be sure that the disks are not busy with any background tasks.

In on-line monitoring a sample is considered to be correct if both the first check before it reports no activity and so does the first one after the sample. This introduces the need for delaying processing of a given measurement until a next check of the RAID background tasks state is performed.

4.4.3. Possibility of a DAC failure

It may rarely happen that degradation of the disk's performance is not actually caused by the device but rather by the DAC layer. Although such situation is not a main focus of the marauder detection component's requirements, it has to be dealt with. It has been decided that if in off-line testing too many disks would seem to be broken, they will not be reported as marauders. Instead an information about too many failures will be given to the user to let him diagnose the problem. Such approach is acceptable, yet it still requires some human action. However, such situation probably indicates a serious error condition.

4.4.4. Logical devices layout

RAID logical devices

Because hardware RAID logical devices are build on top of underlying physical devices, any load on them would translate to some load on the disks. Then disks would appear to be working slower, but they would be handling other requests in the meantime. The actual amount of requests submitted would depend on the level of RAID used in configuration, implementation of the RAID, location of the parity drives for a particular part of data. Therefore, it is hard to be predicted. Also, one physical disk can be used in a few RAID configurations.

To prevent this problem, during both on-line and off-line parts' operations all the logical devices that are hardware RAIDs will be monitored. If the load on them would be high enough to influence the results too much, the following actions will be taken: the results will be invalidated for the off-line tests, and current samples will be removed from the on-line monitoring.

Normal logical devices

Non-RAID logical devices are mapped to the physical disks by the DAC controller. However, in part of the system configurations more than one logical device can be mapped to a single disk. Even then most of the requests should be scheduled through a single logical device.

For the off-line tests this is not very important because it is enough to check just the main logical device, meaning the one containing the stored user data, and distinguishing is not a problem.

In the on-line monitoring, handling such configurations requires summarising the load of several logical devices, as the operating system sees each of them separately. Real time is the same for all the devices, as we read one snapshot of the proc file. There are also two kernel I/O statistics that are used: number of requests done and active time. The first one can easily be summed. The second one is impossible to be evaluated correctly so it has been decided to use a maximum of the available values. This decision allows to stay on the safe side of the error range in terms of false-positives. As the load on auxiliary partitions should be mostly small when compared to the load on the main one the introduced error should not affect marauder detection component's usability too much in such configurations.

Chapter 5

Technology

This chapter will focus on the programming languages and libraries selected for the marauder detection component together with certain issues that follow that choice. Some decisions made for the entire implementation of the on-line and off-line parts will be presented here. The chosen monitoring and testing methods have been described in the previous chapter. Also a scenario of the actions connected to the marauder detection component will be given to show their logical order.

5.1. Programming languages used

Most elements of the marauder detection component are implemented in C++. This is a strongly-typed programming language that offers complex features to an experienced user, such as support for the so-called meta-programming, partial template specification, virtual functions, operator overloading, and even multiple inheritance. C++ is very popular, and there are many good references about it, e.g. [Thinkcpp].

Some benefits of choosing the C++ programming language that help to fulfil the requirements set are:

- integration – most of the system and its libraries are already implemented in C++,
- low impact – thanks to the efficiency this programming language offers, the operations performed by the marauder detection component can use little time and resources,
- correctness – the C++ compilers can do complex static checking.

Python

Considering the requirement of easy adaptation to the hardware change, part of the implementation was done in Python. This is a weakly-typed scripting language that has been built around simple syntax, with many good libraries available (see [Python]). Python has gained a lot of recognition and an increasing number of applications in many fields. It is considered that it reduces the effort required in certain types of projects.

The part implemented in Python handles communication with proprietary software provided by the DAC vendor. Basically, it constitutes an adapter layer between possibly incompatible DAC controlling software and other parts of the marauder detection component's implementation. As each operation invoked here involves at least creating a new process, whose operations can be quite costly as well, an overhead caused by using Python is relatively small.

In exchange, simpler implementation is gained, also it will be easier to incorporate future changes required to support new hardware.

5.2. Libraries used

Proprietary libraries used in the system

HYDRAstor libraries provide many utilities that help to integrate with the rest of the system and the test framework. They include things like: logging, exceptions, means to report error conditions, threading support, handling asynchronous I/O operations, and many others. What is important, this is all done in an uniform way for the HYDRAstor project.

Boost

Among elements provided by the Boost library that the marauder detection component uses are things like: functors, smart pointers, and regular expression support (see [Boost]).

Particularly, a library called Spirit was chosen to implement the configuration file parsing. It allows to specify a grammar purely in C++ code in a way analogical to the BNF notation. This is implemented using overloading of many operators and meta-programming techniques (see [Boost.Spirit]). The use of the Spirit library allowed to easily handle quite flexible format of the configuration file and to simplify the implementation.

5.3. Important aspects of the code design

Testability

It is important to write the code in a way that would enable creating class tests. This involves certain steps during the implementation, some of them are well known object-oriented programming practices:

- use well defined interface for classes – the better the interface the easier it is to test,
- use abstract interfaces – this allows to replace a class with a stub, in C++ this comes down to using abstract base classes,
- use functors – they can conveniently replace an interface with one method,
- add simple constructors for testing purposes only – for some classes an empty stub would not be enough to implement certain tests, such constructor greatly simplifies creating them.

Object lifetime

With great features of the C++ programming language come some nuisances, the most important one is the requirement of managing and being aware of the objects' lifetime. However, there are some steps that simplify this that are used by the implementation:

- use smart pointers – then lifetime management becomes automatic;
- use functors – they manage their callback object lifetime and ensure proper destruction;

- use pointer containers – the Boost library provides a set of containers designed to be the owners of the pointers held. This assumption allows them to release the managed memory upon their destruction. Moreover, they provide better performance than standard containers of smart pointers. What is also important, the use of pointer containers make the code a bit clearer.

5.4. Scenario of actions

In the next chapter modules used to implement the marauder detection component will be presented. A scenario with a failed disk device being reported will be described here to provide an overview of the elements' interactions.

The initial situation would be the one where the HYDRAstor logical server is running, and no errors about possible marauders have been spotted before.

- The first active part of the marauder detection component is the implementation of the on-line monitoring as it is embeded in the server itself. It reads the kernel I/O statistics and also uses the *DACTool* to monitor the state of the hardware RAID devices.

- When the on-line monitoring spots a possibility of a marauder disk, it uses a *DisksProblemsData* class instance to persistently store a warning flag.

- When the components of the HYDRAstor system related to the user interface issue a request to obtain that flag's value, a *DiskCheck* tool is invoked to return the required data.

- After receiving the notification about a possible marauder disk present, user can choose to run the off-line tests at the appropriate time. In order to do this, the required HYDRAstor logical nodes should be stopped, then a request to the *DiskCheck* tool to invoke off-line tests should be issued, and finally nodes should be restarted.

- Then the *DiskCheck* tool will use *DiskTester* tool to run the tests and *DACTool* to monitor the environment. It would store the results using *DisksProblemsData* module. Also, the flag set by the on-line monitoring will be cleared.

- Upon receiving a request to read the results stored in the previous step, the *DiskCheck* tool will be called again to return the required results.

- Finally, the user can read the results and may take further required actions.

5.5. Getting information about present disks

There is some information about the hard disks that needs to be obtained in order to implement the solution presented in chapter four. The method for getting this data is described in this section.

Starting from the top, the file system path to the directory where user data is stored can be obtained using certain HYDRAstor libraries. This allows to get a list of all the devices we would want to monitor together with their identification numbers in the operating system.

What we receive here is actually a partition of the operating system's disk device. However, on-line monitoring requires checking the statistics of the entire disk, also it is better to use the name of the whole device in the communicates for the user. This introduces a need of a

translation layer that changes the partition name to the proper device. It is currently added as an extension to one of the system's libraries.

The next step is to get the model that the DAC's logical device uses inside the operating system. It will serve as a key in the translation between operating system's devices and the DAC layer. Here an *udevinfo* Linux command line utility is used, please refer to [udevinfo] for more details about it.

DAC configuration software is to do the rest of the work. With its help, a set of the logical DAC devices is obtained, containing the following data for each entry:

1. Logical device model.
2. Flag whether this is a hardware RAID.
3. List of physical devices that are hosting this logical device (there can be more than one for hardware RAIDs). Each element has the physical disk model specification.

Once all this data is obtained, it may turn out that more than one operating system's device is using the same logical device. Nevertheless, all the required information is already collected, meaning that for each disk that should be tested the following elements are known:

1. the file system path to the directory mounted on it,
2. the number and name of the main device in the operating system,
3. the physical disk model,
4. a list of other operating system's devices sharing the same physical disk, that is needed by the on-line monitoring.

List of hardware RAIDs mapped to the operating system's devices is also obtained during those steps.

Chapter 6

Design

Main modules and classes that constitute the marauder detection component will be presented here. For a more general description of the methods used see chapters four and five.

6.1. Overall module dependency

Main modules and their dependencies are presented in figure 6.1. *DiskCheck* tool is the part that implements off-line testing. On-line monitoring is part of the HYDRAstor logical server. All the parts depicted there are described in more detail in this chapter. With the exception of *DACTool* presented in section 6.2, all elements are implemented in the C++ programming language.

6.2. *DACTool*

As it was previously described, this is the part encapsulating proprietary shell interface to the DAC. It is implemented in Python. *DACTool* is used as a script taking command line arguments as an input, and printing results in some fixed format to the output.

It accepts the following arguments:

1. Controller type – allows to support many vendor's hardware.
2. Controller identifier – tells on which DAC instance to operate.
3. Operation to be executed.
4. Optionally log file location – used mainly for debugging purposes at implementation phase, also for one of the tests to check if required actions have been invoked.
5. Optionally operation specific arguments.

The supported operations are:

1. Check hardware RAID status – reports if some background RAID tasks are presently active.
2. Change DAC cache state – Used for off-line testing and clean-up after it to set up and restore test environment state. Cache should be turned off or on respectively.
3. Get disks models – it is responsible for getting all the disks information required from the DAC.

