University of Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

BACHELOR'S THESIS

Study programme/specialisation: Datateknologi bachelor	Spring/ Autumn semester, 2020. Open / ~~Confidential~~
Author: Adam James Becker and Lars Kartevoll	
Programme coordinator: Supervisor(s): Veronica Estrada Galiñanes	
Title of bachelor's thesis: Implementing the Python Version of FaiRDy, a Framework to Understand the Dynamic Behavior of Storage Systems	
Credits: 20	
Keywords: FaiRDy, cellular automata, Python library, web interface, erasure code, Django, simulation, storage security	Number of pages: 80 + supplemental material/other: 29 appendix Stavanger, 25/05/2020 date/year

Implementing the Python Version of FaiRDy, a Framework to Understand the Dynamic Behavior of Storage Systems

FaiRDy-Py

University of Stavanger
Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Bachelor's Thesis
Bachelor of Science in Computer Science

Adam James Becker
Lars Kartevoll

Faculty Supervisor
Veronica Estrada Galiñanes

May 25, 2020

Abstract

The amount of data stored worldwide continues to grow exponentially. Whether this data is stored locally on flash memory, or in a distant data center, measures must be taken to ensure the data's availability and integrity. Erasure Codes are widely used to achieve this goal by encoding a given amount of redundancy into the original data stream. Errors can then be apparent or detected and corrected up to specified thresholds. It is important that engineers choose an appropriate erasure code for their application, both to protect the original data from failures, but also to minimize the overhead from redundancy. For this purpose, we have created a set of Python modules to compare different erasure code strategies and policies based on a simple cellular automata model. We have created a module in Python which simulates the life cycle of systems using either n-way replication, Reed Solomon codes, or pyramid codes. We have also implemented a web module to facilitate the processing, storage and visualization of results from the core Python module.

Contents

Abstract **i**

1 Introduction **1**
1.1 Goal, Motivation, and Context 1
1.2 Contributions and Limitations 1
1.3 Organization of this Document 2

2 Project Description **3**
2.1 Erasure Codes 3
2.1.1 n-way replication 3
2.1.2 Reed-Solomon codes 4
2.1.3 Pyramid codes 4
2.2 Cellular Automata 6
2.3 FaiRDy Cellular Automata model 6
2.3.1 Previous Work 8
2.3.2 Fault-injection Module 8
2.3.3 Storage Array 8
2.3.4 Repair Module 9

3 Technology Overview **12**
3.1 Python Programming Language 12
3.2 NumPy Python Library 12
3.3 Matplotlib Python Library 14
3.4 GitHub Open Hosting and Version Control 14
3.5 Hypertext Markup Language (HTML) 14
3.6 Cascading Style Sheets and Bootstrap 16
3.6.1 Cascading Style Sheets (CSS) 16
3.6.2 Bootstrap CSS Library 16
3.7 JavaScript Language and jQuery 16
3.7.1 JavaScript Programming Language 16
3.7.2 jQuery JavaScript Library 18

3.8 Django Web Framework . . . 18
3.8.1 Project structure . . . 18
3.8.2 The Model Layer . . . 19
3.8.3 The View Layer . . . 20
3.8.4 Templates and Forms . . . 22
3.8.5 Authentication System and Sessions . . . 24
3.8.6 Gunicorn . . . 25

4 FaiRDy-Py: FaiRDy Python package 26
4.1 Early tests and ideas . . . 26
4.2 FaiRDy-Py General structure . . . 29
4.2.1 Basic structures and assumptions . . . 29
4.3 FaiRDy-Py structure and code . . . 32
4.3.1 Simulation class . . . 32
4.3.2 Generalized Pyramid Code class (GPC) . . . 38
4.3.3 Reed Solomon Code class (RS) . . . 40
4.3.4 Replication class (Rep) . . . 41
4.4 Package usage . . . 41
4.5 Future possibilities . . . 44
4.5.1 Additional erasure codes . . . 44
4.5.2 Optimization . . . 44
4.5.3 Adding calculations and other quality of live methods . . . 44
4.5.4 Illustrations and visualization . . . 45

5 Django web project 46
5.1 Early Ideas and Specifications . . . 46
5.2 System Design and Architecture . . . 47
5.3 Accounts application . . . 48
5.3.1 User Navigation . . . 48
5.3.2 Implementation Details . . . 52
5.4 FaiRDy application . . . 56
5.4.1 User Navigation . . . 57
5.4.2 Implementation Details . . . 58
5.5 Database . . . 60
5.6 Testing . . . 62
5.7 Deployment . . . 63
5.7.1 Django Development Server . . . 63
5.7.2 On a Home Network . . . 64
5.7.3 On Google Cloud . . . 64
5.7.4 On the UiS Unix System . . . 66
5.8 Future Possibilities . . . 67

5.8.1 Simulation Listing Page 67
5.8.2 Stripe Parameter Validation in Django Form 68
5.8.3 Async and Busy Indicator 68
5.8.4 Random Seed 69
5.8.5 Additional Storage System Parameters 69
5.8.6 Ordering Visualizations 69

6 Results **70**
6.1 Lazy Heal 71
6.2 GPC Stripe Shape 72
6.3 Repair Cycles for GPC Overlapping Stripes 73

7 Conclusion **74**
7.1 Python FaiRDy 74
7.2 FaiRDy website 75

References **77**

A Guidance for Further Development **81**
A.1 fairdypy module 81
A.1.1 Adding classes 81
A.1.2 Modifying or expanding classes 83
A.2 Django web project 83
A.2.1 /web/fairdy/models.py 83
A.2.2 Forms 85
A.2.3 /web/fairdy/views.py 89
A.2.4 /web/fairdy/templates/help/ 89
A.2.5 Adding fields to class Simulation 90

B Website documentation **91**

Chapter 1

Introduction

1.1 Goal, Motivation, and Context

The main purpose of this project is to develop software using the Python programming language which simulates the evolution of data storage systems over their life cycles. The erasure codes used to build these systems are often poorly understood, and FaiRDy-Py attempts to provide an approachable playground environment where different erasure codes and storage system parameters can be tested with low overhead to their effects on system degradation. No data storage system will ever be 100% fail-proof, but our hope is that FaiRDy-Py can help data storage professionals come ever closer to reaching the unattainable goal of perfectly reliable data storage systems.

1.2 Contributions and Limitations

The FaiRDY cellular automaton model to simulate storage system life cycles was created by our faculty advisor, Verónica Estrada Galiñanes[1] [2], who supported us in our implementation of FaiRDy-Py.

We chose to divide the project into two main modules, a Python package which is responsible for all processing of the storage system life cycle simulations, and a web framework that sits on top of this package, and provides database storage and a graphical user interface for running and analyzing simulations. Lars Kartevoll worked mainly with the package processing the simulations, and Adam Becker with the web framework.

Progress on our project was limited at times by the global COVID-19 pandemic. While we made use of online video conferencing tools, this was likely not as efficient as face-to-face meetings which we had planned.

1.3 Organization of this Document

This report is organized into 7 chapters. The first three chapters introduce our project and the various technologies we have used. Chapters 4 and 5 present the two main components of our project, the FaiRDy-Py package which processes the storage system life cycle simulations, and a web module which focuses on the storage and display of these simulations. The final two chapters provide a brief summary of what we have achieved and the insights FaiRDy provides into the study of storage systems.

Chapter 2

Project Description

In this section we introduce the central concepts that this project and thesis build upon. First, we give a brief introduction to erasure codes, they are the fundamental building block of the models we create to simulate the life cycle of large digital storage systems. Next we introduce CA (cellular automata), and the FaiRDy model that is based on them, as a lightweight method to simulate and compare the failure-repair dynamics in storage systems using various erasure coding strategies.

2.1 Erasure Codes

Erasure codes are an old discovery and have played an importance in information technology for many years. An erasure code encodes redundancy into a stream of data, this in turn ensures the data is recoverable even if certain blocks in the stream are erased or otherwise unavailable[3]. Optimal erasure codes enable recovery of data up to the point that the number of erased blocks equals the number of redundancy blocks.

In FaiRDy we define a stripe in terms of the dimensions of the erasure code to be investigated. After the stripe is defined, a specified amount of stripes are concatenated to form a complete storage system model which is then cycled between fault-injection and repair to gain insight into the storage system's properties.

2.1.1 n-way replication

N-way replication is the most simple way to store data redundantly, and in FaiRDy a n-way replication stripe consists of a single data block, and n complete replicas.

Figure 2.1: a two-way replication stripe with 3 blocks total: 1 for data, and 2 for copies of the data

2.1.2 Reed-Solomon codes

Reed-Solomon codes are optimal erasure codes, and tolerate a number of block erasures equal to the number of redundancy blocks in the stripe[4]. In real storage systems the contents of the redundancy blocks are computed and decoded using advanced linear algebra, this is however ignored in FaiRDy and a stripe will be repaired as long as enough active blocks remain to allow this.

Figure 2.2: a Reed-Solomon stripe with 9 blocks total: 5 for data, and 4 for redundancy

2.1.3 Pyramid codes

The Generalized Pyramid codes implemented by FaiRDy are composed of up to two different Reed-Solomon stripe shapes, one in the vertical direction and one in the horizontal direction[5]. This provides an extra layer of protection since each data block is a member of two different Reed-Solomon stripes, with each providing a possibility for repair. Two different Generalized Pyramid code stripe constructions are possible in FaiRDy, standard non-overlapping with the blocks arranged in the shape of an L shaped polygon, and an overlapping with the blocks arrange in the shape of a rectangle. The overlapping construction adds an additional dimension of redundancy, with the overlapping redundancy blocks computed using the vertically redundant as data input blocks[6].

Non-Overlapping

The number of horizontal stripes is equal to the number of DATA blocks in the vertical stripes, and correspondingly, the number of vertical stripes is equal to the number of DATA blocks in horizontal stripes.

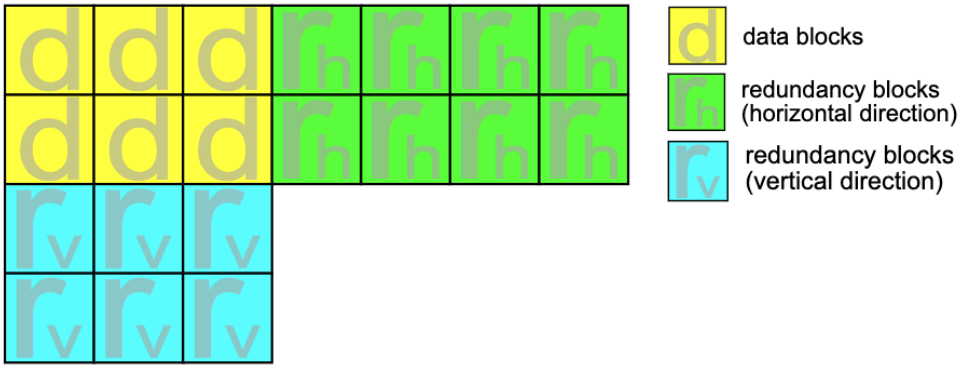


Figure 2.3: a non-overlapping Generalized Pyramid stripe with 20 blocks total: 6 for data, and 14 for redundancy

Overlapping

The number of horizontal stripes is equal to the TOTAL number of blocks in the vertical stripe, and correspondingly, the number of vertical stripes is equal to the TOTAL number of blocks in the horizontal stripe. The overlapping redundancy blocks are computed using the vertical redundancy blocks as the input data blocks, and thereby provide second degree redundancy. This is illustrated in figure 2.4.

Partially Overlapping

It is unclear to us whether partially-overlapping GPCs exist or are used in the real world, but hey are nevertheless supported by the Python module. Keeping in mind the examples for overlapping and non-overlapping GPCs in figure 2.8 with horizontal RS-stripes{3, 4} and vertical RS-stripes{2, 2}. The fully overlapping GPC(28, 6) has totally 28 blocks, and 6 word blocks. Correspondingly the non-overlapping GPC (20, 6) has totally 20 blocks and 6 word blocks. Every GPC in between which has a full vertical stripe, filling vertical stripes from left to right, is considered partially overlapping. This would make GPCs with definition (22, 6), (24, 6) and (26, 6) with the same

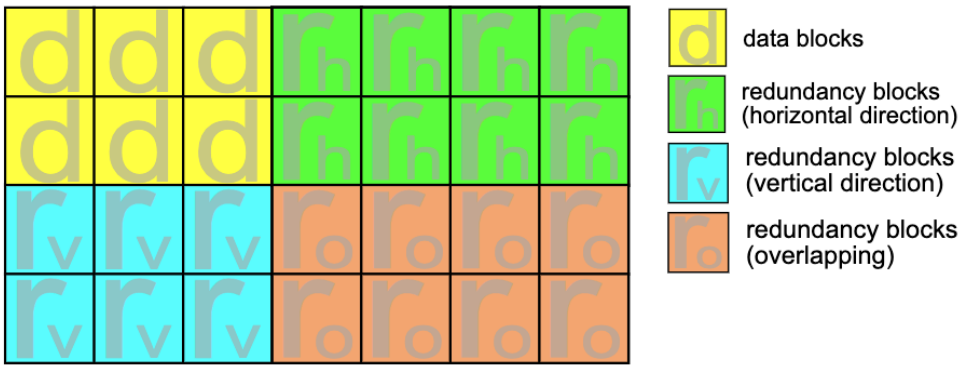


Figure 2.4: an overlapping Generalized Pyramid stripe with 28 blocks total: 6 for data, and 22 for redundancy

horizontal and vertical definition as above compatible with the FaiRDy-Py package. This is not implemented in the web module for FaiRDy.

2.2 Cellular Automata

A cellular automaton (pl. cellular automata, abbrev. CA) is a collection of boolean cells, also referred to as blocks in FaiRDy-Py, which together form a system. This system is then evolved over a set of discrete time cycles, with the state of a block in a given cycle determined by its neighbors' states in the previous cycle [7] [8]. Conway's Game of Life is perhaps the most famous example of a cellular automaton, and it is referred to as a zero-player game as a result of the fact that its entire evolution is determined by its initial state, without any further input. In Conway's Game of Life many interesting initial state configurations have been identified which can evolve perpetually, producing different patterns which can either be solid, oscillating in place, or moving across the grid of cells [9].

2.3 FaiRDy Cellular Automata model

FaiRDy is a lightweight approach to simulate the failure-repair dynamics of storage systems and explore the effects of different redundancy strategies and parameters on system degradation. FaiRDy is based on cellular automata, the blocks of storage system data are represented as cells, which can either be available or unavailable. The simulations are run in two-part cycles, in the first half of the cycle, cells are randomly marked as unavailable according

to the specified probability of failure. In the second half of the cycle, the unavailable blocks are repaired where possible according to rules of the erasure code.

In FaiRDy the CA represents a digital storage system, and the individual cells represent blocks of data contained within this system. To begin with, all data blocks in the system are considered available, however in the first step of each time cycle blocks are randomly marked as unavailable by a fault-injection module, using a pre-specified probability of failure, to simulate real world events which lead to data loss. In the second step of time cycle, an attempt is made to repair the failures introduced in the first step, whether or not this is successful is determined by a set of rules specific to the erasure code being modelled.

In FaiRDy we define a stripe in terms of the dimensions of the erasure code to be investigated. After the stripe is defined, a specified amount of stripes are concatenated to form a complete storage system model which is then cycled between fault-injection and repair to gain insight into the storage system's properties.