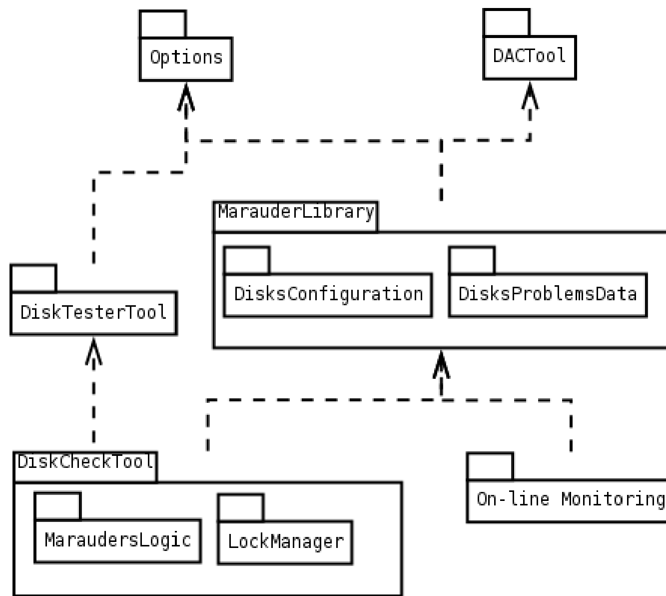

Figure 6.1: Dependencies of the main modules

The last command is executed in 3 phases. The first one is invoking *udevinfo* for each operating system's disk to map them to the DAC's logical devices; this is done as part of the generic implementation. Then current vendor controller utility is called by the appropriate *GenericController* subclass returning a map with data about logical devices and a map with data about physical devices. Finally the generic layer merges all the 3 results obtained so far, and it prints the data in fixed format to the output.

The internal design is presented in figure 6.2 that shows a class diagram for *DACTool*.

6.3. *Options* module

To achieve easier maintenance by simpler configuration procedure, a separate module for marauder detection component's specific configuration has been proposed. It has a simple, but quite flexible, format that consists of:

- a header with version number,
- a group of global options,
- a list of named sections, each one containing a list of named tests.

Each test can have a list of options, each consisting of a pair of a name and value. Also, it can contain a section of the result values. It was decided that options must have a fixed type, and the query of option value should return a specified type. This allows to encapsulate type checking in the options module . To be able to actually use the value its type should be equal to the required one so returning a variant type here would not be really useful, especially that the options contain only simple values.

Each non-default test should have an option specifying what kind of test it is. This is used to create a proper test object, described later.

The value lookup order is designed to support off-line testing semantics when the names of the sections are used as the disk's model. When the query for an option with a given name and type arrives, the following are searched:

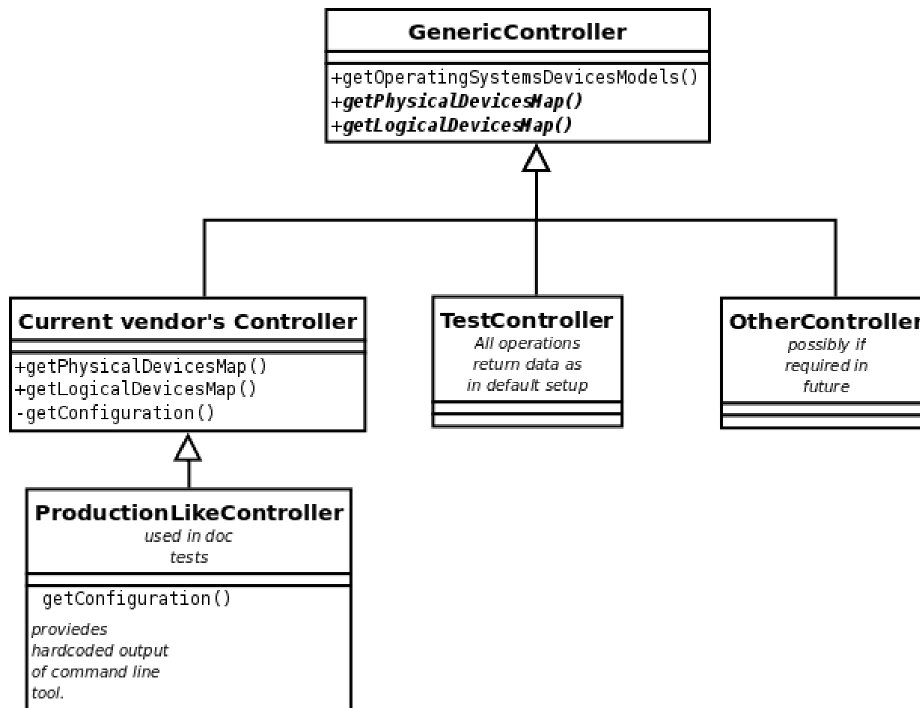

Figure 6.2: Class diagram of *DACTool*

1. current disk model section, under a current test name,
2. current disk model section, under a test named *default*,
3. default section, under a current test name,
4. default section, under a *default* test.

This allows to specify a minimal set of requirements about the tests' results in the default section, disks with unsupported models will be checked using this information.

The layout of option values for on-line monitoring is determined by the assumption that different disks may have different models. There is a special section for the configuration of this part that lists all the tests' names that will be used. For a given disk model the on-line monitoring code will try to lookup options' values in the right section. The test name comes from the list mentioned.

Apart from the representation of a section, a test, and an option container, *Options* module has two important classes *DiskTestOptions* and *DiskTestOptionsReader*.

DiskTestOptions

This is the main class for storing and manipulating options exported to the other modules. It has the following capabilities:

- selecting current section and test,
- reading and setting options,
- serialization to text representation, also method *printForResults* that cuts the results to parts relevant to currently selected disk model, used to reduce the input and output data size of off-line tests,

- interface to enumerate all the tests in the current section.

DiskTestOptionsReader

Acts like a visitor for a grammar generated with Boost Spirit, handling creation of the *DiskTestOptions* objects.

6.4. *DiskTester* tool

It is the part that is responsible for performing the actual off-line tests described in the chapter four. Similarly to the *DACTool*, it is designed to be a command line tool. Because it has to be able to read the disk's device file, it is run with superuser privileges. Therefore, many responsibilities are taken out of this tool and moved to the upper layer. For example to avoid problems with managing root's log files, all warning messages are printed to the standard error stream to be logged later.

To ensure termination after start, this tool takes a lock on a file specified in options. By leaving the file descriptor opened throughout the program execution, the lock is automatically released when the program is shuting down. Application should not close any descriptor leading to this file because it would release the lock. This means that the file used for writing should not be used for locking as it is closed after one test.

To avoid problems with slow disks in this tool's execution the options are parsed from the standard input, the results are printed to the standard output, also using the *Options* module. As the tests use the *O_DIRECT* mode, they require proper alignment of disks requests to the block size. That value is configurable in the options described before.

DiskTester tool performs the following operations:

1. Parse the options from the input stream, their global part should provide important setup, like current disk model, read and write files' path.
2. For each test described in the options:
   (a) check the test type, and if it is an off-line one,
   (b) create proper test object,
   (c) execute the test,
   (d) store results.
3. Print all the results to the standard output.

The use of the *DiskTestOptions* class in the process described is presented in figure 6.3.

Classes

The most important one is the *BaseTest* class. It is responsible for:

- Submitting requests to the disk.
- Calculating results and storing them properly. They are in the form of a number of operations completed in a given time, plus some additional data.
- Measuring duration of the test, and stopping the test when its time elapses. Execution time may be longer than the granted time window due to a single unfinished request by a marauder disk.

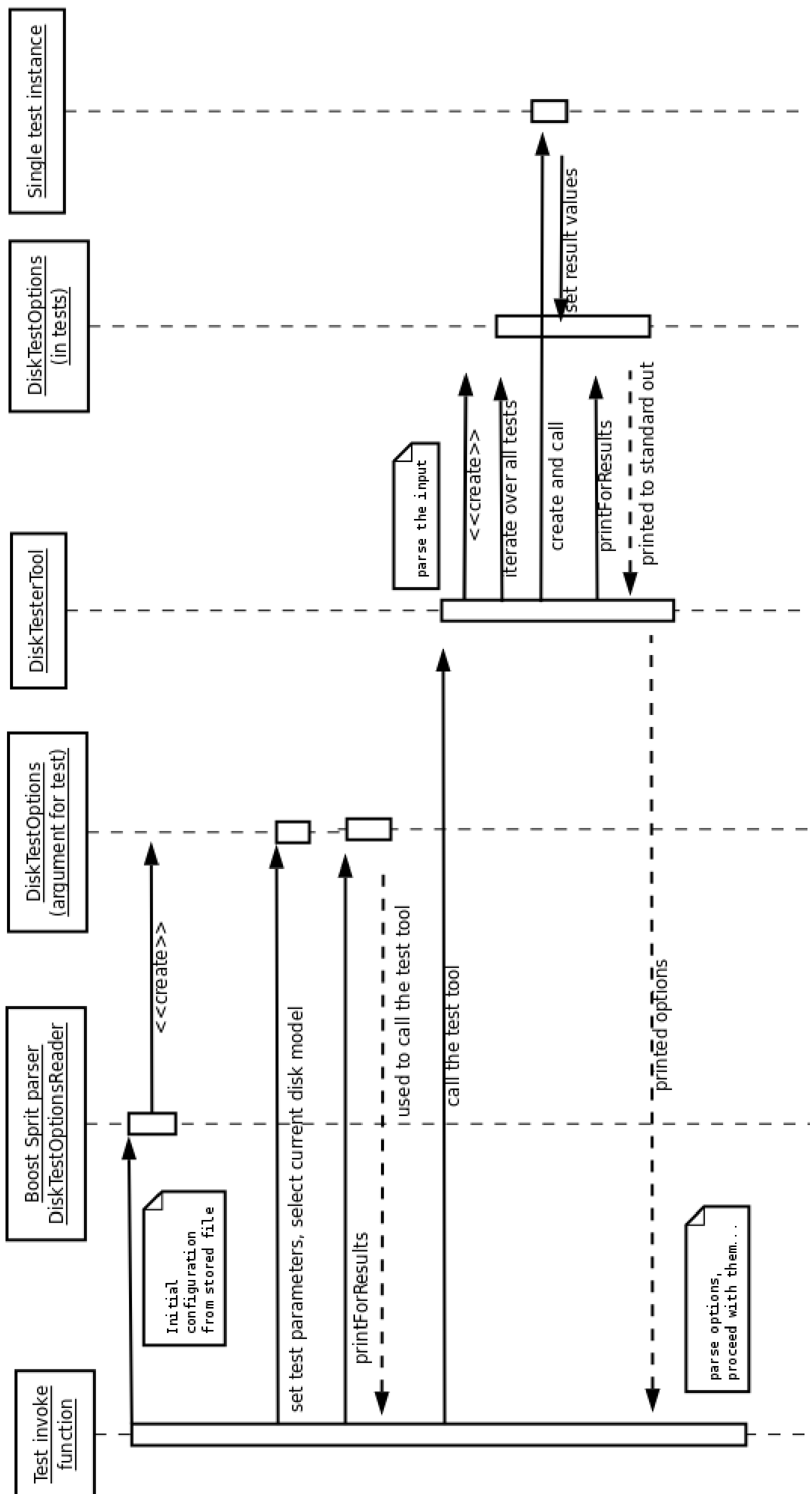

Figure 6.3: Use of *DiskTestOptions* in invoking an off-line test

It is a base class for all the test classes, that are named: *RandomReadTest*, *SequentialReadTest*, *RandomWriteTest*, *SequentialWriteTest*. Their main concern is opening proper file, and deciding what request to schedule.

The sequential write test simply appends data to a given file. Random write one reuses the data generated by the sequential test to do short writes at random location in file. Random read test does single block reads from randomly chosen locations at the disk.

The most interesting one is the sequential read. It uses a right subclass of a class called *OffsetProvider* to choose the read location. The provider currently used divides the disk into a configurable number of sections, and visits them in a round-robin manner. For each section, a random offset that is properly aligned is selected.

Error handling

Due to the requirement of not checking data correctness, errors in submitting the disk requests are generally skipped, with the following two exceptions:

- A few first errors are printed to the standard error output, to indicate in logs that something went wrong. However, too many messages would decrease performance of the test so later ones are not logged.
- If the error is caused by the lack of permissions, it is not ignored because such action could lead to false negatives. One such error means generally that all the requests will fail similarly, giving very good test result. In such case tool exits with an error.

In case of some internal failure the tool crashes with non-zero exit code and prints proper message to the standard error stream.

6.5. Tool wrappers

There are C++ wrapper classes for both *DiskTester* and *DACTool*. They use system libraries to create a subprocess, get its exit code, and read the output. Wrappers do low level handling of some error conditions, but their main concern is parsing the output. They use string operations and regular expressions from the Boost library to do this work.

Quite important is the logging of errors, that is not done in these tools.

6.6. *DisksProblemsData*

This part implements storing results of all the tests performed by the marauder detection component. The information maintained is describing one server instance.

The figure 6.4 presents the hierarchy of elements that can be stored. Classes used here are mapped to the types exported to the user, as described in section 3.3.1. As they are not directly accessible, some additional information can be hidden inside them. An example of such data can be detailed test results stored to allow implementing some of the extensions planned in future. Each non-abstract class provides the following functionality:

- Serialization in binary format.
- Pretty printing as text, the selected format is the same as used for defining Python objects. The output can be simply evaluated as a Python expression giving ready to use objects, this turned out to be useful for integration with the testing framework.

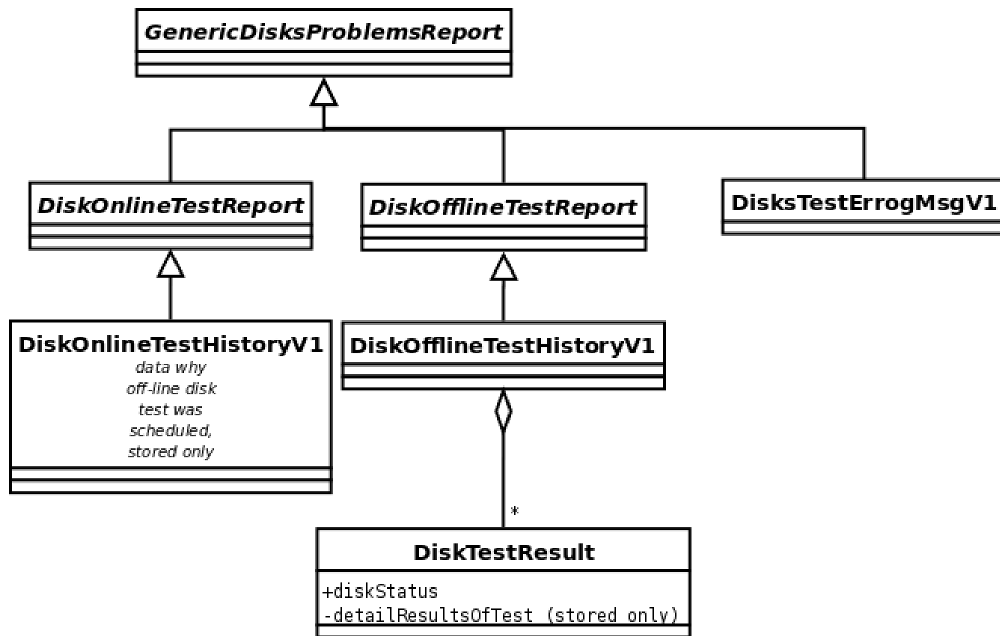

Figure 6.4: Storable entities in *DisksProblemsData*

- The *accept* method for the visitor pattern, used for implementing the user interface to the classes.
- They also have time-stamp field to track when the tests took place.

The abstract base subclasses are provided to allow adding new versions of classes in future in a way compatible with the binary serialization framework. This explains the *V1* suffix in name.

The most important part for the off-line tests results is the disk status that can be one of:

- CORRECT – disk passed the test.
- CONFIRMED_MARAUDER – disk is reported as faulty.
- UNCONFIRMED_MARAUDER – disk failed the test, but errors in the environment have been detected.
- TOO_MANY_MARAUDERS – too many disks failed the tests; this can indicate a problem with the DAC.
- UNKNOWN – there was a fatal error in the testing procedure.

DiskTestHistoryList

This class serves as a container for data objects described before. The most important field is the flag indicating whether the off-line disk test is suggested. This flag is automatically set by the on-line monitoring and turned off after completion of the off-line test for a single HYDRAstor server instance.

Data objects are organized in a queue with new messages discarding the oldest. This gives the ability to present a snapshot of history of results to the user that may be helpful in

diagnosing the possible problem. To reduce the amount of data returned by the tests' history a filter operation is added. It is responsible for removing off-line test results for disks that are no longer used by the system, for example after a user would replace a failed device with a new one.

DisksProblemsData

This class serves as an interface to this part, it implements storing and retrieving operations. Reads and writes are using a proprietary system's library; they are directed to a file stored on a RAID partition. The main benefit of such choice is getting a good implementation of locking.

Because accessing the file may block current thread for too long, the constructor takes as an argument an object of an initializer class, that can be returned from another thread. Such design allows keeping the object under the management of the main thread, and delegating longer task to other threads that can block safely. This simplifies access control management in the multi-threading implementation of the marauder detection component.

To ensure proper usage there is an assertion in destructor that checks if there is no dirty data to be stored.

6.7. *DisksConfiguration*

This part of the marauder detection component collects and provides all the required data describing disk configuration in the system. This contains such elements as: file system path, model key, device name and number, information about hardware RAID devices, also the path to save the tests' results. A main class, named *DisksConfiguration*, is responsible for:

- mapping a disk partition to its base operating system's device,
- getting the list of all disks used by the HYDRAstor server,
- providing information which operating system's disks share the same DAC logical device,
- distinguishing between hardware RAIDs and normal logical devices.

It features two-staged initialization. In the first step HYDRAstor system's configuration and libraries are queried, in the second one the data obtained from *DACTool* is passed to the object, possibly from another thread.

6.8. *DiskCheck* tool

This is the main implementation place of the off-line testing. It has 3 work modes corresponding to its main responsibilities:

1. Invoking an off-line test in a proper environment, verifying the test is not disturbed, and calculating results.
2. Performing clean-up after failure of this tool.
3. Reading test results.

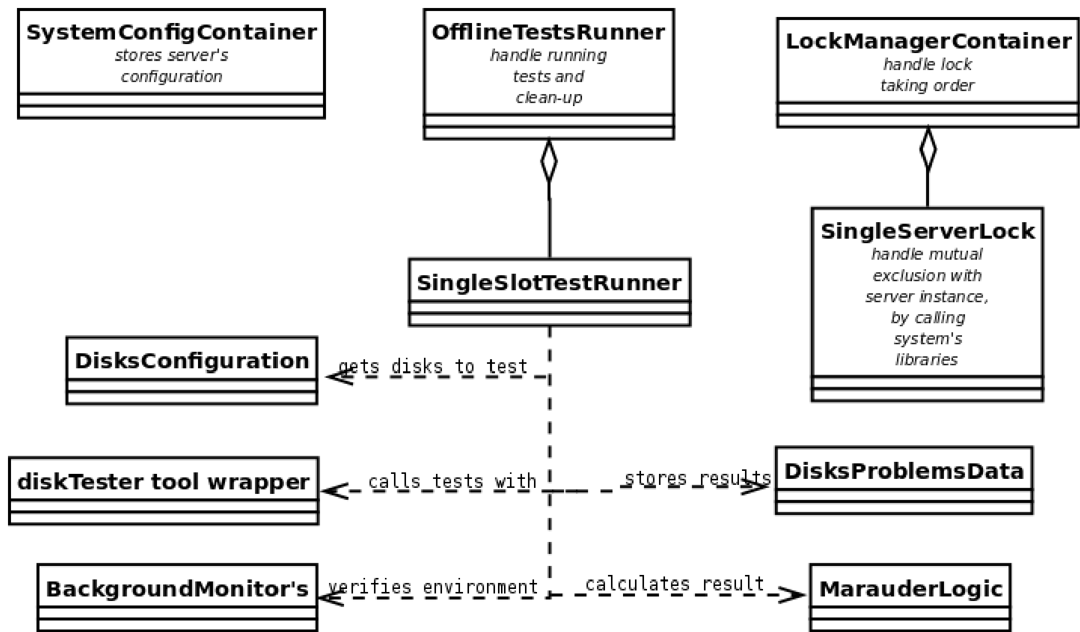

Figure 6.5: *DiskCheck* tool internals

6.8.1. Invoking the tool

The tool is triggered by an external request that is firstly processed by the upper layer. The following steps ensure a safe invocation of the tool in the run test mode:

1. Get the list of the logical node instances running on one physical node.
2. Get their configuration, and determine whether they share the same DAC.
3. Combine obtained data with a list of the logical nodes that the user wants to test.
4. Call the *DiskCheck* tool for each DAC separately.

Again, the tool itself works as a command line utility. It uses Boost to parse program options. They include:

- mode of operation,
- list of servers to check,
- list of servers that share same DAC, but are not going to be checked,
- log directory.

6.8.2. Important classes

The parts constituting the *DiskCheck* tool are presented in figure 6.5, they will be shortly described in this section.

Managing locking

Because the off-line tests can affect and are affected by possibly more than one logical node instance, it is necessary to integrate *DiskCheck* tool with HYDRAstor logical server's locking mechanism. A *LockManagerContainer* has been created for this purpose. It allows for taking and releasing the locks of the required nodes in a safe manner that guarantees:

- Order of locking is fixed for a given server list – this is required to avoid deadlock.
- Order of unlocking is the reversal of the lock order.
- If failure happens during locking, all correctly locked servers will be unlocked.