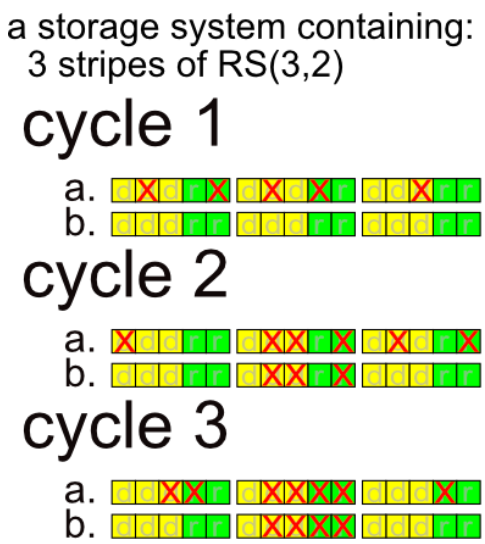


Figure 2.5: Three Reed Solomon systems going through three full cycles, each consisting of one fault injection step, and one repair step. The middle stripe shows that a RS stripe is not able to heal after the amount of unavailable blocks become greater than the redundancy blocks (green)

2.3.1 Previous Work

FaiRDy was originally envisioned and developed by our faculty advisor Veronica Estrada Galiñanes[1] [2]. The first platform was coded in the Wolfram Language[10]. The aim of the platform was to create a framework to "simulate the behaviour of different erasure codes to run simulations and understand how different code settings provide trade-offs between security and ode performance"[1]. The goal of our project was to create a freely distributed Python version for FaiRDy and it's corresponding web platform, so that people could try it for themselves.

2.3.2 Fault-injection Module

A FaiRDy simulation step in actuality consist of two steps. The first "sub-step" in a FaiRDy simulation cycle is the fault injection. In real life it can be envisioned as a failing storage device, or blocks of code being corrupted by bad code or bad actors. When blocks fail, they become unavailable to be read from the storage device and are lost. In the module this is done by setting a Boolean value from one to zero. The chance a block has to fail is based on two factors. the input `p_error` value, and if storage locations are used or not. These are explained below in chapter 2.3.3. If storage locations are not used, the module assumes that every block is in it's own independent storage location. So every block has a `p_error` chance to fail. This value is a factor between zero and one. As seen in figure 2.6, if the chance for a block to become unavailable is 0.1, then, by average, one in ten blocks will become unavailable. If the error value is 0.5, on average five in ten blocks will become unavailable. If storage locations are used. Multiple blocks are placed in the same storage location. When the fault injection is done, this is done on the storage location level. So if a location or device fails, all blocks on that device or location are no longer available. In chapter 4.3.1 under methods, the implementation of the fault-module in Python is explained.

2.3.3 Storage Array

Instead of having an equal chance for every block in the module failing, multiple blocks can as mentioned above, be stored in the same location. This let's simulations be closer to real life applications, where blocks of data would have to be stored on the same storage device, as having a separate device for every block of data would be very inefficient. The module itself has three different built in methods for distributing data across storage devices, and these can all be chosen in the web implementation. If the Python is used on

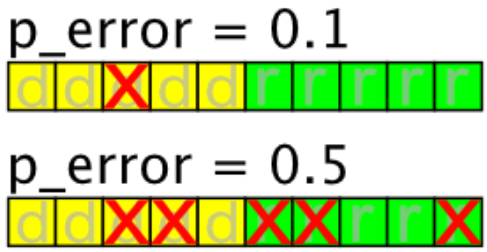


Figure 2.6: On average, with an `p_error` of 0.1, one out of ten blocks will become unavailable for every failure injection step. Similarly five out of ten with a `p_error` of 0.5

it's own it is however possible to produce custom block distributions. Storage methods and implementation is explained in chapter 4.3.1 under methods and `_create_storage_fault_array()`. Figure 2.7 shows a possible distribution of blocks in a RS(3, 2) into three locations.

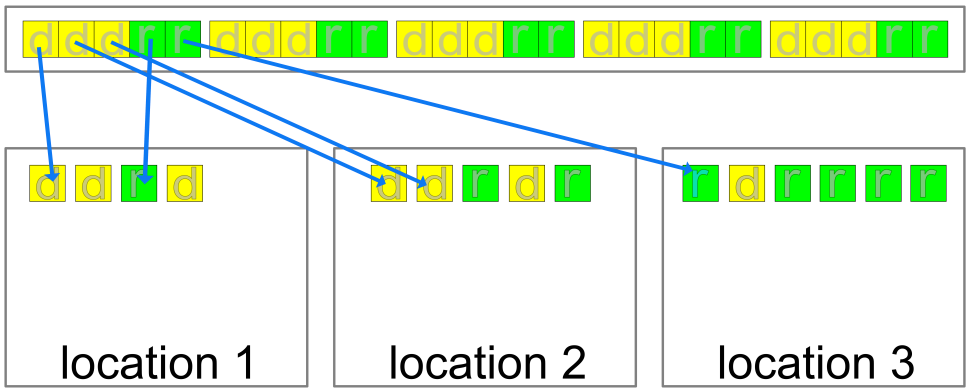


Figure 2.7: Shows a possible distribution of a Reed Solomon stripe across three storage locations

2.3.4 Repair Module

The repair module is the second "sub-step" of a full step in the FaiRDy module. After failures have been introduces, and several blocks are unavailable the repair module comes into play. It will, based on the type of erasure code, try to repair these unavailable blocks of data. The Simplest, the Replication erasure code, will as long as one of the replications, or original block exists, restore that block and its copies to full health. The next in line are Reed

Solomon erasure codes. As long as more or equal to the amount of word blocks in one stripe are still available, a stripe will fully heal during a healing step. This is illustrated in figure 2.5. The last erasure code implemented in FaiRDy is a version of the Generalized Pyramid Code. It is comprised of horizontal and vertical RS-stripes as shown in figure 2.4 and 2.3. The GPC first heals in the vertical direction, and then in the horizontal direction. An overlapping GPC will also heal it's overlapping redundancy stripes, but only in the vertical direction as visualized in figure 2.8. If an unavailable block was healed in the vertical step, that block will not be affected in the horizontal step. For the GPC it is also possible to do multiple heal-passes of vertical and horizontal heals per full step in the simulation. This is because a horizontal heal may make previously dead vertical stripes recoverable. The implementation of repair-modules are shown in chapter 4.3.2 for GPC and 4.3.3 for RS and Rep.

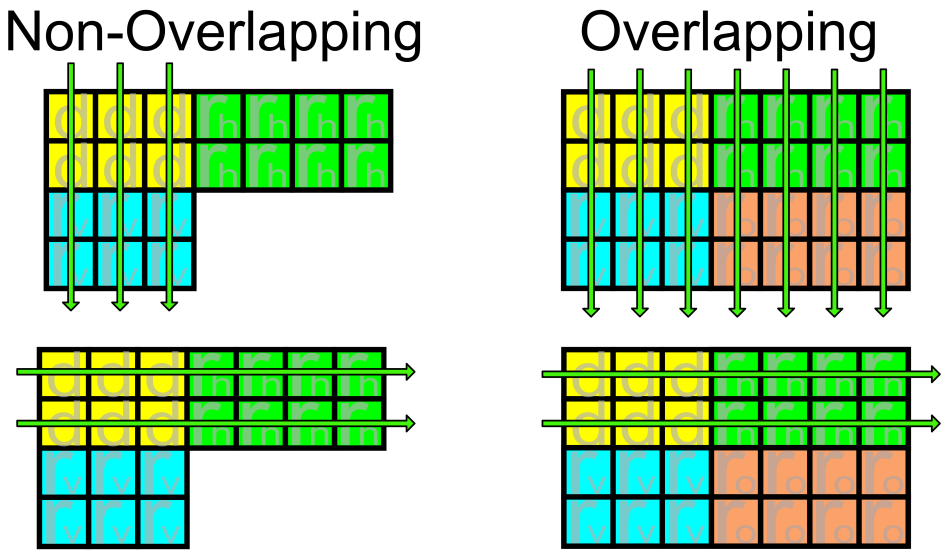


Figure 2.8: This figure shows the steps in the healing process for non-overlapping(20, 6) and overlapping(28, 6) GPCs with H{3, 4}, V{2, 2}

Eager and Lazy Policy

Before healing, it is possible to wait for more than one block in a stripe to fail, before performing a heal cycle. The reason for this, in the real world, is that restoring missing blocks is a time, and resource demanding of a system.

For this reason FaiRDy has the ability to to set a lazy heal policy. By default, the lazy heal policy is set to one. This implies that one block/cell has to be unavailable in the stripe, before a heal is attempted. This is an "eager" heal policy, as it tries to heal as soon as something is wrong. If the value is greater than one, the stripe waits for more than one block is unavailable before trying to heal it is considered a "lazy" heal policy. the `lazy_heal_policy` can be set to a value between one and the amount of extra blocks in a stripe. For GPC, it can be set separately for horizontal and vertical stripes. A heal will not be attempted until the amount of blocks specified in the threshold are missing from a stripe. Regrettably it is not possible to set different heal thresholds for subsequent heal passes in the same step. The implementation is shown in both chapter 4.3.2 for GPC and chapter 4.3.3 for RS and Rep.

Chapter 3

Technology Overview

Here we introduce the main software platforms that have been chosen for this project, including the Python programming language, various Python libraries, and Django the Python-based web framework.

3.1 Python Programming Language

The Python programming language, an open programming software managed by the Python Software Foundation, has been growing steadily in popularity the last decade[11]. Figure 3.1 shows that it has become the most popular language in Stack Overflow, one of the most popular sites on the web to ask and find the answer to programming questions and issues. It is also ranked top 3 programming languages in the TIOBE index as per May 2020[12]. Python was chosen as the programming language for this project not only because of it's popularity, but also because of it ease of use and easy readability as a language. Using significant indents forcing the user to structure code, and no need for defining type for variables, makes the language very straight forward to read. Python also has a lot of public libraries and packages openly available to everyone at no cost. In this project we have used Python 3, as version 2 has reached it's end of life[13]

3.2 NumPy Python Library

NumPy[15] is one of these libraries which is used in this project. The NumPy Library is "the fundamental package for scientific computing with Python. It contains among other things: 'a powerful N-dimensional array object, useful linear algebra, and random number capabilities'"[15]. NumPy is widely

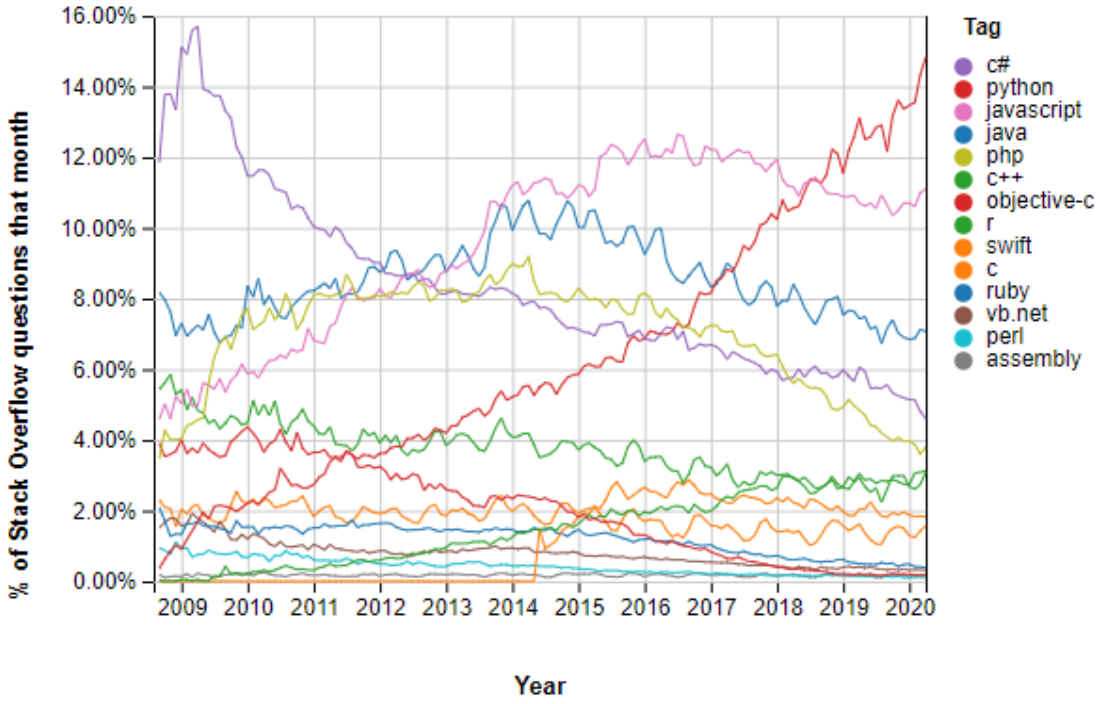


Figure 3.1: Python is one of the most used programming languages as shown by the rising amount of questions on Stack Overflow [14]

used in the Python community, and has Berkeley University as an institutional partner[15]. It is also a part of a larger scientific library for Python called SciPy[16]. NumPy was chosen over basic Python lists because of the functionalities it provides such as summation of matrices over chosen axes, easy array creation, restructuring and manipulation arrays and many more mathematical and quality of life functions[17].

3.3 Matplotlib Python Library

For creating figures and plotting results, the Matplotlib[18] plotting library has been used. This library resembles the plotting functions of MATLAB[19], but with the benefit of being open source and free. It also supports SciPy, and therefore NumPy objects, allowing one to quickly, and easily plot and illustrate results.

3.4 GitHub Open Hosting and Version Control

GitHub was used as it is a widely used platform for hosting open source code, while implementing git as it's version control system. This allows possible users to clone the source code of the FaiRDy-Py project, try it on their own computer, develop it further and make changes, and even contribute to the project in the future. While developing this project it also allowed us to work on the project at remote locations. This turned out to be very important since the project was developed during the SARS-CoV-2 Crisis.

3.5 Hypertext Markup Language (HTML)

HTML (Hypertext Markup Language) is the most basic building block of the World Wide Web. It is essentially a system of data *elements*, represented by *tags*, that tell a web browser how to render and structure an HTML page, which is also referred to as a web page. The *tags* in an HTML page generally appear in pairs, but there are exceptions to this rule. Both the start *tag* and end *tag* contain the *element* name surrounded by angle brackets, however the end *tag* has a forward slash immediately preceding *element* name. In addition to the *element* name, start *tags* can also include other *attributes* used for many purposes.

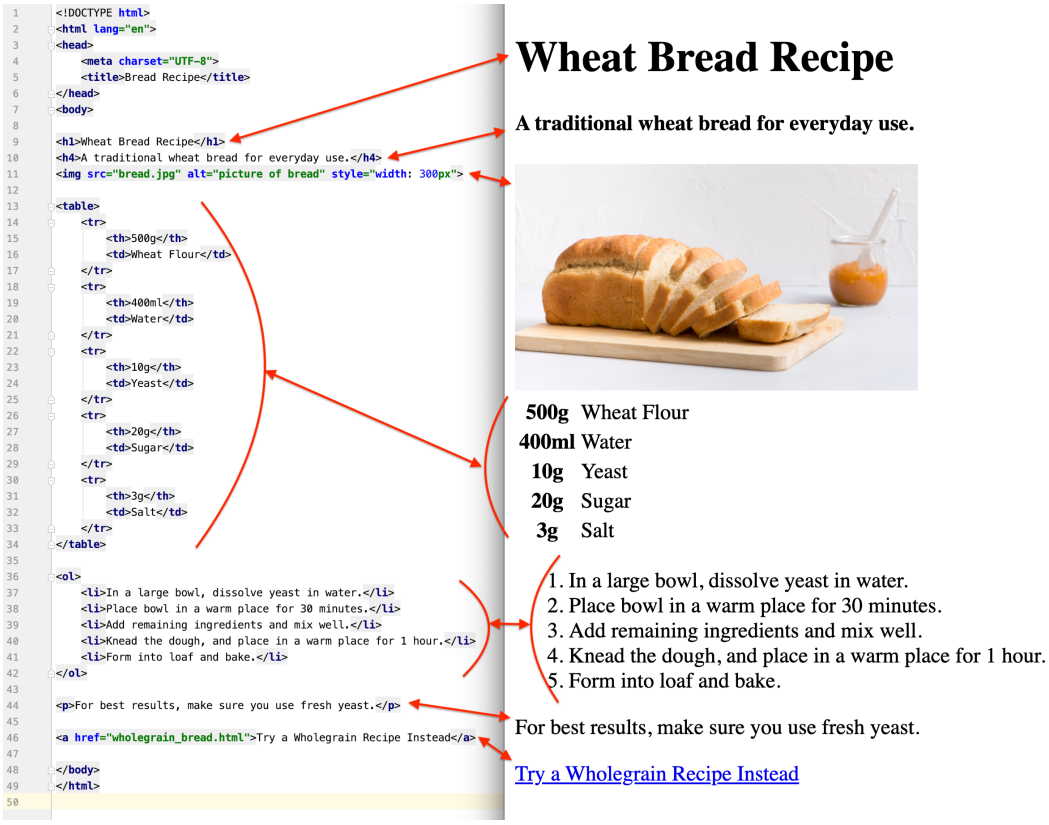


Figure 3.2: On the left, several common HTML elements such as *table*, *h1* and *h4* (headings), *img* (picture), *ol* (ordered list), and *a* (hyperlink) are composed into an HTML page. Most elements have a start `<tag>` and an end `</tag>` with a forward slash. The *img* tag is an exception with only a start tag and several additional *attributes*, including *src* to include the image's file path, and *style* to define an inline styling rule setting the image width. On the right, the page is shown after it has been rendered by a web browser.