MarauderLogic

This class is responsible for interpreting the off-line test results. It receives output of the test tool, status of hardware RAIDs' monitoring, and information about background RAID tasks activity. It compares the outcome test data with configuration, sets the proper error status in case of disturbance detection, adds results and error notifications to the off-line test data that will be stored. *MarauderLogic* object also decides whether to set the status of too many marauders detected, that is why it has access to the tests results for all disks for the given server instance.

OfflineDiskTestRunner

Encapsulates the concept of running all the tests for all the logical nodes. This class mostly delegates all the work to the present *SingleSlotTestRunners*, described in the next subsection. It also handles reading disk configuration with a *DACTool*.

SingleSlotTestRunner

Is responsible for testing and cleaning-up afterwards in scope of a single instance of a HYDRAstor logical server. It manages the tests results, both the logic to get them and the storing mechanism.

It also runs RAID disturbance and background operations checks in separate threads to validate the test environment. The design of this part is presented in figure 6.6.

The listeners provide a function call operator with proper argument type, and they use specific logic to calculate the result value. For example, RAID background tasks status listener will ignore idle status, and will store active status if any is encountered.

To implement environment monitoring separate threads are required as the main thread is waiting for the test tool to complete. *BackgroundMonitor* encapsulates multi-threading aspects; a C++ template allows to achieve better code reuse in its implementation. It works as follows:

1. Stores an listener object in the constructor.
2. A new thread is started during the *start* method of the *BackgroundMonitor* instance.
3. Thread periodically runs proper data providing functor, it passes the result to the listener.
4. Thread is safely stopped in the destructor – this allows for automatic clean-up when an exception occurs.

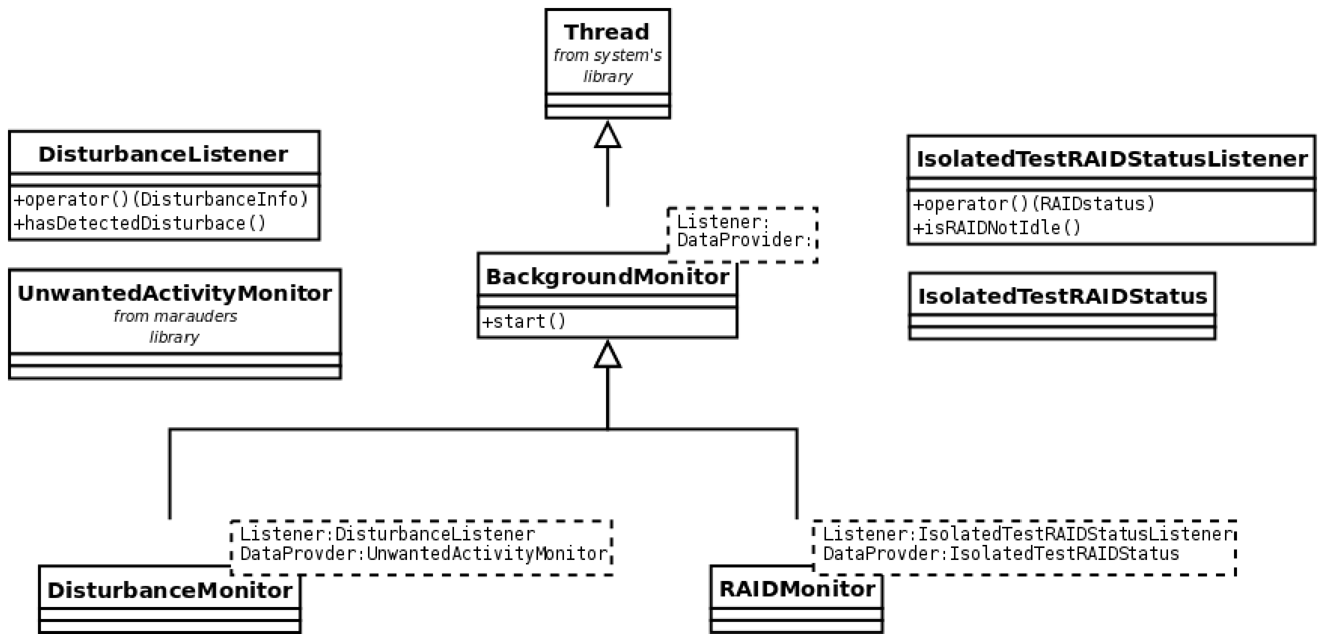

Figure 6.6: Off-line testing environment monitoring design

5. Finally, listener object can be used to get the required results.

The results from listeners are interpreted by the *MarauderLogic* object described before.

6.8.3. Clean-up mode

The clean-up include 3 steps:

1. Terminating any possibly running *DiskTester* tool. This is done by checking the lock file and sending a kill signal to the owner of the lock if any is detected.

2. Removing tests' write files.

3. Enabling the DAC cache.

Normally the clean-up is performed after the test's successful execution. However, if some fatal error occurs during the tool operation, system's locks on logical servers are not released cleanly. Then during next operation the invalid state will be detected and the *DiskCheck* tool will be run in clean-up only mode to recover the previous state.

6.8.4. Read results mode

As previously described in chapter three, tests results should be always accessible. Therefore, this mode does not require any environment checks or locking.

One more thing left to be decided is the output format. As this is a command line tool, exporting binary data may cause some problems, for example with logging. *DisksProblems-Data* provides a text format for its classes, however, it is not designed to be stable, and it may not handle future version changes correctly, but binary format overcomes these limitations. As a solution a serialization of the binary data in the base64 format was chosen. This allowed reusing existing implementation and keeping the text format simple.

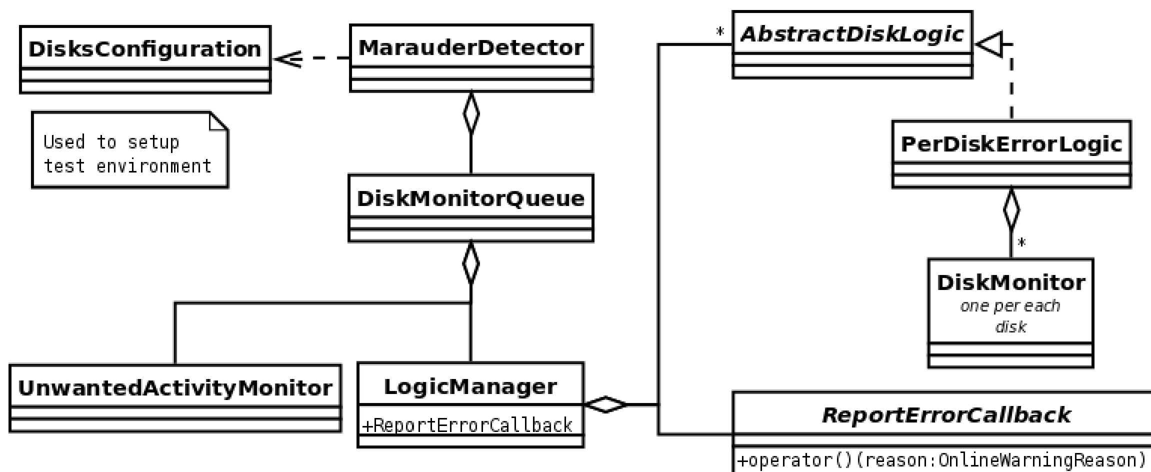

Figure 6.7: On-line monitoring design

6.9. On-line monitoring

This is the part that is implemented inside the HYDRAstor server because it is required to operate together with the logical node. Some issues appear because of that; thread management is more restricted, especially the main thread of the on-line monitoring component should not be blocked for too long. Also, clean shut-down must be ensured. The design of this part is presented in figure 6.7.

MarauderDetector

This is a main class in on-line monitoring that encapsulates all the execution. It also takes care of most of the requirements mentioned before.

During its operation, a *MarauderDetector* object periodically calls two actions:

- getting kernel's I/O statistics,
- checking current RAID background tasks status.

These results are forwarded to the *LogicManager* object created. To be accepted as part of the server, it uses proprietary library to do this work – it internally stores and manages tasks to be executed by a timer class.

I/O statistics are tagged with measurement time-stamps as the objects in the processing chain may introduce delays.

Apart from tasks executed by the timer there are 3 kinds of operations that can block the on-line monitoring main thread, for each one of them a special callback class is implemented:

- Reading tests' result file – the callback returns an initializer object for a *DisksProblemsData* instance.
- Storing a suggestion to run off-line tests.
- Invoking *DACTool* to check RAID background tasks status and to get disk model information.

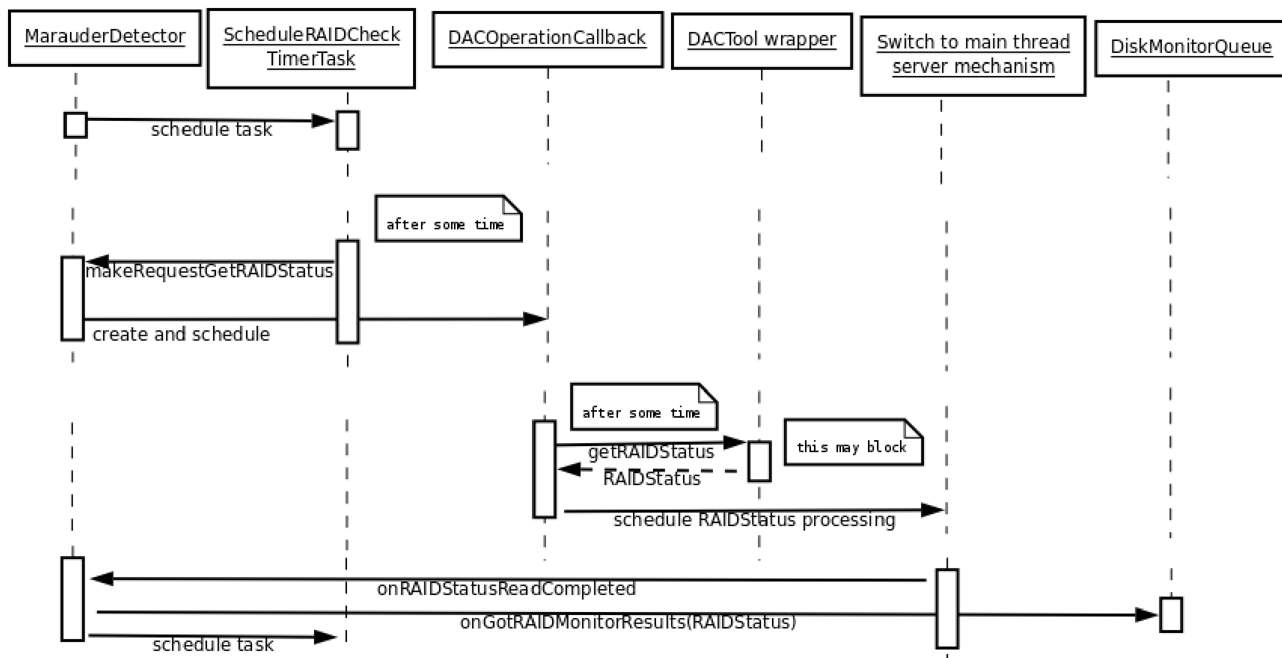

Figure 6.8: *MarauderDetector* callbacks handling for obtaining the background RAID tasks status

They are executed in a separate single thread by a special class from the system's library. During clean finalization of the server all the timer tasks and callback operations should be canceled and finished.