3.6 Cascading Style Sheets and Bootstrap

3.6.1 Cascading Style Sheets (CSS)

A Cascading Style Sheet (CSS) is a collection of styling rules used to position and format the elements which are used to compose an HTML page. The word 'cascading' refers to the fact that these rules are processed in a hierarchical order, for example, a rule defined in CSS embedded inside an HTML element's `style` attribute, would take precedence over a rule defined in an external CSS file and imported using `<link>` and `</link>` tags. There are many ways to select which HTML elements styling rules are applied to. One of the most common methods, and the one most used in our project, is the addition of class attributes to HTML elements' start tags. The class attributes serve to group styling rules and facilitate the simultaneous addition of multiple styling rules to the specified HTML elements.

3.6.2 Bootstrap CSS Library

Bootstrap is a popular open-source CSS library which predefines attractive styles for most commonly used elements on HTML pages. Bootstrap originated as an internal project at Twitter in the early 2010s and once released to the public, quickly became one of the world's most popular open-source projects. To use Bootstrap on a web page, you must first add an HTML link element pointing to both the main Bootstrap CSS file and an auxiliary Bootstrap JavaScript file, the JavaScript is used to accomplish effects not possible with CSS alone. Then, by referring to the Bootstrap documentation we can find and apply the appropriate classes to style the elements which compose our HTML webpage. By using Bootstrap we are able to dramatically limit the amount of custom CSS and JavaScript we needed to write for the project, while still maintaining a reasonably attractive user interface. We primarily chose Bootstrap because of previous our experience with the framework both at UiS and privately.

3.7 JavaScript Language and jQuery

3.7.1 JavaScript Programming Language

JavaScript is a programming language and one of the core technologies of the World Wide Web. It is primarily executed on the client side, inside of a web browser, and is used to create and control the interactive user interface of most web sites on the internet. By processing code which drives

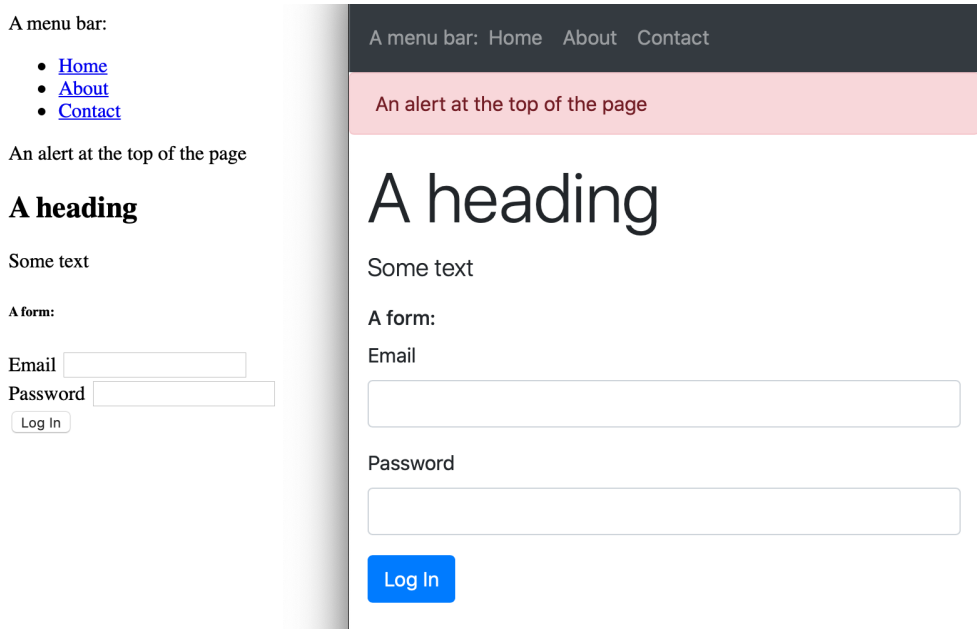


Figure 3.3: The two web pages have the exact same HTML tag structure, the only difference is the page on the right has been configured with links to the Bootstrap files and Bootstrap class attributes added to the tags.

```
<!--ALERT, AT TOP OF PAGE-->
<div>
    An alert at the top of the page
</div>

<!--MAIN SECTION-->
<div>
    <h2>A heading</h2>
    <p>Some text</p>

    <!--SIMPLE FORM-->
    <form method="post" target="#">
        <h6>A form:</h6>
        <div>
            <label for="email">Email</label>
            <input type="email" name="email" id="email">
        </div>
        <div>
            <label for="password">Password</label>
            <input type="password" name="password" id="passwo
        </div>

        <button type="submit">Log In</button>
    </form>
</div>
</div>
```

```
<!--ALERT, AT TOP OF PAGE-->
<div class="alert alert-danger">
    An alert at the top of the page
</div>

<!--MAIN SECTION-->
<div class="container">
    <h2 class="display-4">A heading</h2>
    <p class="lead">Some text</p>

    <!--SIMPLE FORM-->
    <form method="post" target="#">
        <h6 class="">A form:</h6>
        <div class="form-group">
            <label for="email">Email</label>
            <input class="form-control" type="email" name="email"
        </div>
        <div class="form-group">
            <label for="password">Password</label>
            <input class="form-control" type="password" name="pas
        </div>

        <button type="submit" class="btn btn-primary">Log In</but
    </form>
</div>
</div>
```

Figure 3.4: The HTML for the previous webpage, with the Bootstrap class tags added and highlighted on the right

a web site's user interface on the client side, we spare resources on the web server and limit the impact of network congestion on user experience. While JavaScript syntax is similar to the Java programming language, and also partly inspired by it, they are completely separate programming languages with very different areas of application.

3.7.2 jQuery JavaScript Library

jQuery is a library written in the JavaScript language which is used by over 70% of the internet's most visited sites. Its main purpose is to simplify and reduce the lines of code required to accomplish common tasks related to manipulating the data and elements on an HTML web page. Bootstrap requires jQuery, this combined with our previous experience with the library made the decision to include it in our project very easy.[20]

3.8 Django Web Framework

Django is an open-source web framework, one of several written in the Python language. [21] A web framework contains libraries of pre-written computer code which seek to automate and streamline many common web development tasks. The modules are generally written, reviewed, and updated by a community of experts using an online collaboration platform such as GitHub. This enables a developer to focus on what is unique for their web project and rely on the web framework to handle lower-level details which are common across many web projects, such as managing user sessions and database connections.

3.8.1 Project structure

The top level of any Django web application is called the *project*, it exists as a file system directory. All Django projects contain at least one *application* which exist as sub-directories of the main project directory. The purpose of *applications* is to group modules with common data and related behavior together into units of a manageable size which are easy to reuse in other projects. By default a Django project will contain an application directory with the same name as the project, this contains modules which control settings for the entire project, most notably *settings.py* and how the individual applications are configured as part of the larger project.

The project directory contains a module *manage.py* which is used as a command-line utility to perform various tasks on the project, for example

running the built-in test server or creating database tables. The main project directory will also contain directories of other file types which are relevant to the entire project, not just individual applications, including static files and HTML templates.

Each application directory added by a web developer to the project will contain several modules defining the application's behavior, and potentially sub-directories which contain files specific to the application, like HTML templates and custom utilities. Among the most important modules contained in an application are models.py which is used to define how the application's data is structured, manipulated, and mapped to a database, *views.py* contains all of the functions which are necessary to process user input and respond appropriately, while *urls.py* maps specific URLs, web addresses, to the functions contained in views.py so they are accessible in a web browser.

3.8.2 The Model Layer

Django documentation defines a model in the following way:

> A model is the single, definitive source of data about your data. It contains the essential fields and behaviors of the data you're storing. Generally, each model maps to a single database table.[22]

A Django model defines the tangible concepts an application is meant to manage, and how they interact with each other. Each model contains one or more fields, each of which hold a piece of data like a number, a date, or a string of text. In addition to these simple field types, models can also have fields which contain another model, or lists of other models, to define the relationships between models. Model names are usually nouns, and an application's models are defined in its models.py file, as individual Python classes, which all inherit from a base class contained in Django's Django.db.models module. By inheriting from this module, our models gain a wide range of useful fields and functions predefined by Django, in addition to the ones we define ourselves. Models also essentially define the database layout, Django's command line utilities invoke an object-relational manager to create a database structure reflecting the models, and also update the database safely when models must be added, updated, or removed.

Figure 3.5 is used to illustrate the Django models, and corresponding database tables, which might be used in a simple Django application called boats, for a boat rental company. The structure of the Django project is shown on the top left, with the models.py file for the boats application located inside the application folder, and indicated by a red box. Three models are defined inside Python classes in the top center: Customer, Boat,

and Reservation. The models have several different types of fields, including CharField for a string of text, PositiveIntegerField for a positive whole number, and DateTimeField for a date time combination. Additionally the Reservation model has two ForeignKey fields, representing relationships with the Customer and Boat models. In our example, an individual reservation has one customer and one boat, with each customer and each boat possibly associated with many different reservations. Even though it is not directly defined in the model, Django will add an extra field to both the Customer and Boat models called reservation_set, which references all the reservations for the boat or customer. On the bottom of the figure, two commands using the project's manage.py module are indicated with red boxes. The first command, **makemigrations boats**, looks at the current state of the database and then creates a list of instructions to bring the database's structure into alignment with the structure of the application's models. The second command, **migrate**, applies these instructions to the instance of the database defined in the Django project's settings.py file. On the top right, the resulting database structure is shown, the database table names are the Django application name and model name separated by an underscore. An additional field, id, has been automatically added to each table. This field is an automatically incrementing integer (whole number) which is guaranteed to be unique for each record in the database, called a primary key. A table's unique primary key is subsequently stored as a foreign key in other tables to represent the relationships between them.

3.8.3 The View Layer

In Django, a *view* is view is meant to contain the logic responsible for processing a user's request and returning a response to them. Views are Python functions contained in the Django application's views.py file, these functions are mapped to URLs in either the urls.py file belonging to either the Django application, or the entire Django project. On a Django website, when a user clicks a link or enters a URL in their address bar, Django will first check the project's urls.py for an entry matching the URL, if none is found an error page will be returned. It is also possible to add a line to the project's urls.py file which will defer all URLs matching a certain pattern to an application's urls.py file, and the final match is made here. This ensures the project's URL file remains a manageable size and the relevant information is stored within the application it is unique to.

Continuing with the example from the previous section, regarding a web application for a boat rental company, it is now possible to define Django forms based on the models, and build views to send these forms to the user

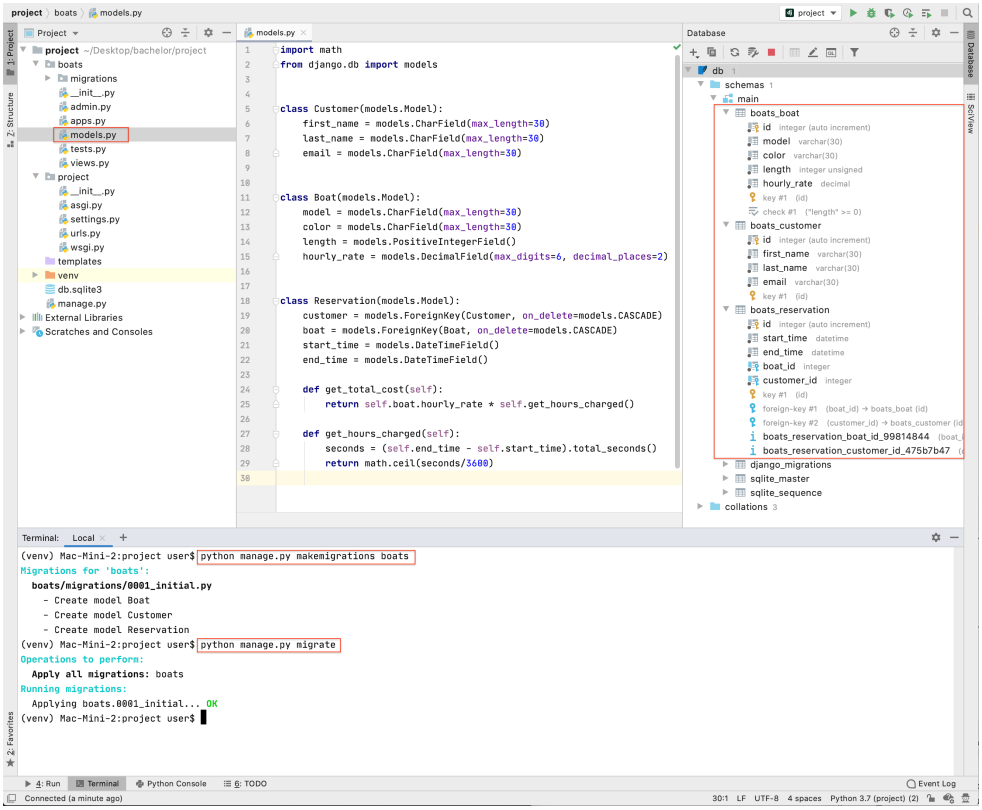


Figure 3.5: top left: location of *models.py* in the Django project structure
top center: models defined as Python classes inheriting from *Django.db.models* module
top right: tables generated in the database by Django
bottom: commands entered to transform the classes in *models.py* into database tables

and process the input which is returned.

```
class BoatForm(forms.ModelForm):
    class Meta:
        model = Boat
        fields = ['model', 'color', 'length', 'hourly_rate']
```

Listing 3.1: Simple Django form, defined in the application's forms.py module, which specified the fields from the model which should be included.

```
def new_boat(request):
    if request.method == 'POST':
        form = BoatForm(request.POST)
        if form.is_valid():
            form.save()
        redirect('boat_list')
    else:
        form = BoatForm()
    return render(
        request,
        'boat_form.html',
        {'form': form}
    )


# allows users to view a list of boats
def boat_list(request):
    boat_list = Boat.objects.all()
    return render(
        request,
        'boat_list.html',
        {'boat_list': boat_list}
    )
```

Listing 3.2: Two functions in the application's views.py module, *new_boat(request)* either sends the user a form to add a new boat to the database, or validates the user's input and stores it to the database, depending on whether the request method is a HTTP 'get' or 'post'. *boat_list(request)* queries the database for a list of all boats it currently contains, and then passes this list to the Django templating engine together with the HTML file *boat_list.html* to be rendered and returned to the user.

3.8.4 Templates and Forms

Templates are special HTML files that define the structure and main composition of a web page, but not the actual data the page will contain. When a template-based web page is requested, a templating engine inserts data into

specified positions in the template and also processes other simple computer programming control structures inserted directly into the HTML syntax, such as if, while, for, etc. HTML templates can generally also be composed of other HTML templates, this means that an item like a navigation bar, that appears on the top of every page on a web site, is only defined in one single place and not across possibly hundreds of individual HTML files. The reuse of a single HTML template to generate HTML pages on-the-fly using data from a separate data source, also allows data to be stored in an appropriate and central place, like a database, instead of spread across hundreds of HTML files. files.[23] Django has its own templating engine, which uses a Python-based syntax to insert dynamic content into HTML templates. This provides a range of built in *template tags* and filters which can be used to process and display the data-containing variables passed into the templating engine along with the HTML template engine.[24]

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Boat Form</title>
</head>
<body>

<form action="{% url 'new_boat' %}" method="post">
    {% csrf_token %}
    {{ form }}
    <input type="submit" value="Submit">
</form>

</body>
</html>
```

Listing 3.3: The HTML template which is combined with the Django form inside the view function, and returned to the user. The special non-HTML variables and commands used by the HTML templating engine are enclosed inside {% %} and {{ }} sections. By simply including {{ form }} Django renders all fields specified in the forms.py module into HTML input elements appropriate for the fields' data types.

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Boat List</title>
</head>
<body>
```

```

<a href="{% url 'new_boat' %}">Add a new Boat</a>

<table>
    <tr>
        <th>model</th>
        <th>color</th>
        <th>length</th>
        <th>hourly rate</th>
    </tr>
    {% for boat in boat_list %}
    <tr>
        <td>{{ boat.model }}</td>
        <td>{{ boat.color }}</td>
        <td>{{ boat.length }}</td>
        <td>{{ boat.hourly_rate }}</td>
    </tr>
    {% endfor %}
</table>

</body>
</html>
```

Listing 3.4: This HTML template is used to generate a table containing a list of all boats currently in the database. After being combined with the variable *boat_list* in the templating engine, the {% for boat in boat_list %} structure creates an HTML table row for each boat in the database.

3.8.5 Authentication System and Sessions

Django has its own user authentication system, this handles user accounts, passwords, and cookie-based sessions to name a few features. The base authentication system provides a *User* model with many common fields and behaviors for a website user account. The User model provides functions to check password strength and also to hash and salt passwords before storing them securely in the database, not in plain text format. The cookie-based sessions pass a small text file, referred to as a cookie, to the web browser of logged-in users along with the web pages. When the user then requests a new page or submits a form, the cookie is sent back to the server along with the request, allowing the sever to access additional information about the session such as the current user. Hypertext Transfer Protocol (HTTP), along with Hypertext Transfer Protocol Secure (HTTPS), define the rules for sending HTML web pages over networks, and are referred to as stateless protocols. Stateless refers to the fact that each request is sent as a single entity, without any implicit connection to other requests before and after, and this is

the reason we need to use cookies. Without the cookie to track the user's session, it would be nearly impossible for the server to know whether request originated from a specific logged-in user or an anonymous hacker, and then verify this user has the permissions required to execute the request.

3.8.6 Gunicorn

Gunicorn, short for 'Green Unicorn' is an open-source HTTP web server for Unix-based operating systems like Linux. While Django's development server is convenient for use while writing developing a website, it is not designed or tested to be deployed in a production situation.[25] Dedicated HTTP server software like Gunicorn has been thoroughly tested to respond securely, quickly, and efficiently to HTTP requests coming from multiple users simultaneously. One may wonder why Django doesn't just build an HTTP web server in like it does with so many other components, the reason is that HTTP web servers like Gunicorn are a specialized communications tools, and web frameworks like Django are specialized content creation tools, both requiring expertise in very different fields to perform securely and efficiently. Gunicorn communicates with the Django project across what is called a Web Server Gateway Interface (WSGI, pronouced 'whiskey'), this is a specification which defines how HTTP servers communicate with web frameworks like Django.[26] Gunicorn is highly configurable, however it may be launched as simply as by passing it a wsgi.py definition module belonging to a web application. Django web projects include this wsgi.py file in the main project folder, together with settings.py.

Chapter 4

FaiRDy-Py: FaiRDy Python package

The FaiRDy-Py python package has been created with the intent for it to be easy to use and show the principals presented in chapter 2 for failure-repair dynamics and redundancy strategies. The package is openly available on GitHub in its entirety on $https://github.com/nextgen-codes/fairdy-py$.

4.1 Early tests and ideas

In the beginning of the project, a simple proof of concept was made for Reed Solomon simulations. The early implementation was to get a feel with the idea and visualize how the simulation would behave as time passed. The implementation did not store any history, only current step information was accessible, but past information was printed to console to visually inspect. Both failure- and repair-steps were printed to console so that ideas of how the FaiRDy-Py module would behave could be seen for simple RS-codes.

After the initial tests, some general ideas and principles were laid down:

- Historic information of simulation progress to be available
  - Block availability factor
  - Fault injection step
  - Repair step
- Structured so that future modification would be easy
- Speedy simulation even for large redundancy schemes

- Hierarchical structure to easily see implementation and connection between GPC, RS and Rep

The first iteration was an object-based approach. The general idea was that each object in a simulation would be created initially. As the simulation runs, object references would make storing information and referencing the correct blocks easy.

The objects were planned to be:

- Block
- Reed Solomon code
- Generalized Pyramid code
- Simulation

The block would store block-specific information, like how many times it had died, how many times it had been successfully healed, what type of block and so forth. Next an object for a Reed Solomon stripe was created. On initialization, an array stored on the Reed Solomon object would be populated by block objects. The RS-object would also store information about size, word-blocks, extra blocks, lazy heal threshold and more. Functions handling healing of the stripe was also defined. Next was the simulation. A base class was created for which to base all other simulation classes on. A simulation for a GPC or RS would then inherit functions and definitions from this base class. This would make it possible to define general functions which would hold for simulations of all erasure codes, at the same time making it possible to create specialized functions per erasure code to increase processing time. The simulation object held an array containing containing every RS-stripe. To run the simulation, a loop through each stripe object in the array would inject faults and then repair for each step. A move away from this object-implementation was made because when keeping a history of previous simulation steps, just making a copy of a previous state is very time-consuming when having to copy a great amount of objects. There are, of course, smarter ways of implementing the history, but the move was made towards using "NumPy" which "adds support for N-dimensional array object useful linear algebra and random number capabilities" [15]

```
class block:
    def __init__(self):
        self.status = 1 #alive or unavailable
        self.type = 0   # wordblock or extrablock
        self.times_died = 0 # counter for deaths
        self.times_healed = 0 # counter for heals
    def fault(self, p_error):
        if random.random() < p_error:
            self.status = 0
            self.times_died += 1
    def heal(self):
        if self.status == 1:
            #Check for internal error while developing
            raise Exception("a live block cannot be healed")
        self.status = 1
        self.times_healed += 1
```

Listing 4.1: FaiRDy-Py block class showing how stats can be stored per block on a block object by changing attribute values

The second iteration of FaiRDy-Py, as stated earlier, moved away from the per block object-based approach to conserve memory and increase calculation speeds. Since NumPy is widely used for scientific purposes and array programming in Python, NumPy arrays were chosen as the main structure. In this reiteration one block is a 0 or 1 stored in an array. each stripe is stored as a one-dimensional array so the full structure is a m times n array where m is the number of blocks in a stripe and n is total the number of stripes for a simulation. The Simulation structure was continued from the previous version. A base simulation class from which both the RS, GPC and Red objects would inherit properties and methods which are shared. The amount of memory used by using NumPy arrays over separate object for each block was reduced significantly. Processing speed also increased greatly as stripes could be processed and altered in one operation instead of having to set the value of each block individually. This allowed for much larger simulations, and also keeping the history for each step in the simulation larger simulations if needed.

The third and final reorganization was made while having to code functions separately for GPC, RS, and Red. Keeping track of changes made to one of the classes and making sure that they all fit recent changes. We then realized that every erasure code we were working with was some GPC' special case. Each class could therefore be realized by construction the class above it in a hierarchical sense.

4.2 FaiRDy-Py General structure

The general structure of the final version of FaiRDy-Py follows from experiences and ideas learned from version one and two as mentioned above. A class based hierarchical structure with inheritance making sure that a special case object would process as expected, but could be improved with specialized handling for each erasure code type. A Simulation class is the base class to base all other erasure codes on. It is meant to be used as an abstract class (although not implemented in python as such[27]). This class holds definition for needed methods and attributes for all subsequent classes, and cannot be initiated. The first realizable class is the Generalized pyramid code, it inherits from the Simulation class. Reed Solomon code is a special case of GPC, and Replication is a special case of RS. The package is stored as four files, one file for each class. The package `__init__`-file makes it so that the simulation class is not imported as a class which can be created.

4.2.1 Basic structures and assumptions

Erasure codes in FaiRDy-Py

In an erasure code simulation the structure for blocks is stored in a matrix. The x-axis of the matrix corresponds to every stripe in the simulation, while the y-axis corresponds to the blocks in each stripe.

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{bmatrix}$$

Figure 4.1: An array holding three Reed Solomon stripes of five blocks where green blocks are main blocks (m), and blue blocks are extra blocks (k)

Figure 4.1 shows a Reed Solomon stripe as its created and stored by the FaiRDy-Py package. This also holds for a GPC, although its storage orientation is not as straight forward. Figure 4.3 shows how: Generalized Pyramid Codes are normally displayed in figure 4.3a , it's corresponding indexes when stores as a one dimensional array in figure 4.3b, and how the array is stored by the FaiRDy-Py package with both values and indexes on one row each in figure 4.3c. Each stripe is stored as a one dimensional array in the package for easy access and looping. This was chosen because every stripe can be seen as independent when it comes to repair unavailable blocks. When an erasure code object is created in Python. It will initiate by creating an array of live and empty blocks for the specified erasure code structure, size

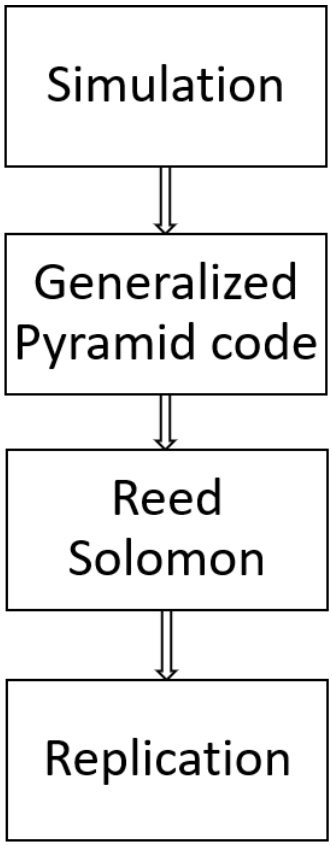


Figure 4.2: This is the hierarchy of class objects and inheritance in FaiRDy-PY. Inheritance goes only one way, with the arrow in the illustration

and number of stripes. Empty blocks are set to the negative sum of horizontal and vertical length. This is so that the sum of a stripe with empty blocks is always negative, and will never be repaired. When creating the structure it will also perform a number of sanity checks to check that the specified size is an actual valid structure. If not, it will throw an error to prevent unintended behaviour. When a storage mode is created, a mirror-array of the same size is created holding the storage location for a block. How blocks are distributes is based on the mode chosen, and may also be customized. This is discussed closed under chapter.

block fault injection and healing

After the erasure code object has been created, it is ready to run simulations. These are run by executing the function: "`loop_simulation(loops,`

`p_error)`" on the erasure code object. The simulation will then run through the specified number of loops. For each loop, the simulation will first inject failures based on the `p_error` input. The `p_error` can either be a single value between zero and one, or it can be an array of values the same length as number of loops. It will then use the percentage for the index corresponding to loop the simulation is at. This functionality is however not implemented in the web interface. Failure modes also vary between random, and a few storage modes, which are chosen when creating the erasure code object. When random mode is used, it behaves as if every block is stored on a separate storage device and there is a chance for each device that it will fail. Storage modes place blocks in storage locations, and there is a chance for each storage location that all blocks in that location will die.

After failures have been introduced, the erasure code will try to repair unavailable blocks in each stripe one at a time. Repair procedures are based on type of erasure code. for RS the stripe is a one dimensional array, so there is only a one dimensional heal check. If remaining live blocks in the stripe are more than or equal to number of initial word-blocks, the stripe will fully heal. If not, it will remain in its current state. If the stripe already is fully healed, it does nothing. Possibly failing more for each step, until all blocks have died or the simulation ends. The GPC works in a similar manner. The difference being that it is composed of both vertical and horizontal RS-stripes. The GPC will first go through vertical stripes and attempt to heal them. Then after the vertical stripes it will go through the horizontal stripes.

When healing with the erasure code some assumptions are made. when storage locations are specified, blocks will be stored in "the same" location after healing. the assumption correlates to a failed storage device being replaced by a new one, and all healed blocks being placed on the replacement. The amount of storage locations will therefore remain the same during the whole simulation. Lazy heal thresholds may also be set for healing. A lazy heal threshold is implemented in such a way that the number of missing blocks in a stripe must meet the lazy heal threshold before an attempt to heal is made. The threshold cannot be larger than the number of extra blocks, and by default it is set to one. This implies that by default it will attempt to heal as soon as one block is missing in a stripe.

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & -6 \end{bmatrix}$$

(a) General pyramid code (8, 4), H{2, 1}, V{2, 1} as commonly displayed

$$\begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

(b) Indexes of figure 4.3a when stored by the FaiRDy-Py

$$\begin{bmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & -6 \\ 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \end{bmatrix}$$

(c) showing one stripe of the (8,4) GPC as initiated with values in the first row, and indexes in the second row

Figure 4.3: GPC(13, 6) as it is commonly displayed, and it's indexes when stored as a one dimensional array in FaiRDy-Py. Green block are main blocks, blue blocks are horizontal extra blocks and red blocks are vertical extra blocks

4.3 FaiRDy-Py structure and code

This chapter will go through each class of the FaiRDy-Py package, and explain how it works and operates.

4.3.1 Simulation class

Initiation and attributes

The simulation class is, as mentioned above, the building blocks and the main class from which all other classes inherit. This class has methods for initialization of the erasure code object as well as general sanity checks for storage settings when creating a class object. It also has a lot of attributes for storing settings and information about the erasure code. The initiation of a simulation object requires only two parameters of the total 10 possible. The required inputs are `blocks`, which is the total amount of blocks in one stripe, and `num_of_stripes` which is the number of stripes in the object. The other optional parameters include: `storage_fault_mode` and `num_of_storage` which control the storage modes. These are by default set to None (Python Null). `lazy_heal_threshold_hor` and `lazy_heal_threshold_vert`. These set the amount of blocks which have to die in a stripe before a heal is attempted. It is by default set to one. `repair_cycle` sets how many cycles to perform of vertical and horizontal heals, only usable by GPC. `include_history` and `include_fault_history`, these are both by default

set to False, and tells the module if it should store every step in the simulation. `verbose` is False by default. If set to true, it will print out messages while setting up and running the simulation. When the Simulation object is initialized, it creates an array of live blocks in the size of (`num_of_stripes, blocks`) using NumPy. it then performs a number of setup sanity checks to make sure that nonsensical erasure codes are not created. The sanity checks are explained under Methods.

Methods

Methods will be discussed and explained in this section. Ideas and methods and assumptions while using the module will be explained.

`_storage_check()`:
performs a self check that `storage_fault_mode` is one of the predefined modes. These modes are explained under `_create_storage_fault_array()`.