The use of the timer tasks and callbacks is presented in figure 6.8.

The initialization of a *MarauderDetector* object is divided into the following phases:

1. Object constructor sets up the member objects.

2. Delay happens to let the HYDRAstor server start a little bit faster than with initially started on-line monitoring.

3. Reading tests' results from the file where they are stored.

4. Obtaining disk configuration.

The initialization is designed to handle the system closing event, even if it is not completely finished.

DiskMonitorQueue

It is responsible for filtering I/O statistics samples that are affected by the environment problems. Information about current RAID background tasks status and kernel I/O statistics are delivered by the *MarauderDetector* object.

It delays I/O statistics samples until next RAID activity report is obtained. If there has been a background task detected, the measurements are dropped.

UnwantedActivityMonitor is invoked on the I/O statistics obtained in order to remove measurements that are done during heavy hardware RAID's activity.

LogicManager

This class acts as a container for all the possible on-line tests. It passes the I/O statistics to them and collects the results. They can most importantly contain a suggestion to run the off-line test.

To reduce the dependencies of this class, *MarauderDetector* object passes a callback for storing results' data that encapsulates calls to the *DisksProblemsData* object.

PerDiskErrorLogic

It handles the current implementation of the on-line testing logic by serving as a container of the *DiskMonitor* objects. It is responsible for summarising I/O statistics from all the required operating system's disk devices, as described in section 4.4.4.

DiskMonitor

This class is responsible for the on-line monitoring implementation in scope of a single disk. It uses a few helper classes:

- *RunningAvgQueue* – implements the part described in section 4.2.4; provides required statistics for the entire window of monitoring; allows invalidating samples.
- *FallingEdgeDetector* – implements the part described in section 4.2.5.
- *ErrorFrequencyChecker* – implements the part described in section 4.2.6.
- *StatisticsBacklog* maintains a backlog of I/O statistics that is used for more elaborate error reporting.

The processing flow after receiving an I/O statistics sample can be described as follows:

1. The measurement is passed to the *FallingEdgeDetector* that returns a list of already validated samples.
2. For each sample in the list:
   (a) sample is inserted to the *RunningAvgQueue* object,
   (b) if the queue is filled:
      i. *DiskMonitor* performs statistics validation for the entire running average queue window as described in section 4.2.2;
      ii. if any warning is generated, it is passed to the *ErrorFrequencyChecker* object, and all samples in the queue all invalidated to avoid generation of many errors for the same sample.

All the errors that are validated by the *ErrorFrequencyChecker* object are finally passed to the *LogicManager* instance.

There are 3 places that can delay error reporting:

- *FallingEdgeDetector* – only for a few samples' duration;
- *RunningAvgQueue* – until a queue is filled;
- *ErrorFrequencyChecker* – for a specified interval.

However, the introduced delay is pretty small when compared to the expected system's up-time.

Chapter 7

Parameters' evaluation

The solution described in the previous chapters requires certain configuration variables. In case of on-line monitoring they include values used as the following thresholds:

- the average request service time below which monitoring warning is reported,
- the utilization of the disk above which the device is considered to be active,
- the amount of load on a single hardware RAID device that does not disable monitoring.

In off-line tests they include:

- the duration of the check,
- sizes of the reads and writes to submit,
- threshold of the number of requests expected to be done in a successful test execution.

In order to choose proper values for them, the performance of the disk devices has to be investigated. This chapter presents some results of the tests done for this purpose.

7.1. On-line part

The most interesting issue here is the behaviour of the average request service time, the value selected for monitoring, during different work loads. The correlation of this I/O statistic with the other ones influenced the design of the on-line monitoring method.

7.1.1. Foundations of the decisions made

The final decision about on-line monitoring configuration was based on the statistics of the disks performance gathered during the functional tests of the system. It was necessary to check them in different physical set-ups to eliminate the possibility of a variation in the behaviour. This data provided an insight into the service time of disk requests under a typical work load of the HYDRAstor system.

To verify the results obtained in such way, the values of the operating system's I/O statistics were also measured during the artificial tests, especially interesting were the observed anomalies. Such procedure helped a lot in understanding the performance characteristics of the disks.

7.1.2. Sample results of the monitoring

Some results obtained during the I/O statistics monitoring are presented in figures 7.1–7.6. In those graphs the x-axis indicates time in seconds. The left y-axis shows both utilization measured in percentage and the average request service time given in milliseconds. The right y-axis describes the number of the requests processed during a sample time duration.

Figures 7.1 and 7.2 show data obtained during artificial tests performed that imposed heavy load on the disk device, what is proved by the high value of the utilization parameter. In the first graph the average request service time is low, what indicates that conditions when the disk is busy should not worsen the average request time measure.

Figure 7.2 shows an unusual usage scenario that the system is very unlikely to generate in practice. When the random seek test starts, the average service time increases significantly, however, these values are still lower than some reasonable threshold, placed at about 30ms. This indicates that the selected measure depends quite strongly on the type of load on the disk, however practical results are expected to be better than the ones seen in the graph.

Figure 7.3 presents results gathered during normal operating system work. Even though the utilization of the disk oscillates, the average request service time is very low. This again indicates that this measure should behave correctly.

Figures 7.4–7.6 present results found during analysing possible anomalies of the I/O statistics' values. The first one contains peaks of the average requests service time value, up to nearly 200ms. However, the utilization of the drive is low what would not happen with a marauder disk. Therefore, this result shows why such bad samples are filtered out in the on-line monitoring.

Figures 7.5 and 7.6 present a single peak of the average request service time. The anomaly occurs at the end of a very busy period for the disk, as indicated by a quick drop of the utilization parameter's value. The results presented in the first graph indicate the need for adding the falling edge detection, described in section 4.2.5, because otherwise a false error could be reported for the first sample with the peak. The second sample containing the anomaly will be invalidated by the low utilization criterion. This idle time filter can handle some of the falling edge scenarios, as shown in figure 7.6. Because of that, the need for adding the falling edge detection is not present in part of the results obtained during the tests done for the parameter evaluation.

7.2. Off-line part

The most interesting parameter here is the test duration. It is important to find the balance between the time required to run the off-line checks and the accuracy of the tests. Also, the predictability of the results needs to be verified.

7.2.1. Variability of the tests' results

The tables 7.7–7.10 present the correlation between the test duration and the variability of the values obtained. For all the test kinds the results show a similar tendency. Naturally, the number of request completed increases together with the time spent doing them, therefore, the standard deviation of this value gets bigger as well. However, the observed ratio of the standard deviation to the average result is practically constant for the presented data. This suggests that the results can differ between executions, but their variability is normally limited so a reasonable lower bound of the results can be chosen to serve as a threshold.

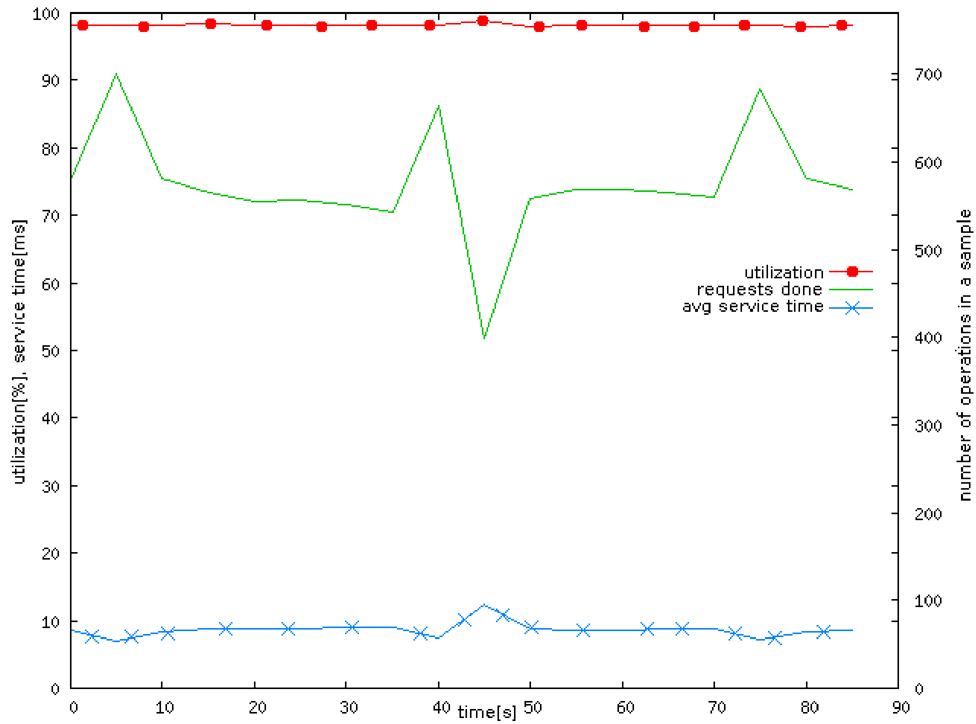

Figure 7.1: I/O statistics from an artificial test with heavy disk load; the average request service time is low and stable.

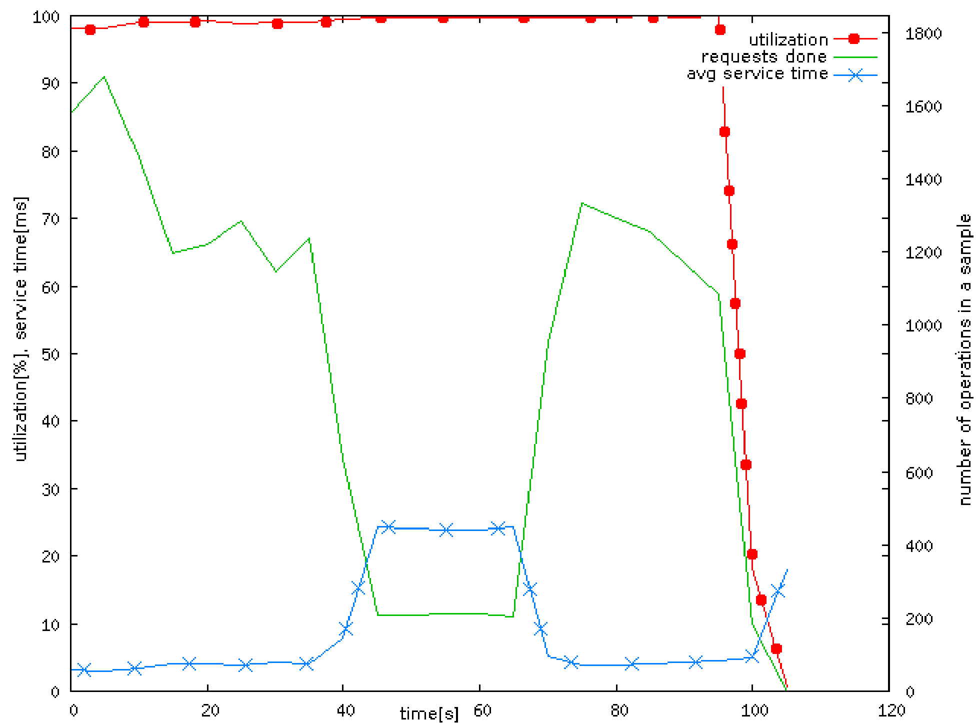

Figure 7.2: I/O statistics from an artificial test; random seeks are started at about 40s. In practice the average service time value should be lower.