`_storage_check()`:
does some simple type checks for `num_of_stripes`, `num_of_storage` and `repair_cycle` ensuring that they are all integers.

`_create_storage_fault_array()`:
Listing 4.2 generates an array the same size as the storage array. containing the storage location for each block. There are three implemented storage location methods: `random`, `equal_shuffle`, and `hash_block_index`. It is also possible to create your own using mode: `custom`.

`random`: for each block, a random integer between 1 and the number of storage locations using `random.randint(a, b)` from the Python standard random library [28]. Set this number as the storage location for said block. A random seed can be set using `random.seed()`.

`equal_shuffle`: Distributes the blocks as evenly as possible among the storage location with the intention of having an equal amount of blocks in every location. for every block:

1. Take the block id, a number between zero and the total number of blocks in the system minus one
2. modulo the result with the number of storage locations
3. add one to the result from step 2
4. after this has been performed for each block, shuffle the array.

Output from step 3 is a one dimensional array cycling over number of storage locations. For three storage locations it would look like this: $[1, 2, 3, 1, 2, 3, ...]$. When the array is shuffled, blocks will be randomly distributed equally in every storage location. A random seed can be set using `numpy.random.seed()`.

`hash_block_index`: This approach works in a similar way to `equal_shuffle`, but uses a hash-value instead of index and random to create a different distribution.

1. take the block id, a number between zero and the total number of blocks in the system minus one
2. take the string version of the resulting number
3. apply hash function to the string number
4. modulo the result with the number of storage locations
5. add one to the result from step 4
6. set this number as the storage location for the block

This creates a number between 1 and the number of storage locations for each block. The hash-function used is part of the built-in Python functions [29]. The string version of the number is used since for the built-in version of the hash-function `hash(int) = int`. The hash-seed is by default random for each new instance of python, but can be controlled by setting the environment variable `PYTHONHASHSEED`.

`custom` lets you define your own custom storage fault array. This can be done by initiation the erasure code, then setting `.storage_fault_array` equal to your custom array. The array must be the same shape as the erasure code array (number of lines, blocks in each stripe), and have the same amount of storage locations as defined in the erasure code, or unexpected behaviour will be encountered.

```
def _create_storage_fault_array(self):
    # places the blocks randomly into storages
    if self.storage_fault_mode == "random":
        fault_array = [
            random.randint(1, self.num_of_storage)
            for block in range(self.num_of_stripes
                * self.blocks)]
        self.storage_fault_array = np.reshape(fault_array,
        (self.num_of_stripes, self.blocks))
        return

    if self.storage_fault_mode == "equal_shuffle":
```

```
        # Places the blocks randomly in
        # equally large (if possible) capacity
        # for 100 blocks in 10 storages
        # you get 10 blocks per storage.
        # does not account for empty blocks
        fault_array = np.array([
            (block % self.num_of_storage) + 1
            for block in range(self.num_of_stripes
            * self.blocks)], dtype=int)
        np.random.shuffle(fault_array)
        self.storage_fault_array = np.reshape(fault_array,
            (self.num_of_stripes, self.blocks))
        return

    if self.storage_fault_mode == "hash_block_index":
        # Apply hash function to the string
        # value of the index of the block modulo
        # the hashvalue with number of storage locations
        # add 1 to get a storage placement
        fault_array = np.array([
            (hash(str(block)) % self.num_of_storage) + 1
            for block in range(self.num_of_stripes
                * self.blocks)], dtype=int)
        self.storage_fault_array = np.reshape(fault_array,
            (self.num_of_stripes, self.blocks))
        return

    if self.storage_fault_mode == "custom":
        return
```

Listing 4.2: Function showing how the arrays holding the storage location is created based on different methods.

```
def _fault_injection(self, p_error):
    #Fault only for valid blocks
    if (self.num_of_storage is not None and
       self.storage_fault_mode is not None):
        # create empty fault array
        fault_array = np.zeros(self.array.shape)

        # roll chances to survive per storage
        chance_to_survive = np.random.rand(self.
   num_of_storage)

        # Create fault_array
        for i, survival in enumerate(chance_to_survive):
            fault_array = (fault_array +
                np.multiply((self.storage_fault_array
                    == i+1), survival)
```

```

        # fail storages where
        # survival_chance is below p_error.
        # In cace of non-overlapping GPC,
        # preserver empty blocks
        fault_array = np.multiply(self.array, fault_array)
        fault_array = ((fault_array > p_error)
            + (fault_array < 0))
        self.array = np.multiply(self.array, fault_array)
    else:
        fault_array = np.multiply(self.array,
            np.random.rand(self.num_of_stripes,
                self.blocks))
        # if arrays are empty they are
        # set to a negative number.
        # since the fault_array >= p_error
        # is mutually exclusive to fault_array < 0
        # these can be added, and empty blocks
        # will never be altered
        fault_array = ((fault_array > p_error)
            + (fault_array < 0))
        self.array = np.multiply(self.array, fault_array)
```

Listing 4.3: Function showing how the array holding the storage location for each block is created

`_fault_injection(p_error)`:
This function in Listing 4.3 introduces failures into the erasure code array based on the `p_error` input. The fault injection works similarly for both storage methods and when no storage is chosen. But there is a slight difference in implementation. When both `storage_fault_mode`, and `number_of_storages` are defined the fault are introduced in the following manner described below. for no storage location, the only difference is instead of rolling a chance of survival for each storage location, it rolls for every block.

1. Create an array of zeros the same size as the block array
2. Create an array the size of number of storage locations called `chance_to_survive`
   (a) For each storage location, roll a random number between 0 and 1 and store in the array from the step above
3. for each entry in `chance_to_survive`
   (a) take the index of the entry add one to get storage location

   (b) multiply, block-wise, the `storage_fault_array` where storage location equals the result from above with the chance to survive entry.
   (c) add the resulting array to the empty array of zeros

4. take the resulting array, containing the chance to survive for each block based on it's storage location and multiply with the block array
   - This produces an array containing positive numbers, which is the chance to survive for all live blocks, negative numbers for all empty blocks, and zero for all previously unavailable blocks.
5. Based on the previous array, create a new array setting ones for all blocks where the chance to survive is greater than the `p_error` input, and ones for all blocks where there are negative numbers.
   - This array now contains a one for every block which survives, and a one for every empty block. All dying, or already unavailable blocks will be 0
6. multiply the array above with the block array, the result is failing the apropriate blocks

`_threshold_check()`:
Performs a simple check on lazy heal thresholds to check that they are not larger than the number of extra blocks.

`loop_simulation(loops, baf_limit=0.0, p_error=0.2)`:
This method is executed on the erasure code object, and performs the actual simulation. The inputs are `loops`, the number of loops to perform. `baf_limit` with default value 0.0. If this baf-limit is reached before the total number of loops. the simulation will stop. `p_error` with default value 0.2, is the chance for a block or storage location to fail while running the simulation and must be between zero and one. `p_error` can also be passed as an array of floats. The array must then be the same size as loops, and the error value used for each loop will correspond to the value at the corresponding index in the array.
The simulation, will for each loop first inject failures based on `p_error`. Then a vertical heal is run (if `horizontal_extrablocks` is greater than 0, currently only for GPC). before running a horizontal heal. These steps will be repeated multiple times for each loop based on the `repair_cycle`. At the end of each loop, the BAF is updated, and compared to the `baf_limit()`.

There are three other methods: `_self_vertical_heal()`, `_self_horizontal_heal()`, and `_update_baf()`. These are not implemented, but simply defined, for use in the `loop_simulation`. These are implemented in GPC, and subsequent erasure codes if simpler or different implementations are available or needed.

4.3.2 Generalized Pyramid Code class (GPC)

Initiation

The GPC inherits directly from the Simulation object. And starts by initiating, and therefore inheriting, from the Simulation object. A simple initialization of GPC can be seen in figure 4.4. The GPC class has 4 required parameters: `gpc_definition` which is a tuple containing the total amount of blocks in the GPC, and the amount of wordblocks in the GPC. `horizontal_rs` is a tuple containing the word blocks and extra blocks of the horizontal RS stripes and `vertical_rs` is a similar tuple defining the vertical stripes. `num_of_stripes` is the same as for Simulation in chapter 4.3.1. The optional parameters are the same as for Simulation in Chapter 4.3.1.

```
import fairdypy as fp
gpc = fp.Generalized_pyramid_code(
    gpc_definition=(16, 8),
    horizontal_rs=(2, 1),
    vertical_rs=(4, 2),
    num_of_stripes=800
)
```

Listing 4.4: An example creating a (16, 8)-GPC with 800 stripes

Methods

All methods which are not redefined here, are inherited from the Simulation class

`_self_check_gpc()`:
This method does a number of sanity checks when creating the GPC-object. checking that word blocks cannot be less than zero, total blocks in GPC cannot be less than word blocks, if total amount of blocks in GPC equals word blocks, there can not be any extra blocks. GPC word block definition must match the product of vertical and horizontal word blocks, and total amount of blocks cannot be greater than the product of vertical and horizontal stripe length. lastly there is an implementation supporting partial overlaps, but it is unclear if these are actually

implemented in the real world. If a partial overlap is created, the overlap must create a full vertical stripe.

`_check_overlap()`:
This methods checks what kind of overlap the created GPC has. Either full overlap, no overlap or partial.

`_update_baf()`:
Calculates the new BAF-value by summarizing all of the live blocks, and dividing on the original number of blocks. The GPC-module accounts of empty blocks which often occur in GPC.

`_initiate_empty_blocks()`:
If the GPC is not fully overlapping it will initiate empty blocks as the negative sum of horizontal length and vertical length. This will stop these rows from ever healing according to current implementation.

`_self_vertical_heal()`:
A vertical heal is performed with the following steps for every stripe in the GPC erasure code.

1. Create a restructured copy the one-dimensional array. The restructured copy is in the actual shape of the GPC.
2. create a reverse of the resulting array from above. Setting live blocks to zero and unavailable blocks to one and also setting empty blocks to zero.
    - Adding this matrix to the above copy would create a fully repaired GPC.
3. Summarize the array from step 1 over the vertical axis.
4. Loop through each sum and set all blocks in the corresponding row in the array from step 2 to 0 if one of the following
    (a) The vertical sum is less than number of word blocks
    (b) Missing blocks don't meet the lazy heal threshold
    (c) The sum is less than zero
5. Add the structured stripe and the now altered reverse structured stripe

Only blocks which fulfill all of the requirements are healed.

`_self_horizontal_heal()`:
A horizontal heal is performed in a similar fashion to the vertical heal from above. There is however one difference. It will never heal horizontal stripes which only consist of extra blocks. This is done by checking index against number of vertical word blocks. This implies that it will never encounter empty blocks, to the check for sum less than zero is left out as shown in Figure 4.5

```
def _self_horizontal_heal(self):
    for i, stripe in enumerate(self.array):
        # one stripe at a time, reshape for easy access
        structured_stripe = np.reshape(stripe,
            (self.vertical_length, self.horizontal_length))
        # create reverse matrix where 1 is 0 and 0 is 1
        anti_structured_stripe = 1 - (structured_stripe != 0)
        # array of horizontal sums
        sum_horizontal_stripe = np.sum(structured_stripe,
            axis=1)
        for j, h_sum in enumerate(sum_horizontal_stripe):
            #Perform three checks when not to heal:
            #1. cannot heal is h_sum is less
            #   than horizontal wordblocks
            #2. don't heal horizontally below
            #   vertical wordblocks
            #3. at lest threshold number of blocks
            #   must be missing to heal
            if(h_sum < self.horizontal_wordblocks or
               j > self.vertical_wordblocks - 1 or
               h_sum > (self.horizontal_length
                    - self.lazy_heal_threshold_hor)
               ):
                anti_structured_stripe[j, :] = 0
        self.array[i] = (structured_stripe +
            anti_structured_stripe).ravel()
```

Listing 4.5: Showing how a horizontal heal is performed

4.3.3 Reed Solomon Code class (RS)

Initiation

A Reed Solomon code inherits from the GPC class, and therefore also inherits from the Simulation class. A RS-object must therefore be able to pass both GPC and Simulation initial sanity tests. RS has three mandatory inputs:

`word_blocks` is the number of word blocks per stripe. `extra_blocks` is the number of extra blocks per stripe. `num_of_stripes` and all other optional parameters are the same, except for one. `lazy_heal_threshold` replaces the horizontal and vertical threshold parameter. This is because a Reed Solomon code is a one-dimensional array and can therefore only heal in one direction.

Methods

`_rs_threshold_check()`:
Ensures that the lazy heal threshold must be less than the number of extra blocks

`_update_baf()`:
Same as for GPC, although implemented in a simpler way, because there are no empty blocks to account for.

`_self_horizontal_heal()`:
A simpler implementation of the horizontal heal. Loops over every stripe, and cheks if:

1. Sum of the stripe is more than or equal to the number of word blocks
2. The number of missing blocks in the stripe meets the lazy heal threshold

If both of these are true, all unavailable blocks in the stripe are fully healed.

4.3.4 Replication class (Rep)

The replication erasure code class inherits from RS, an in turn from GPC and Simulation. It only needs to define the number of `replications` and the number of stripes. All other inputs are similar to Reed Solomon. The replication class does not define any new methods.

4.4 Package usage

Below some simple examples of how to use the package are shown. Before using the package, it must be downloaded for Github, and either placed in your Python package path, or in the same folder where you are using the FaiRDy-Py package. You then import it as a normal Python package "`import fairdypy as fp`".

Creating an erasure code object

A GPC-object can be seen created in figure 4.4, and below in figure 4.6 examples for RS and Rep erasure codes can be seen.

```
import fairdypy as fp

# create a reed solomon erasure code
# include all history and verbose print
rs = fp.Reed_solomon(
    word_blocks=32,
    extra_blocks=16,
    lazy_heal_threshold=4,
    num_of_stripes=400,
    verbose=True,
    include_fault_history=True,
    include_history=True
)

# create a replication erasure code
# use storage locations ans hash block method
rep = Replication(
    replications=20,
    num_of_stripes=10000,
    num_of_storage=900,
    storage_fault_mode='hash_block_index'
)
```

Listing 4.6: Examples showing creating a Reed Solomon and a Replication object.

Running a simulation

Examples in figure 4.7 show how to run a simulation by calling the function `loop_simulation(loops)` on the erasure code objects.

```

# run gps for 200 loop, with p_error of 0.2
# stop simulation if baf_limit goes below 0.3
gpc.loop_simulation(
    loops=200,
    p_error=0.2,
    baf_limit = 0.3
)

# create a p_error_array going from 0.2 to 0.3 in 101 steps
p_error_array = [0.2 + (block / 1000)
    for block in range(101)]
# run Reed Solomon with this p_error_array
```

```
rs.loop_simulation(
    loops=101,
    p_error=p_error_array
)


# Create and run an erasure code with
# a custom storage fault array
gpc = fp.Generalized_pyramid_code(
    gpc_definition=(30, 8),
    horizontal_rs=(4, 6),
    vertical_rs=(2, 1),
    num_of_stripes=600,
    storage_fault_mode='custom',
    num_of_storage=16
)
# Create random placement with NumPy
storage_placement = np.random.randint(16,
    size=(gpc.array.shape)) + 1

# set storage_fault_array to defined above
gpc.storage_fault_array = storage_placement
gpc.loop_simulation(
    loops=150,
    p_error=0.2
)
```

Listing 4.7: Examples on running a simulation for defined erasure objects.

Accessing results

Currently only some results are extracted from the data, therefore the possibility to keep every step in the simulation is also available. The main outputs from the simulation are: `baf_history` which is the block availability factor for every step. `blocks_healed_history` contains how many blocks were healed for each step and `blocks_died_history` contains how many blocks became unavailable for each step. For the full block status for each step `fault_history` contains the status for each blocks after each a fault injection and `history` contains the status for each block after each step. The length of all arrays will be the amount of loops in the simulation plus one. This includes the initial status, and then the status after each loop. there results are accessed by using a dot-notation on the erasure code object, as per standard with object oriented programming e.g. `gpc.baf_history`.

4.5 Future possibilities

Some of the future possibilities and ideas which has not been implemented in the FaiRDy-Py module will be discussed here.

4.5.1 Additional erasure codes

Additional erasure codes, or expanding the current ones would let more advanced structures to be analyzed. GPC does not have to be limited by geometrical lines, as it is currently, but could create RS-stripes based on block indexes in any kind of directions and combinations. Erasure codes not discussed in this thesis could also in the future be added to let users compare a vast range of different structures and methods.

4.5.2 Optimization

Currently, the methods for performing heals and storing steps in the simulation are not very well optimized. Especially for GPC, healing is done by restructuring a one-dimensional array as the actual GPC. Then looping over every RS-stripe, one at a time, doing calculations and checks before updating the entire GPC-stripe. This can probably be optimized by using index-slicing, and utilizing NumPy functions to a better degree than now. This would increase the calculation speeds.

Storing all previous steps in the simulation, is currently being done by storing a full copy of the array with every blocks' status at each step. This uses a lot of memory if large erasure codes or many stripes are used. A better method would be to store less data, and then be able to calculate the individual steps. Alternatively a library such as pickle[30] to serialize and store data on disk, to relieve memory usage.

4.5.3 Adding calculations and other quality of live methods

Getting more interesting data and results from simulations would help with illustrating and understanding the evolution of erasure codes over time. Currently this possible by storing the full history for an erasure code object, and analysing the steps programmatically after a simulation has been run. It would however be better to do this during the simulation, so that more results are readily available to the user immediately.

The module does not currently has a method for resetting the erasure code object, and run the code multiple times, to get a range of results possible with different randomized failure patterns.

4.5.4 Illustrations and visualization

Implementing illustrations and visualisation, using Matplotlib, would help a user easily visualize and create plots from simulations the user has run. If multiple runs for the same erasure code and settings could easily be run. Then easily creating statistical plots showing mean values and standard deviation, for example in the shape of a plot with whiskers. Showing the stability of different simulations over time would give a user greater understanding of the simulated system.

Chapter 5

Django web project

The FaiRDy-Py web framework is implemented as a Django project named *web*. It is meant to sit on top of the FaiRDy-Py package, relying on it for the computation of storage system simulations, and itself focus on the storage and display of these simulations. The Django web project is hosted on the University of Stavanger's Unix system, and is available at the URL https://fairdy.ux.uis.no.

5.1 Early Ideas and Specifications

At the beginning of this project the specifications for the web framework component of FaiRDy-Py were general, but included at least the ability to run storage system simulations from a web-based graphical user interface, in addition to storing the results of these simulations in a database to allow analysis and comparison of different erasure coding strategies. These specifications are mainly covered by the *fairdy* Django application. The account management functionalities implemented in the *accounts* Django application were not directly specified, but since the website is intended to be available on the public internet, it followed that there must be some sort of restrictions to prevent misuse of the system.

Our first prototype version of the web framework processed only Reed-Solomon codes with a very small set of parameters, and had no user management functionality. While limited, it provided a proof of concept and a springboard to begin discussions about the user interface and different usage scenarios. This first prototype version included the implementation of the fault-injection and repair modules directly on the Django models for convenience, even though we had already made the decision that the simulation logic should be implemented in a separate module, that could also function

as an independent command line application to conduct simulations.

The second prototype of the web framework integrated the user management functionalities, and was reconfigured with a MySQL database hosted in the Google Cloud to enable hosting of the website in Google's App Engine.

The third prototype, which eventually evolved into our final product, brought back the SQLite database after it became clear we would be able to host the website on a UiS server, and integrated the FaiRDy-Py package with the Django models to achieve our desired architecture.

5.2 System Design and Architecture

The Django web project contains two applications, the first is a user management module providing user registration and authorization functionalities, the second uses the Fairdy-Py Python package to run web-based life cycle simulations of storage systems, with an integrated database and graphical user interface. We chose to separate the web project into two applications in order to keep our individual Django modules a manageable size, and the project structure as organized as possible. Another key advantage Django provides by splitting individual features of a web project into applications, is that the applications are easily moved between Django projects[31], enabling practical reuse of computer code.

The use of SQLite for our database has kept the interface between Django and the database simple. The database exists as a single file which is located inside of the Django project folder. In contrast to the database, the sending of validation emails to users of the website cannot be handled from within the project folder, and requires communication with the UiS Unix System's mail server over the private local network. Aside from the mail server, the only other external dependency for the web project is on Content Distribution Networks (CDN) for obtaining Bootstrap and jQuery JavaScript and CSS modules. We alternatively could have chosen to copy these modules into our project's static files folder, and serve them from our server along with the images we have included, however CDNs are generally reliable and many users will have copies of the files they serve already in the cache of their web browser, preventing duplicate downloads and saving network capacity.

The interface between the web framework and the FaiRDy-Py package, which does the actual simulation computations, is located within the views.py module of the fairdy application in the function *run_module_simulation*. This function accepts the Django Simulation model that is created from the user's input on the form, and instantiates a separate simulation object using the Simulation class of the FaiRDy-Py package, and the parameters stored in the

Django model, which originated in the web form.

5.3 Accounts application

The *accounts* Django application implements features intended to provide a basic level of security and user-friendliness when the entire web application is served to a publicly available URL. By requiring that a user register an account before running simulations, we associate a username with every simulation. Users can then access a list of the previous simulations they have ran, and also filter the broader data set of all previously-ran simulations based on a colleague's username. Since the simulations also need a certain amount of processing power and memory to conduct, requiring users to register and validate before running simulations also adds an extra layer of control against possible misuse of the system.

When the web application is downloaded and run locally with the Django development web server, the accounts application functionality should be disabled in the project's main settings.py file. This enables the user to run unlimited simulations, processed and stored on their local machine, without any administrative formalities.

5.3.1 User Navigation

Registration *(url: /accounts/register)*

This page contains a form to register a new account on the website by providing a valid email address and choosing a username and password. The requirements for the username are defined by Django's default User model, this means they must be 150 characters or fewer, and composed of alphanumeric, _, @, +, ., and - characters.[32] The password requirements are also Django defaults, and the raw password can contain any characters and be of arbitrary length. There are several checks on password strength including: not similar to username, minimum length of nine characters, not in a list of the 20,000 most common passwords, and not entirely numeric.

The form also contains a list of terms and conditions in accordance with the European Union's General Data Protection Regulation[33], this regulation is intended to codify the rights and obligations operators of websites have when targeting and processing the data of European users. These terms and conditions detail what type of personal data the website stores, and that this data will never be shared with any third parties. After clicking submit, if the username and password do not meet the requirements, an error message will

indicate the problem. If the username and password do meet the requirements, and the terms and conditions are agreed to, a new account is created in the database. The user is then automatically logged in, redirected to their profile page, and a verification link containing a unique token is emailed to the provided address. In order to run simulations the user must first validate their email address by following the link sent to their email address.

Login *(url: /accounts/login)*

This simple page allows a user to log in to the website. The form contains two fields, the username and the password, with links to the forgot password and registration pages. After clicking submit, the server validates the username and password, and redirects the user to their profile page. If the username or password are incorrect, a single error message is displayed "Your username and password didn't match. Please try again."

Forgot Password *(url: /accounts/forgot_password)*

From this page a user can reset the password for their account. If no matching user is found, the page will provide an error message indicating this. If a matching user is found, the server will immediately invalidate the password for the account in the database, preventing further login. An email validation link is then sent to the email address registered on the account, and only after following this link is the user able to enter a new password and log in to their account again.

Profile *(url: /accounts/profile)*

Immediately after a user registers or logs in they are redirected to their personal profile page, from here additional links are available for the user to update their username, email, and password. Updating a username requires satisfying Django's built-in requirements for the format of the username, and that all usernames are unique. Changing a password requires correctly entering the current password, and confirming a new password which also meets Django's built-in requirements. To update their email address, a user must complete the email validation process again with the new email, and will be restricted from running any new simulations until the new email address associated with the account has been validated. The profile page also contains a progress bar labelled "Remaining Credits", described further in the next section, to track the user's use of server resources.

While we have deliberately limited the amount of personal data stored about users, details such as email address, and potentially also username

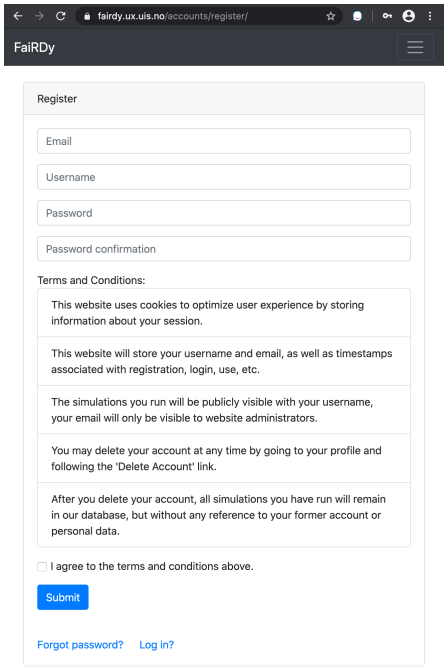


Figure 5.1: registration form

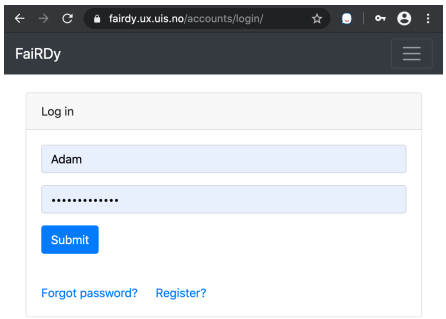


Figure 5.2: login form

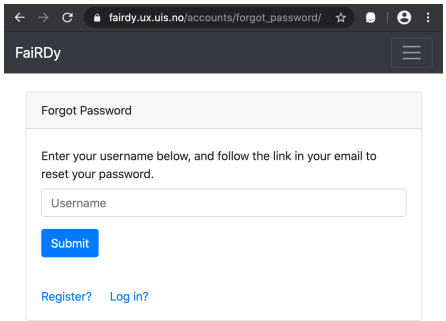


Figure 5.3: forgot password form

and timestamps, fall under the GDPR's definition of "Personal Data" and as such must be processed according to certain principles. One of these key principles set out by the GDPR is the "Right to be Forgotten", and to this end we have implemented a delete account feature to enable a user to delete all personal data stored in the project's database, and remove their username from any simulations the user may have run.

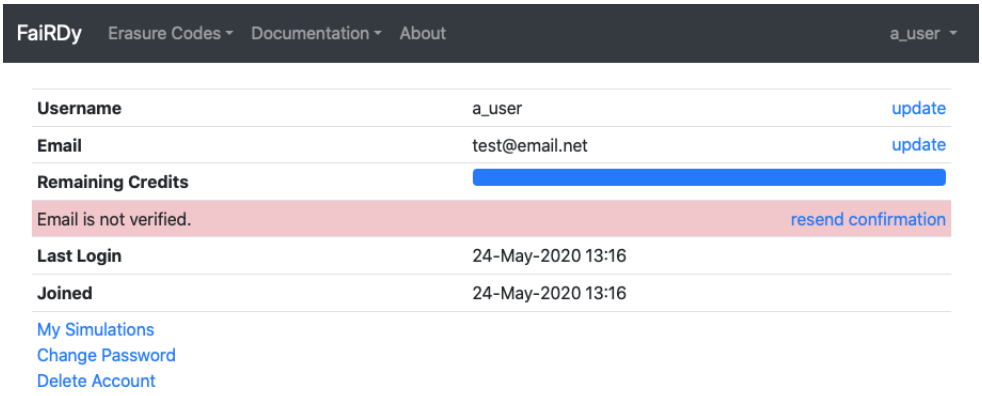


Figure 5.4: The profile page of a user who has not yet validated their email address.

Block Cycle Limit

While FaiRDy is a lightweight framework for characterizing the reliability of storage systems over their lifecycle, it still requires a certain amount of compute and memory to process the simulations. For this reason and because the server is hosted on hardware at the University of Stavanger, all accounts have a Block Cycle Limit (BCL) defined, which limits the amount of simulations within a month. Some privileged accounts have their BCL set to a value of -1, this indicates the account is allowed to run unlimited simulations. The BCL is calculated as the size of the storage system (in blocks) multiplied by the number of cycles. New accounts have a default BCL of 500 million, and remaining credits are shown with a progress bar on the profile page under the heading "Remaining Credits". Considering that most systems reveal few additional insights after 100 cycles, the default limit allows the running of 100 simulations with up to 50,000 total blocks in each system. This should be sufficient for most cases to casually explore the effects of erasure code parameters on storage system life cycles.

Change username *(url: /accounts/change/username)*

This is a simple form accessible from the profile page, it enables a user to change their username so long as the new username they choose meets the requirements defined in the registration section. The only requirement is that the user is currently logged in.

Change Email *(url: /accounts/change/email)*

Another simple form accessible from the profile page, enabling a user to update the email address associated with the account. After successfully submitting this form, the user must validate their new email address according to the procedure in the registration section before they are again able to run simulations.

Change Password *(url: /accounts/change_password)*

Allows a currently logged in user to change their password, assuming they remember their current password. Since the user is required to provide their current password, no email validation of the account is required as in the Forgot Password section.

5.3.2 Implementation Details

The accounts application implements and extends features provided by Django's built-in authentication system. To activate the built-in authentication system the *django.contrib.auth* package must be included in the *INSTALLED_APPS* setting in the Django project's settings.py file.

FairdyUser, a Django Model

Because the fields contained on the *User* model provided by django.contrib.auth were not sufficient for the needs of this project, we implemented an additional *FairdyUser* model to store the custom fields we need for a user of this website, in line with Django recommendations.[34] The FairdyUser model has a one-to-one relationship with the Django standard User model, and as such, a table will be created in the database for each model, *accounts_fairdyuser* and *auth_user* respectively.

The FairdyUser model stores fields to track whether the user's email is validated or their password has been reset, a token and its expiration date, as well as the user's block cycle limit. In addition to these fields, the model

implements a number of functions which operate on the data contained in the fields.

- **get_bcl_millions(self)**
  Returns the user's block cycle limit, in millions.
- **set_bcl_millions(self, new_limit)**
  Sets the user's block cycle limit, in millions.
- **get_available_bcc(self)**
  Returns the user's remaining block cycle credits, calculated by summing the block cycle credits used on each of the user's simulations ran over the last month, and subtracting this from their limit.
- **can_run_simulation(self, web_sim)**
  Returns true if the user has enough block cycle credits to run the Simulation object passed in as a parameter.
- **get_remaining_bcc_percent(self)**
  Returns a number between 0 and 100 representing the percent of the user's block cycle credits which remain, used for the progress bar on the profile page.
- **set_token(self)** Generates and stores a new token and expiration date for the user.
- **token_is_expired(self)** Returns true if the token's expiration date has passed.
- **send_email(self, subject, html)**
  Helper function used by the other two email functions to call on Django's email backend.
- **send_reset_email(self)**
  Sends an email to the user with a link that contains a token to reset the account password.
- **send_validation_email(self)**
  Sends an email to the user with a link that contains a token to validate the email address.

Just as the FairdyUser inherits from Django's internal User model, the Django form class we define for our FairdyUser model inherits from Django's internal UserCreationForm class, and it is only necessary to define form behavior for

fields not already included there. Since the standard UserCreationForm class already had fields for username and password, it was only necessary to add fields for an email address and agreement with the terms and conditions to our form.