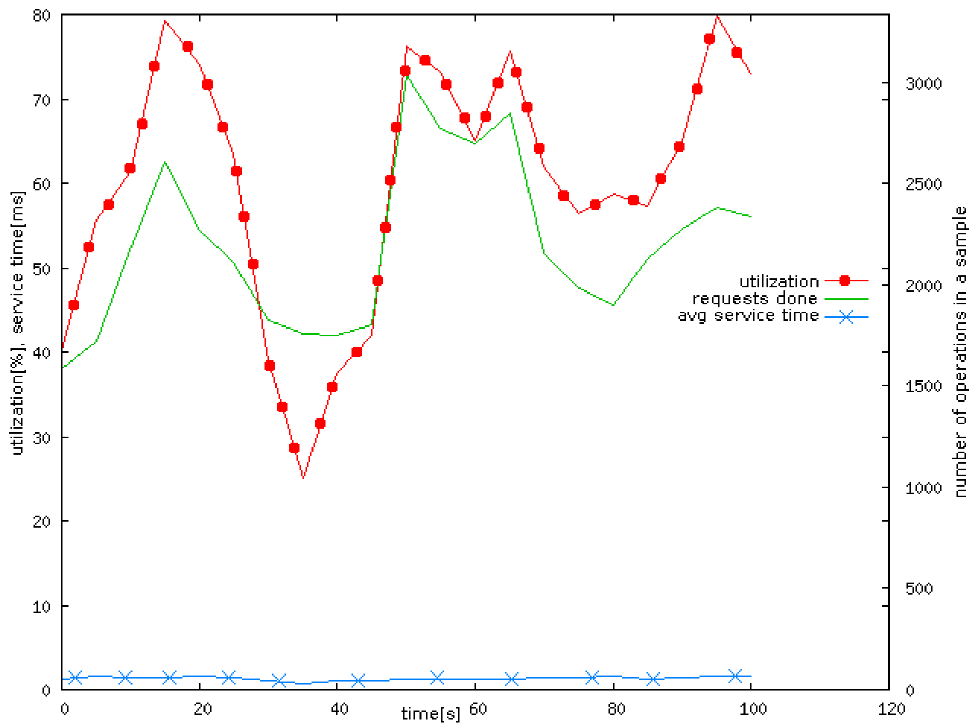

Figure 7.3: I/O statistics gathered during operating system work; very low average service time values

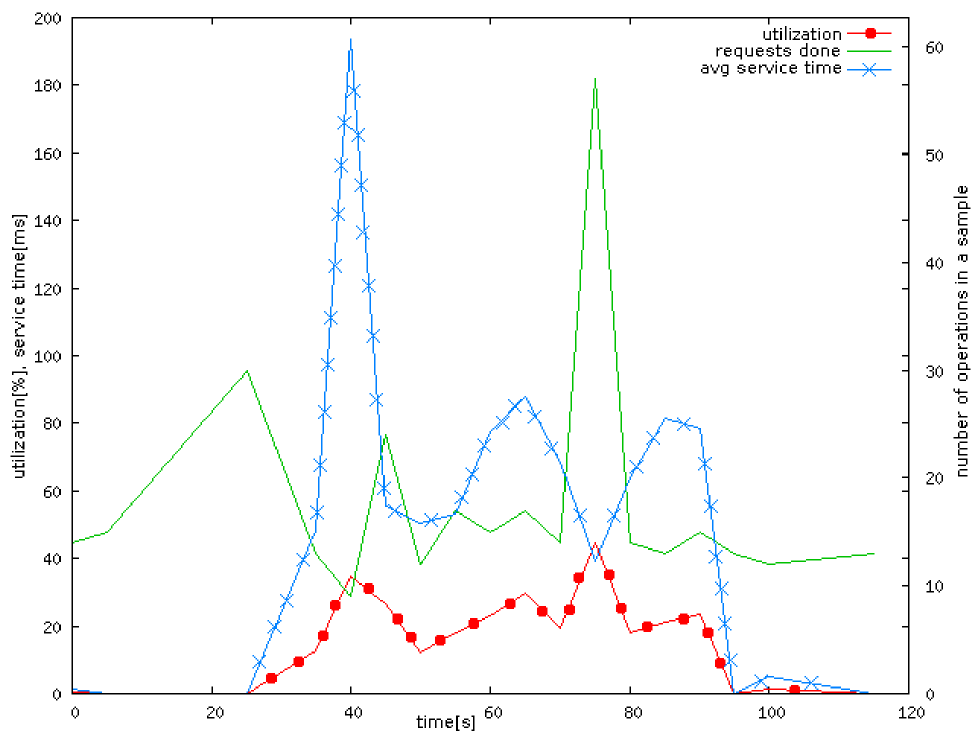

Figure 7.4: I/O statistics gathered during operating system work; low utilization of the disk causes anomalies in the statistics' values.

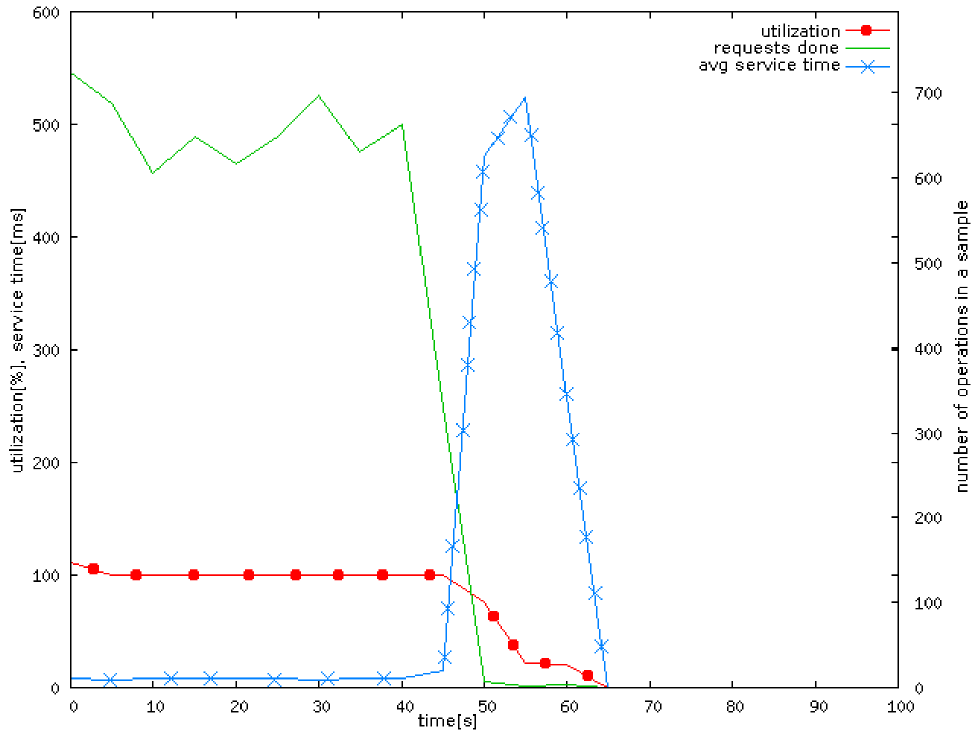

Figure 7.5: I/O statistics showing a quick drop of the disk utilization; the sample taken at 50th second needs to be invalidated by the falling edge detection in on-line monitoring.

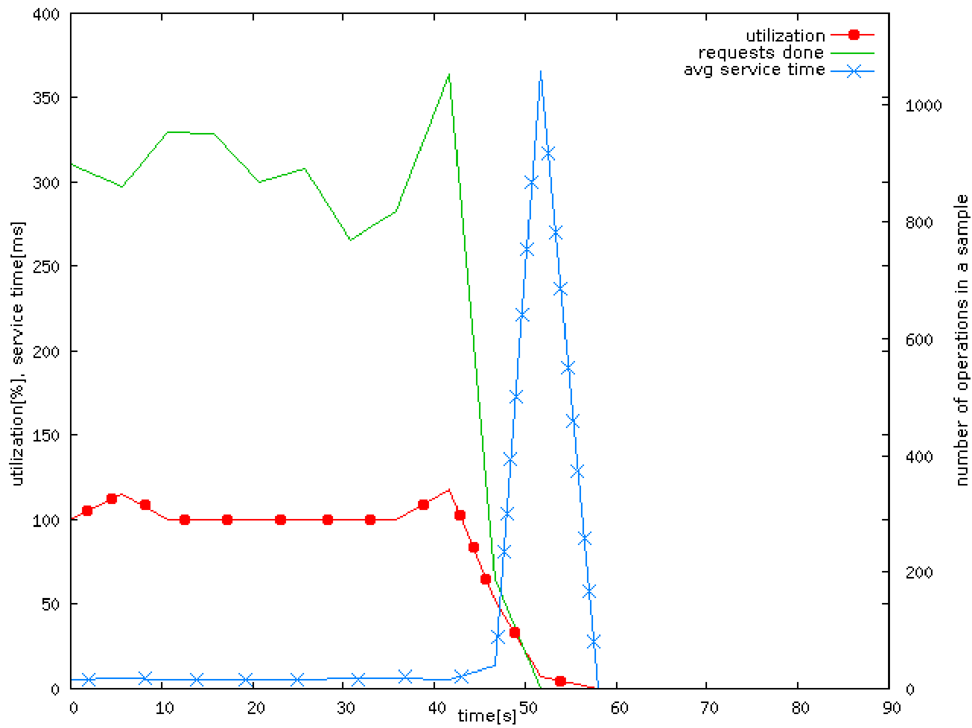

Figure 7.6: I/O statistics showing a quick drop of the disk utilization; the anomaly occurs during the period with lower utilization value.