Views

It is the view functions which provide the behavior described in the User Navigation section, are all location in the accounts application's views.py module. As an example the function enabling the registration of accounts on the URL /acounts/register is shown below. If the originating request is an HTTP get it will hop to line 20, and instantiate the form, before Django's templating engine renders it together with register.html and returns it to the user. If instead, the originating request is an HTTP post, and all criteria for form validation are met, the new user is saved to the database and logged in. The one-to-one FairdyUser model is then instantiated, saved to the database, its function called to send a validation email, and the new user is redirected to to their profile page.

```
1  # REGISTER
2  # register a new account, and send a confirmation email
3  def register(request):
4      if request.method == 'POST':
5          form = FairdyRegistrationForm(request.POST)
6          if form.is_valid():
7              username = form.save()
8              password = form.cleaned_data.get('password1')
9              user = authenticate(
10                 username=username,
11                 password=password
12             )
13             login(request, user)
14             fairdy_user = FairdyUser.objects
15                 .create_fairdy_user(user)
16             fairdy_user.send_validation_email()
17             messages.success(request, 'Thank you for
    registering an account on fairdy.no! Follow the link in
    your email to complete the process.')
18             return redirect('accounts:profile')
19     else:
20         form = FairdyRegistrationForm()
21     return render(request, 'register.html', {'form': form})
```

Listing 5.1: View function for the registration of a new user, it is mapped to the /accounts/register URL in the accounts application's urls.py

Email verification

Account validation emails and emails to reset passwords are processed and sent using Django's *django.core.mail* module. This module provides a function, *send_mail(subject, message, from_email, recipient_list)* which sends email messages, all four parameters shown are required, and additional optional parameters exist. Before using the send_mail() function it is necessary to specify the email server Django will use to actually send the emails in the project's settings.py file. Django enables the use of many types of email servers and connections between itself and the email server, with all of these settings defined in the settings.py module for the project. In the case of FaiRDy-Py we were lucky that a mail server was available for our use on the UiS Unix system, so we simply had to point to this server. Because both the server hosting the web framework and the email server are on the same private network, encryption of the emails is not critical as they are transmitted from Django to the mail server.

For the content of the emails we used an open-source template we found in this GitHub repository: https://github.com/leemunroe/responsive-html-email-template
With some minor modifications, this template worked well to create attractive HTML emails for FaiRDy.

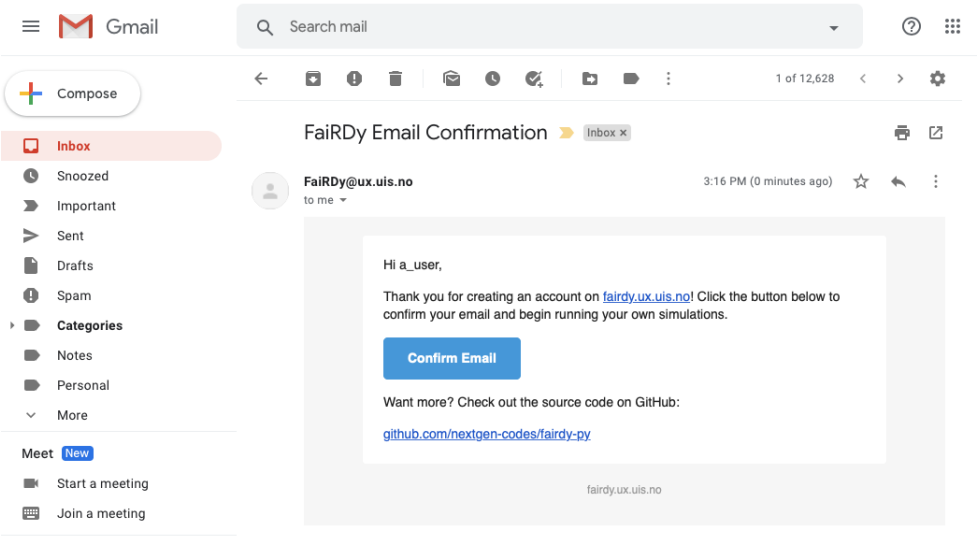


Figure 5.5: An email validation message from the FaiRDy web framework, as displayed in Gmail

```
# SMTP server used to send email validation messages
EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
EMAIL_HOST = 'broremann.ux.uis.no'
```

Listing 5.2: The two lines in settings.py required to connect the Django app the mail server on UiS' Unix system.

Validation tokens

For security purposes, it is necessary to ensure that the links FaiRDy sends to users for confirming their email or resetting their password cannot easily be guessed, allowing validation to be bypassed. For this reason we generate random strings of URL-friendly characters that are so long that the chances they could ever be correctly guessed approach zero. For this purpose we use the Python language's built in *secrets* module to generate a unique token 64 characters in length which is included in the validation link. The advantage of using Pythons's secrets module is that it generates the tokens randomly in a way which is impossible to predict, while the output of other modules meant to produce random strings, but not intended for security purposes, can be predicted in certain circumstances.[35]

Figure 5.6: The validation URL in address bar of a web browser, showing the secure token partially at the end

5.4 FaiRDy application

The *fairdy* Django application handles everything related to running the life cycle simulations of storage systems which form the core of this project. The actual processing of the simulations is carried out by the Fairdy-Py package, the purpose of the fairdy Django application is to call on this package to run the simulation, store the results in the database, and finally illustrate the system's evolution by displaying a plot of the system's block availability factor by cycle.

5.4.1 User Navigation

Index *(url: / & /fairdy/index & /fairdy/_code_)*

This is the page from which the user can view a list of and browse through previously ran simulations. Each simulation record also contains a checkbox, which once checked enables the "Download" and "Compare" buttons. The boxes which have check marks are stored in the URL query string, preserving the user's selections even as the browse between multiple pages of simulations. This page can also be filtered by username, to show the simulations ran by a single user, by including the username in the query parameter. On the other hand, filtering this page by erasure code type does not make use of the URL query string but rather the URL itself. For example, pages filtering the list for the erasure codes we have implemented in FaiRDY-Py are available on these routes:

- /fairdy/replication
- /fairdy/reed_solomon
- /fairdy/generalized_pyramid

Visualization *(url: /fairdy/visualization)*

This is the main route from which a user can view and compare the results of simulations, and one or more simulation ids must be included in the URL's query parameters so the server knows which simulations to plot and display. The page includes a graph showing the evolution of the simulated storage system(s) in terms of the block availability factor after each cycle of evolution is completed, and also a listing of the plotted systems in the same format as the index page.

Run *(url: /fairdy/run)*

This is the main form to enter the desired parameters to create a new storage system simulation. In the first section of the form the stripe that will be used to build the storage system is defined. Depending on the erasure code which is selected in the drop down box, certain fields will either be activated, or disabled for use input and calculated automatically by JavaScript on the page. This assists the user by limiting possible errors in parameter entry and reducing the amount of fields to consider.

CSV *(url: /fairdy/CSV)*

From this route a user can download a comma separated values (CSV) file containing all the details of one or more simulation, so that they may plot or analyze the simulation results in software of their choice. Users will generally be directed to this route by checking off simulations on the index page, and then clicking the "CSV" button. For each cycle, the block availability factor, number of blocks faulted, and number of blocks healed are recorded.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	id	code_typ	n_vali	k_val	m_vali	num_stri	num_cyc	p_error	laz	sto	nu	k_horiz	m_hori	k_vert	m_ver	lazy_h	num_re	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4
2	7	GP	32	8	24	100	100	0.4	1		#	4	4	2	2	1	2	1	0.9319	0.8903	0.8697	0.8538
3	faulted_blocks																	0	1293	1207	1139	1130
4	healed_blocks																	0	1075	1074	1073	1079
5	6	GP	32	8	24	100	100	0.3	1		#	4	4	2	2	1	2	1	0.9631	0.9609	0.9547	0.96
6	faulted_blocks																	0	992	910	920	878
7	healed_blocks																	0	874	903	900	895
8																						

Figure 5.7: CSV output from the web framework displayed in Microsoft Excel.

5.4.2 Implementation Details

Listing Simulations

The list of simulations which appears on both the index page and on the visualization page uses the same HTML template file, which reuses the code and means it only needs to be updated in a single place if a new parameter is added or the appearence is updated. The HTML template file contains a single HTML table element, which in-turn contains subordinate table elements in some of the fields to enable multiple lines of data in the same cell. The thumbnail used to give a preview of the BAF curve, is generated in a similar way to the graph on the visualize page, however in this case it is generated by a function on the Simulation model we have defined in Django.

```
    def get_thumbnail(self):
        max_cycle_count = self.num_cycles + 1
        y = self.cycle_set.values_list(
            'block_availability_factor',
            flat=True
        )
        if len(y) < max_cycle_count:
            y = list(y) + ([None]*(max_cycle_count-len(y)))
        x = range(0, max_cycle_count)
        pyplot.plot(x, y, linewidth=7)
        pyplot.xlim(0, max_cycle_count - 1)
        pyplot.ylim(0, 1.1)
```

```
        buf = BytesIO()
        pyplot.savefig(buf, format='png', dpi=20)
        pyplot.close()
        return base64.b64encode(buf.getvalue())
            .decode('utf-8')
            .replace('\n', '')
```

Listing 5.3: The function located on the Django Simulation model which returns a thumbnail of the simulation's BAF curve

Plotting Simulations

Simulations are plotted in the visualize function of the fairdy application's views.py module. We have chosen to avoid storage of any plot images and generate them on-the-fly, the visualize function coverts the plot to a byte stream representing a png image file, this is then inserted into the HTML by the templating engine.

```
def visualize(request):
    if 'id' not in request.GET:
        raise Http404()
    ids = request.GET.getlist('id')
    sims = Simulation.objects
        .filter(pk__in=ids)
        .order_by('-id')
    # increment the view count on each of the simulations
    for sim in sims:
        sim.increment_view_count()
    data = {'sims': sims}
    # get the highest num_cycles any of the sims have and add
    # an additional one to compensate for cycle 0 with baf 1
    max_cycle_count = sims.aggregate(
        Max('num_cycles')
    )['num_cycles__max'] + 1
    labels = []
    for sim in sims:
        y = sim.cycle_set.values_list(
            'block_availability_factor',
            flat=True
        )
        # if simulation goes to 0 before completing all
        # cycles, add None(s) to baf list so graph goes
        # to full cycle count
        if len(y) < max_cycle_count:
            y = list(y) + ([None]*(max_cycle_count - len(y)))
        x = range(max_cycle_count)
        pyplot.plot(x, y)
        labels.append('{}, sim{}'.format(str(sim), sim.id))
```

```
    pyplot.legend(labels)
    pyplot.xlim(0, max_cycle_count - 1)
    pyplot.ylim(0, 1.1)
    pyplot.xlabel('time (in cycles)')
    pyplot.ylabel('Block Availability Factor')
    pyplot.title('Block Availability Factor vs. time')
    buf = BytesIO()
    pyplot.savefig(buf, format='png', dpi=150)
    pyplot.close()
    data['graph'] = base64.b64encode(buf.getvalue())
        .decode('utf-8')
        .replace('\n', '')

    return render(request, 'visualize.html', data)
```

Listing 5.4: The visualize function in the fairdy application's views.py module. First the simulations matching the ids in the query string are fetched from the database, their view count is updated, and the highest cycle count of any of the simulations is computed for the graph's x (horizontal) dimension. Next we loop through the simulations from the database and plot them using the **pyplot** module from the Matplotlib Python library. Finally labels and other parameters are added to the plot, and the plot is encoded into a byte stream to be inserted into the HTML page by the templating engine.

5.5 Database

While we also deployed the database for this project using MySQL in the Google Cloud, we ultimately decided to use SQLite for the reasons outlined in Deployment section. Because of Django's powerful object relational mapper we have had very little direct interaction with the database throughout the course of this project. The creation and updating of database tables, based on the classes we define the the applications' models.py files, is handled by Django's manage.py utility module. Running these three commands from within the Django project folder creates the SQL scripts to either create or update the database tables, and applies them to the instance of the database defined in the project's settings.py module. Additionally, the insertion, modification, and deletion of data in the database has occurred almost exclusively via Django, and without any SQL statements.

In the FaiRDy-Py web framework, database data is only created, modified, or deleted based on the submission of a form by the website user. When a user submits a form, Django first validates the input and if successful instantiates an object of the Django model the form represents. All objects inheriting from the Django models.Model class have a series of functions

which can then be called to send the new or updated record to the database. When running simulations in FaiRDy-Py, the system evolution is first computed by the FaiRDy package, the cycle results are then copied to the Django model object, and the *save()* function is finally called to send the records to the database.

```
$ python manage.py makemigrations fairdy
$ python manage.py makemigrations accounts
$ python manage.py migrate
```

Listing 5.5: The commands used to invoke Django's object relational mapper, and update the database based on the definitions in the application's models.py module.

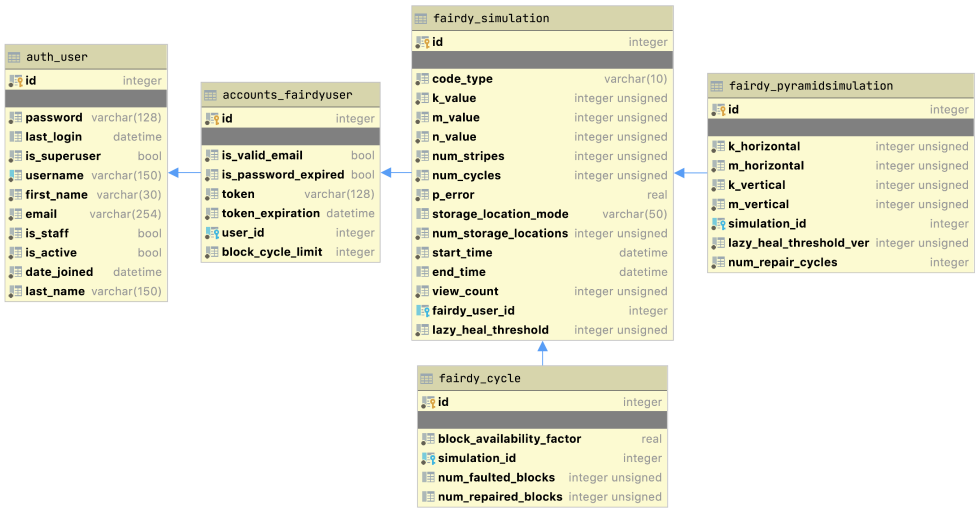


Figure 5.8: A diagram of the database tables for the FaiRDy web framework. The **auth_user** table is created from Django's built in User model and, **accounts_fairdyuser** is created from the accounts application's FairdyUser model. The **fairdy_simulation** table stores the parameters for all simulations, while the **fairdy_pyramidsimulation** table has a one-to-one relationship with the fairdy_simulation table and contains the additional fields required by Generalized Pyramid Simulations. Finally, the **fairdy_cycle** table holds data for the individual cycles that each simulation is evolved through.