Test's duration	average result	standard deviation	minimal	maximal	std. dev./average
5s	740,38	66,24	583	814	0,09
7s	1015,63	100,22	740	1118	0,1
8s	1170,35	107,45	895	1290	0,09
10s	1473,44	121,36	1149	1620	0,08
12s	1760,6	150,55	1313	1904	0,09

Figure 7.7: Random Write test done on a partially filled disks. The result presented is the number of operations done.

Test's duration	average result	standard deviation	minimal	maximal	std. dev./average
5s	700,44	46,53	561	753	0,07
7s	981,81	64,79	788	1053	0,07
8s	1129,94	74,34	899	1205	0,07
10s	1410,96	95,34	1098	1493	0,07
12s	1685,31	113,98	1359	1786	0,07

Figure 7.8: Random Read test done on a partially filled disks. The result presented is the number of operations done.

Test's duration	average result	standard deviation	minimal	maximal	std. dev./average
5s	72,92	5,19	58	78	0,07
7s	101,77	6,41	85	108	0,06
8s	117,56	7,6	91	124	0,06
10s	148	9,4	117	157	0,06
12s	177,46	11,26	145	187	0,06

Figure 7.9: Sequential Read test done on a partially filled disks. The result presented is the number of operations done.

Test's duration	average result	standard deviation	minimal	maximal	std. dev./average
5s	48,48	3,56	38	52	0,07
7s	67,85	4,68	56	73	0,07
8s	77,46	5,88	61	83	0,08
10s	97,58	6,74	77	104	0,07
12s	117,02	8,13	92	125	0,07

Figure 7.10: Sequential write test done on a partially filled disks. The result presented is the number of operations done.

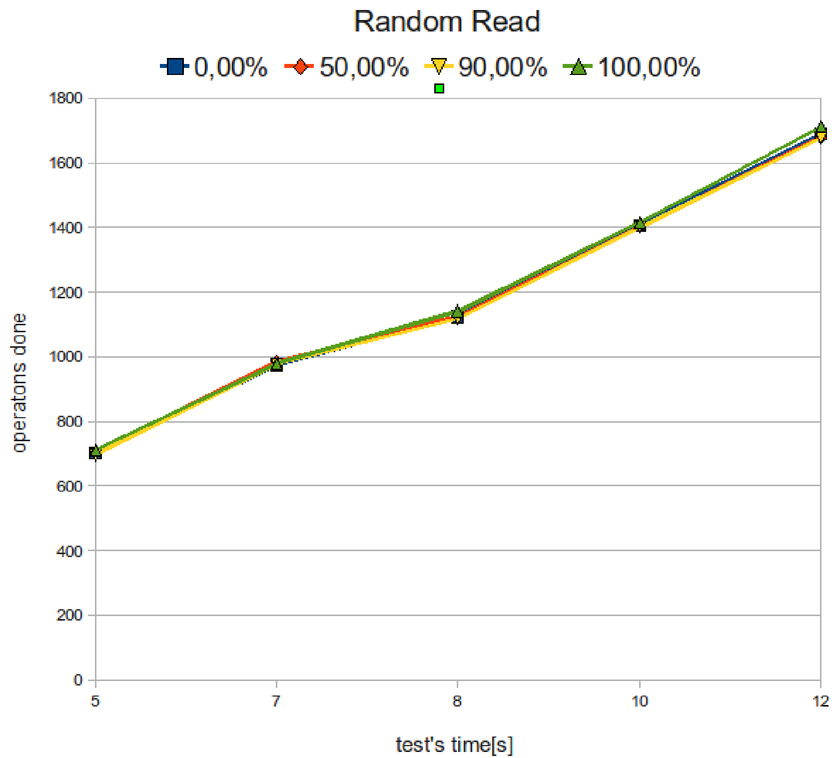

Figure 7.11: Random read test results dependency on the test duration and the amount of disk space occupied before.

7.2.2. Dependency on the space available on disk

The figures 7.11-7.14 present results measured in another experiment. They also show the dependency between the test duration and the number of operations finished, actually the average of this value. As it might have been expected, the resulting function turned out to be very similar to a linear one.

These charts also present the result of the tests done while a different amount of disk space was available. The lines tagged *0%* are executed on nearly empty disks and *100%* on disks being nearly full. A bit surprisingly, the observed impact of the amount of free disk space on the results is not as big as it might have seemed. It is noticeable only for sequential write test that produced significantly higher results on empty disks. However, other results of this test are similar, and these are used to determine the expected threshold.

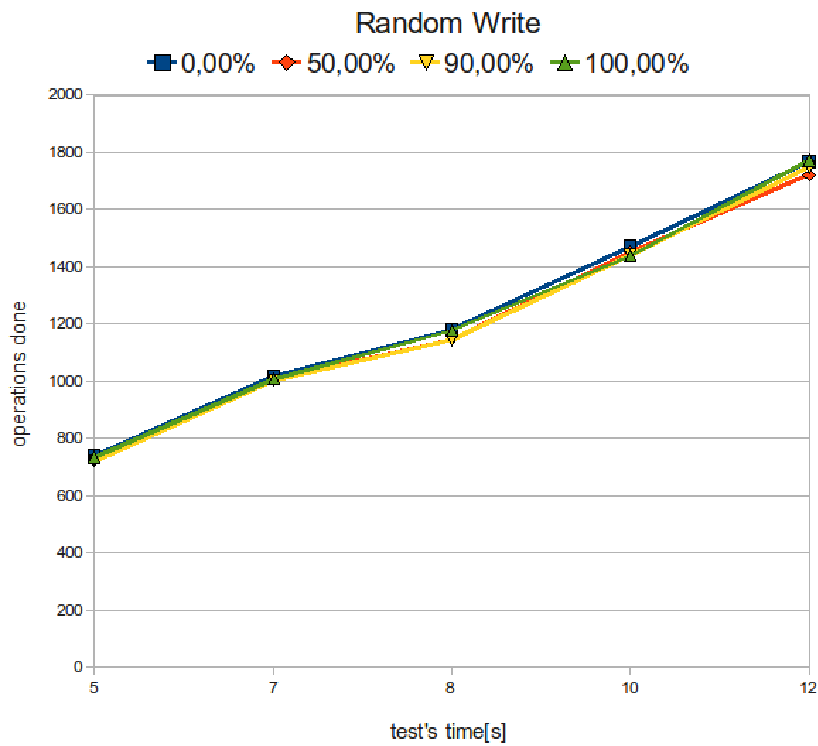

Figure 7.12: Random write test results dependency on the test duration and the amount of disk space occupied before.

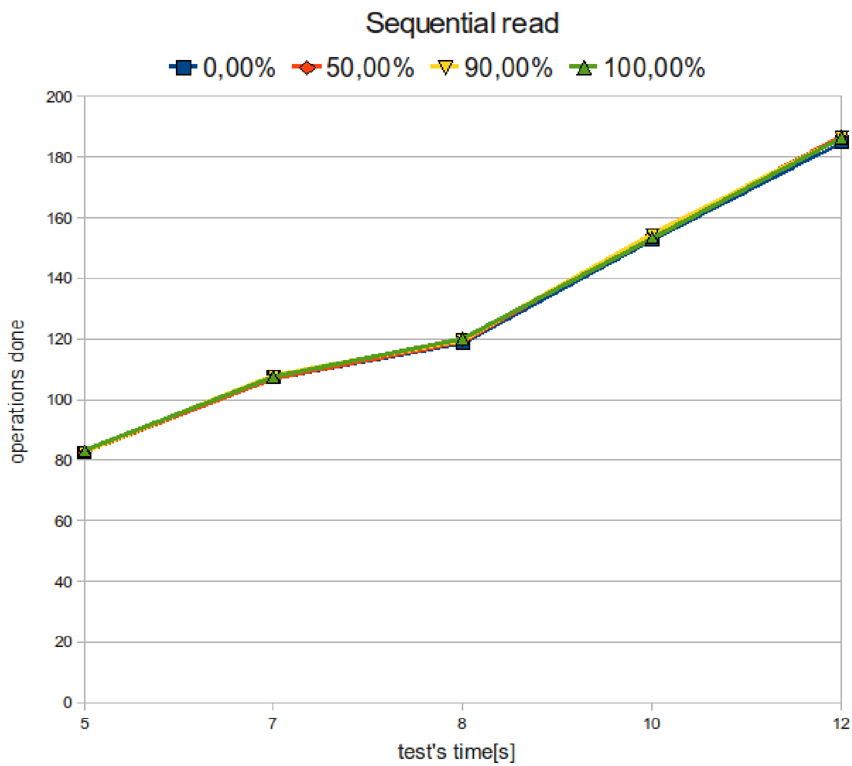

Figure 7.13: Sequential read test results dependency on the test duration and the amount of disk space occupied before.

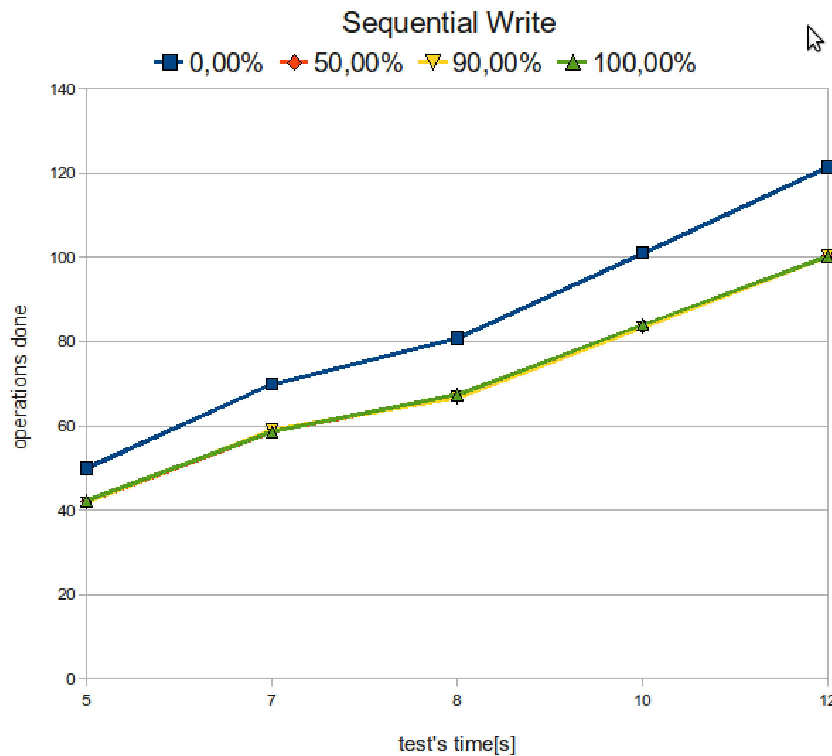

Figure 7.14: Sequential write test results dependency on the test duration and the amount of disk space occupied before; the tests on empty disks are significantly faster.

Summary

The results obtained indicate that the threshold of average request service time in on-line monitoring can be chosen reasonably if the described anomalies are excluded basing on the utilization statistic value.

For each off-line test kind the time granted to it was chosen so that:

- The sum of all the durations is slightly smaller than the time limit granted for testing a single disk.
- The standard deviation of the results is small.
- Time given for random write test is reduced to allow the other three ones to run longer.

The sizes of read or write requests were chosen to simulate operations performed by the HYDRAstor server.

Finally, the threshold of the number of operations successfully done in a test can also be selected according to the results sometimes obtained for the worst case scenario.

Chapter 8

Tests

As the marauder detection component will become a part of the HYDRAstor system, a high level of verification is required for acceptance. All the tests created are using the system's testing framework. This allows to easily execute them automatically and periodically to protect against regression.

8.1. Unit tests

Unit tests were created together with the implementation. They cover nearly all functionality and error handling code. They serve the following purposes:

- Allow finding an error close to its source. This reduces the time required for debugging because such tests typically are short and easy to run.
- Check behaviour in conditions that are hard to create artificially.
- Check that certain errors lead to the desired failure. This would be more difficult to do in the higher layer tests.
- Simplify the implementation process by allowing to execute code that does not provide all the functionality for the higher layers.

8.1.1. Googlemock

An useful tool for writing class tests is a good mock library. One of the well known ones, available for the C++ language, is the Googlemock (see [Googlemock]). It allows to simply specify which methods should be called, with what kind of arguments, and what the results should be. This library provides more functionality as well, but just these basics are enough for typical purposes.

Here is an example of a simple test scenario implemented using the mocking library. Its purpose is to check if *DiskMonitorQueue* correctly handles different background RAID tasks statuses and passes appropriate samples to the *LogicManager* object. The steps of the test are:

1. Create a mock for the *LogicManager* object.
2. Specify that the mock object's method which receives samples would be called certain number of times. Also the calls have to satisfy a constraint stating that the samples must contain a specific key that will correspond in the test scenario to the different

background RAID tasks statuses possible. This way, for the busy status the number of the expected method invocations is zero, and for the idle status, it is equal to the number of samples that should be passed to the *LogicManager* object in the scenario.

3. Use the mock object to instantiate the *DiskMonitorQueue* object that is tested.

4. Execute the test scenario using the queue object created before. For each background RAID tasks status we want to:

   (a) call a method of the queue object to set the current status,

   (b) add some samples to the queue object; the samples' keys should correspond to the status set before.

5. During the mock object destruction if the actual method calls done before differ from the expected scenario, the test will fail with an exception.

The benefit of such approach is the simple specification of the expected result. Also, this step is separated from the implementation of the scenario that is executed in the test. Deletion of the mock object must be ensured because this is when the checks actually happen.

8.1.2. *Doctest*

For the part implemented in Python there is a very simple and robust framework for unit-tests called *doctest* (see [Doctest]). It allows to embed test instructions directly into the comments of the source code. It features an interesting way of checking commands correctness – for each line of code it compares the output of the Python's interpreter with a literal string provided in the test. *Doctest* is quite simple to use, also thanks to some tools, the test can automatically become part of the code documentation.

8.2. Sample integration test scenario

Apart from the unit tests, higher level ones are designed to check the integration between different layers.

One example of such scenario will be presented. The main purpose of its creation was to allow testing user requests handling in a situation similar to a running system. The steps of the test are:

1. Set up a configuration where two HYDRAstor logical servers share a single DAC.

2. Start both servers.

3. Issue a request to read disk test results history, check that it finishes correctly.

4. Issue a request to perform off-line disk testing of one server, check that it fails because the other server is still running.

5. Stop one server.

6. Repeat steps number 3 and 4.

7. Stop the other server.

8. Issue a request to perform off-line disk testing of one server, and wait for its successful completion.

9. Issue a request to read the results, and check that no marauders were detected.

The success of this test depends on correctness of a few elements that participate in passing the request so in case of failure it is not obvious where an error happened. However, the integration between several layers is checked.

There is a similar test scenario that is designed to check what happens when the logical nodes do not share the same DAC. Then it should be possible to invoke a successful off-line test of one server while the other is running.

8.3. Ensuring correct functionality

The focus of all the test kinds described so far was merely to check low level implementation correctness. To make sure that all the modules can interoperate well, and that the proposed algorithms are correct, functional tests are used.

Firstly, there are many scenarios in which numerous system operations are performed in various conditions. With the reasonable assumption that the testing hardware is correct, one can expect that on-line monitoring should not report problems. System functional testing framework can be used to specify such requirement for all its tests.

Secondly, dedicated functional tests of the on-line and off-line parts are planned. The most difficult task here is preforming the tests where simulating a marauder disk is desired. For the off-line monitoring the functional testing framework offers a solution. It provides a library that allows delaying the read and write requests, in fact simulating the behaviour expected from a marauder disk. Using this mechanism several test scenarios are planned.

In on-line part this would not provide a correct solution of the problem. However, there are some possibilities of designing such tests. One idea was to use a virtual machine. Performing some heavy IO load in the host operating system, would slow down the requests processing in the guest one, and the reason of this will be invisible in the system running on the virtual machine.

Another approach, chosen because it allows easier automation of the test execution, is to insert hooks in the I/O statistics reading library to worsen the results at will. There is still an important problem that remains to be solved: it is hard to tell what kind of anomalies would a marauder disk cause, and in effect what should the statistics returned by the test hooks be like. The solution chosen for the tests provides fake I/O statistics that have high utilization and low number of requests done. This is the behaviour that defines a marauder disk.

Additionally, a real instance of a marauder disk had been found in the testing infrastructure. It has been tested using the developed off-line tool. The results showed a slowdown in the write test, what turned out to be expected from the analysis of the previous performance of the disk.

Chapter 9

Summary

A combination of the on-line and off-line parts seems to be a reasonable solution to the problem that the marauder detection component tries to solve, especially considering the requirements stated. On the one hand, the weak point of the on-line monitoring is not showing exactly which disk has failed, what is not an issue for the off-line test. On the other hand off-line part is too expensive to be invoked frequently, yet the on-line one can work continuously. As we can see a solution composed of both elements can overcome the problems they have separately, and it still provides most of the benefits that each of them has.

Even though the implementation is quite specific to the HYDRAstor system, as it is highly coupled with its design and libraries, the solution approach described here is general enough to be used in other storage systems, or in any different software that depends a lot on the performance of the underlying disk devices. Such systems could benefit from earlier detection of possible problems that reduces the impact of the upcoming failures. The cost of performing the marauder checking in normal conditions seems low enough compared to the benefits granted.

Tests performed so far prove that the marauder detection component performs its operation correctly. Therefore, this component will become a part of the future HYDRAstor system release, and ultimately will be available to the system administrators. Hopefully, marauder detection will further increase the reliability of the system, and it will simplify diagnosing problems connected with malfunctioning hard disks drives.

9.1. Future work

During the design and the implementation phase there were many ideas about how to possibly extend the marauder detection component. Because of the time constraints, and the need to keep the the task focused on the main issue, they were not realized so far.

We can name here a few ideas:

- Focusing off-line test on the worst performing sections of disk – currently not implemented due to the time requirements set.
- Invoking automatic actions after detecting a marauder disk – they could include stopping the use of the hard drive for storing new data and retrieving any required information from it, before a fatal failure happens.
- Self-tuning of the on-line monitoring – after suggesting an off-line test that detects no errors, the on-line monitoring should be less eager to advise an off-line test again, especially when the same error condition is met.

Although important benefits will already be provided for the HYDRAstor system, the ideas mentioned allow to consider the marauder disks detection to be a base step towards further improvements.

Bibliography

[Bonnie++] Home page of the Bonnie++ disk benchmark, http://www.coker.com.au/bonnie++/

[Boost] The Boost library home page, http://www.boost.org/

[Boost.Spirit] The Boost Spirit library home page, http://boost-spirit.com/home/

[Doctest] Documentation page of the doctest Python module, http://docs.python.org/library/doctest.html

[Ele09] Jon Elerath, *Hard-Disk Drives: The Good, the Bad and the Ugly*, Communications of the ACM, June 2009, 38-45

[FailureTrends] Eduardo Pinheiro, Wolf-Dietrich Weber, and Luiz André Barroso, *Failure Trends in a Large Disk Drive Population*, 5th USENIX Conference on File and Storage Technologies, 2007

[Googlemock] Home page of the Google C++ Mocking Framework, http://code.google.com/p/googlemock/

[HDPerf] Home page of the hdperf disk benchmark, http://hdperf.sourceforge.net/

[HYDRA09] Cezary Dubnicki, Leszek Gryz, Lukasz Heldt, Michal Kaczmarczyk, Wojciech Kilian, Przemyslaw Strzelczak, Jerzy Szczepkowski, Cristian Ungureanu, and Michal Welnicki, *HYDRAstor: a Scalable Secondary Storage*, Proccedings of the 7th USENIX Conference on File and Storage Technologies, 2009, 197-210

[IntraDisk] Ilias Iliadis, Robert Haas, Xiao–Yu Hu, and Evangelos Eleftheriou, *Disk Scrubbing Versus Intra-Disk Redundancy for High-Reliability RAID Storage Systems*, Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems, 2008, 241-252

[IOstats] Linux kernel documentation, included with the distribution, http://www.kernel.org/, *file Documentation/iostats.txt*

[LatentErrors] Phillipa Gill, Sotirios Damouras, Bianca Schroeder, *Understanding latent sector errors and how to protect against them*, 8th USENIX Conference on File and Storage Technologies

[ORION] James F. Koopmann, *Measuring Disk I/O – Oracle's ORION Tool*, http://www.jameskoopmann.com/docs/MeasuringDiskIOOraclesORIONTool.htm

[Python] Home page of the Python distribution, http://www.python.org/

[RAIDmonitor] Patent *Method for detecting problematic disk drives and disk channels in a raid memory system based on command processing latency* http://www.freshpatents.com/-dt20090423ptan20090106602.php

[StaggeredScrubbing] Alina Oprea, Ari Juels, *A Clean-Slate Look at Disk Scrubbing*, http://www.rsa.com/rsalabs/staff/bios/aoprea/publications/scrubbing.pdf

[Thinkcpp] Bruce Eckel, *Thinking in C++*, http://www.mindview.net/Books/TICPP/ThinkingInCPP2e.html

[udevinfo] Manual of the *udevinfo* utility, http://linuxmanpages.com/man8/udevinfo.8.php