5.6 Testing

Django includes tools for automated testing of the modules you have written with certain pre-configured scenarios in order to verify that your code is working as expected, and often, still working as expected after making changes in a completely different area of your code. Unfortunately, because of time constraints with this project we were not able to develop as many automated test as we would have liked, but several test functions were created for the accounts module, including the following code which simulates the registration of a user account on the website, checks a special virtual email inbox for the validation link, follows the link to validate, and then logs out of the account.

```
class RegistrationTest(TestCase):
    def setUp(self):
        c = Client()
        response = c.post(
            '/accounts/register/',
            {
                'email': 'test@fairdy.no',
                'username': 'testuser',
                'password1': 'kjH45636&&',
                'password2': 'kjH45636&&'
            }
        )
        self.assertEqual(response.status_code, 200)
        user = User.objects.get(username='testuser')
        fairdy_user = FairdyUser.objects.get(user_id=user.id)

        # Test that the new user is now in the temp database
        # used for testing
        self.assertIsNotNone(fairdy_user)

        # Test that the test email mailbox now
        # contains 1 message
        self.assertEqual(len(mail.outbox), 1)

        # Test that the email contains the token
        # for the fairdy_user
        token = fairdy_user.token
        self.assertIn(token, mail.outbox[0].body)

        # Test validation
        response = c.get(
            '/accounts/validate/{}/'.format(token)
        )
        self.assertEqual(response.status_code, 302)
```

```
35
36          # Test validation only works once
37          response = c.get(
38              '/accounts/validate/{}/'.format(
39                  fairdy_user.token
40              )
41          )
42          self.assertEqual(response.status_code, 404)
43
44          # Test log out
45          response = c.get('/accounts/logout/')
46          self.assertURLEqual(response.url, '/accounts/login/')
47          self.assertURLEqual(
48              c.get('/accounts/profile/').url,
49              '/accounts/login/?next=/accounts/profile/'
50          )
```

Listing 5.6: Test class checking user registration and email validation.

5.7 Deployment

Throughout the course of development of the FaiRDy-Py web framework several deployment methods were used before ending with deployment on the *gorina2* server in the UiS Unix system.

5.7.1 Django Development Server

The Django Development Server is the most basic way to deploy a Django project, it is however not suitable for use in a production setting on the public internet.[25] It is activated for the project using *manage.py* with the *runserver* command.

```
1 $ python manage.py runserver
2     Watching for file changes with StatReloader
3     Performing system checks...
4
5     System check identified no issues (0 silenced).
6     May 22, 2020 - 12:06:57
7     Django version 3.0.6, using settings 'web.settings'
8     Starting development server at http://127.0.0.1:8000/
9     Quit the server with CONTROL-C.
```

Listing 5.7: Invoking Django's development server on the command line.

After issuing the runserver command with no parameters, the Django project is made available as a web site to all web browsers on the local ma-

chine by entering 127.0.0.1:8000 into the address bar, it will not be available for other computers on the network.

```
$ python manage.py runserver 0.0.0.0:8010
```

Listing 5.8: Invoking Django's development server on the command line and specifying an IP address/port combination.

By including the IP address 0.0.0.0, the Django Development Server will make the website available to all computers on the local network. The IP address 0.0.0.0 indicates the web site should be available to other devices on the network by entering the IP address of the device running the development server into the address bar of a web browser. The port, here 8010, can be left blank to use the HTTP default, 80, here we have used 8010 because this is the port that was assigned to our project for use on the UiS Unix system.

5.7.2 On a Home Network

In the process of development we have have hosted the website using the Django Development Server on a private home network by entering 0.0.0.0 for the IP address so it is reachable by other devices on the local network. This was helpful for checking the website appearance from the web browser on a smart phone or ran on a different desktop operating system. In addition to accessing the website from within the home network, we were also able to successfully add a port-forwarding rule to the home router, such that other team members were able to view the website from their homes by entering the IP address of the home where the development server is running.

5.7.3 On Google Cloud

Before we were sure we would be able to host the FaiRDy-Py web framework on the Unix system at the University of Stavanger, we investigated and successfully tested deploying the website on Google Cloud's App Engine. The App Engine basically works by executing the Django project inside of a very limited virtual server called a container. [36] Publishing the project itself to an App Engine container in the Google Cloud was relatively straightforward, the main challenge with this was the migration of the database, since Google App Engine containers do not support the use of SQLite databases with Django projects. Because of this we were required to migrate our database to a separate Google Cloud product, Cloud SQL, which was an instance of a MySQL database hosted in the cloud.

In the end we were grateful that we were able to host the website on a server at UiS instead of having to rely on the Google Cloud. There were

several reasons for this, among them that the website seemed to run far slower when hosted on the Google Cloud than when hosted on our local machine, and also the expense appeared to be excessive. We were granted a trial account with a little over NOK 1,000 in free credits, so no payment was necessary, but these trial credits were spent very quickly. It appeared the majority of the costs were coming from the Cloud SQL data storage, and we were able to reduce these costs partially by downgrading the technical specifications of the SQL instance, but they were still substantial and this seemed to make the performance even worse. Google Cloud is an extremely complex and varied product, about which entire masters' theses could be written, so it's likely we overlooked some trivial adjustments that would have both increased performance and reduced cost.

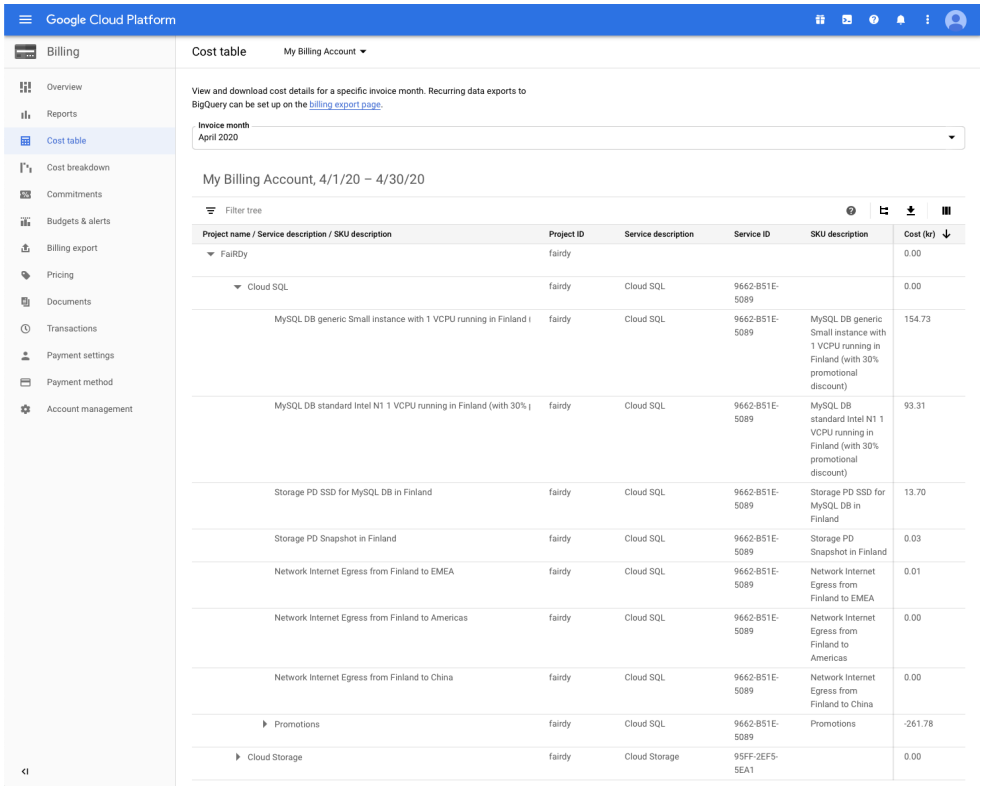


Figure 5.9: Screenshot from the Google Cloud Platform dashboard showing charges of NOK 261.78 for the month of April.

5.7.4 On the UiS Unix System

On the UiS Unix system we were assigned a file server directory for the FaiRDy project, accessible by all team members on the path */home/prosjekt/fairdy/*. We were given access to server *gorina2* to run the web server executable responsible for hosting the website, and port 8010 on this server was mapped to the URL https://fairdy.ux.uis.no for us by Theodor Ivesdal, administrator of the UiS Unix system.

Python Virtual Environment

The main FaiRDy project folder located at /home/prosjekt/fairdy/ contains a sub-folder *venv* which contains the Python virtual environment from which we run the web server. The purpose of this virtual environment is to provide a self-contained Python execution environment containing only the exact versions of Python packages and frameworks needed for this project. If we had installed Django, NumPy, and Matplotlib into the server's base Python installation instead, and executed our web server with this environment, somebody else could come along later, upgrade these packages, and render our project unusable.[37]

```
user@gorina2:~$ cd /home/prosjekt/fairdy
user@gorina2:/home/prosjekt/fairdy$ source venv/bin/activate
(venv) user@gorina2:/home/prosjekt/fairdy$ cd fairdy-py/web
(venv) .../web$ python manage.py runserver 0.0.0.0:8010
    Watching for file changes with StatReloader
    Performing system checks...
    System check identified no issues (0 silenced).
    May 22, 2020 - 13:29:44
    Django version 3.0.5, using settings 'web.settings'
    Starting development server at http://0.0.0.0:8010/
    Quit the server with CONTROL-C.
```

Listing 5.9: Activating the virtual environment for the FaiRDy project and running the Django development server after connecting to gorina2.

GitHub Repository

The second sub-folder in the main FaiRDy project folder, *fairdy-py*, contains the GitHub repository for the FaiRDy-Py project.

```
user@gorina2:~$ cd /home/prosjekt/fairdy/fairdy-py
user@gorina2:/home/prosjekt/fairdy/fairdy-py$ git pull
```

Listing 5.10: Commands to pull changes committed to the web repository down to the local repository on gorina2.

If changes are made to the Django models that would also result in changes to the database, the following commands are used on gorina2 to update the database, which is excluded from the GitHub repository.

```
$ python manage.py makemigrations fairdy
$ python manage.py migrate
```

5.8 Future Possibilities

Many possibilities exist to extend and improve the FaiRDy-Py web framework component. Here we discuss a couple issues we wish we had more time to address.

5.8.1 Simulation Listing Page

We discussed adding a filter functionality to the page which lists simulations, such that it would be possible to filter for simulations with a certain erasure code, stripe dimension, error probability, etc., but as this is not absolutely essential to the core functionality, this was not highly prioritized, and was not able to be completed in the allotted time for the project. Addition of this functionality would not be overly complex, we had envisioned using the query parameters, which form part of the URL, to store this information. This would function in a very similar way to the already-implemented filtering of this list by username. It would be necessary to add fields for the filter input to the HTML template, and accompanying JavaScipt code would need to be updated to retrieve the input from these fields and append it to the URL before submission to the server. On the server side, the index function in the fairdy application's views.py file would need to be updated, again following the pattern already used for the username, these additional parameters would need to be parsed out of the query string, and applied to the database query in the function.

For reference, URL's query string is contained after the question mark (?) character, here it contains a username, 'john_doe':
https://fairdy.ux.uis.no/fairdy/index/?username=john_doe

```
sim_list = Simulation.objects.all()
# filter the sim list from database by username if present in
    the query string
if 'username' in request.GET:
    fairdy_user = FairdyUser.objects.get(
```

```
5         user__username=request.GET['username']
6     )
7     sim_list = sim_list.filter(fairdy_user=fairdy_user)
```

Listing 5.11: Section from the index function of the fairdy Django application's views.py file, which filters for a username in the request's query parameters. To filter by additional parameters on the server side, it is only necessary to repeat the if statement on lines 3, 4, and 5 for each additional parameter.

5.8.2 Stripe Parameter Validation in Django Form

We made the decision to leave the majority of the logic concerning stripe shape validation, most notably Generalized Pyramid stripes, to the FaiRDy-Py package, and independent of the web framework component. This was meant to prevent the repetition of code, and to have one definitive source for the validation, but it also introduced an unintended problem where the entire form is reset and emptied when an error message flows up from the FaiRDy-Py package, through the Django view, and is displayed to a user on the website. The cause of this problem is that the Django form must be successfully validated in order to access the simulation parameters it contains, which are in-turn required to call the FaiRDy-Py package and create a simulation, but once a Django form is successfully validated its contents are erased. Normally in Django, when a form fails validation its contents are retained and passed back to the user, so that it's not necessary to also refill other fields which where valid. Unfortunately in this case our architecture requires us to first consider a form containing erroneous information as valid, before we are even able to determine this is false.

The best solution to this problem is not immediately obvious to the authors, but all possibilities will likely include implementing a custom validation method on the Django form. Several ways to do this would be worth investigating: finding a way to import validation functions from the FaiRDy-Py package to the Django form's custom validation function, instantiating the module simulation object within the form's validation method, or simply repeating the rules from the module in the validation function.

5.8.3 Async and Busy Indicator

When running storage system simulations with large numbers of blocks in the system, generally those with more than a few thousand blocks total, processing the simulation can take a considerable amount of time, sometimes

minutes. While this is somewhat apparent to desktop users of some web browsers in the form of a spinning indicator or similar, this is not always easy to notice, and sometimes doesn't exist at all, especially on mobile platforms. With this in mind, it is desirable to add some sort of obvious indicator so the user knows their simulation is in process, and does not push the submit button multiple times and run the simulation several times. Additionally, we have not investigated the server's behavior when in the process of running a long simulation for one user while another simultaneously makes another request. If performance is an issue in this case, Django is in the process of adding support for asynchronous view functions, and it may be worth investigating if this would improve the server's response time for all users. It is also possible large improvements in these parameters could be made by ensuring proper configuration of HTTP server and load balancer.

5.8.4 Random Seed

As FaiRDy-Py stands, no random seed is set, either in the module or the web framework, and Python receives the seed from the OS of the machine running FaiRDy-Py. For larger simulations with total system sizes exceeding several thousand blocks, the variation between simulations ran with identical parameters tends to be minimal, however for systems with fewer blocks the variations can be more noticeable. To make all simulations exactly reproducible, including ones with systems containing few blocks, it should be possible to set the a random seed before running a simulation on both the web form and with the FaiRDy-Py package.

5.8.5 Additional Storage System Parameters

There are two storage system options that are implemented in the FaiRDy-Py package, but not the web framework. They are passing in a vector of error probabilities, and Generalized Pyramid Stripes which are "partially overlapping". These are not critical options, but should ideally be added to the web framework.

5.8.6 Ordering Visualizations

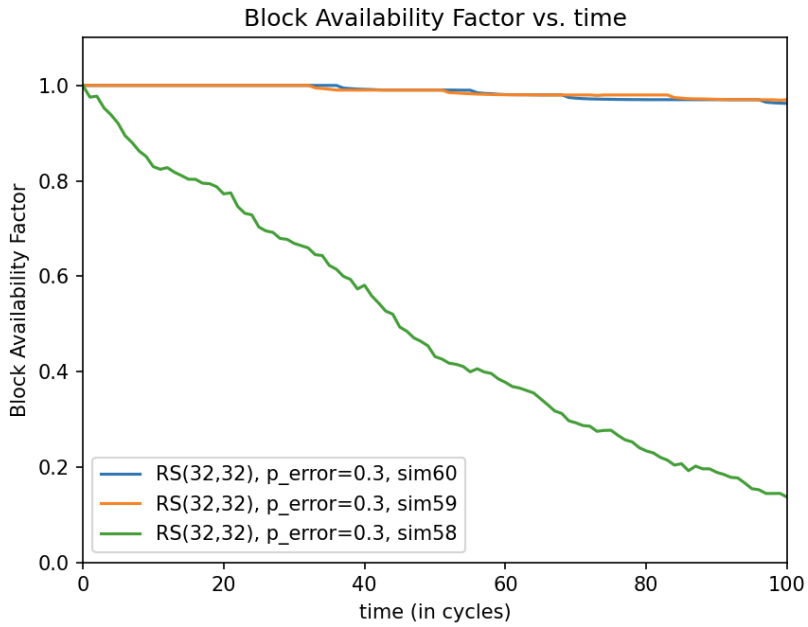
It would be desirable to be able to set some basic parameters for the comparison graph. For example to include other parameters such as number of repair cycles, which could be the focus of inquiry, in the graph's legend. The ability to arrange the order in which simulations appear in graphs would also be desirable, currently the most recently ran is always on top.

Chapter 6

Results

In this chapter we will present some observations about the storage system life cycles that we have noticed while using the software we have developed for this project. This chapter also serves as a demonstration of what has been achieved during the project, both the underlying python module, and the website interface and display.

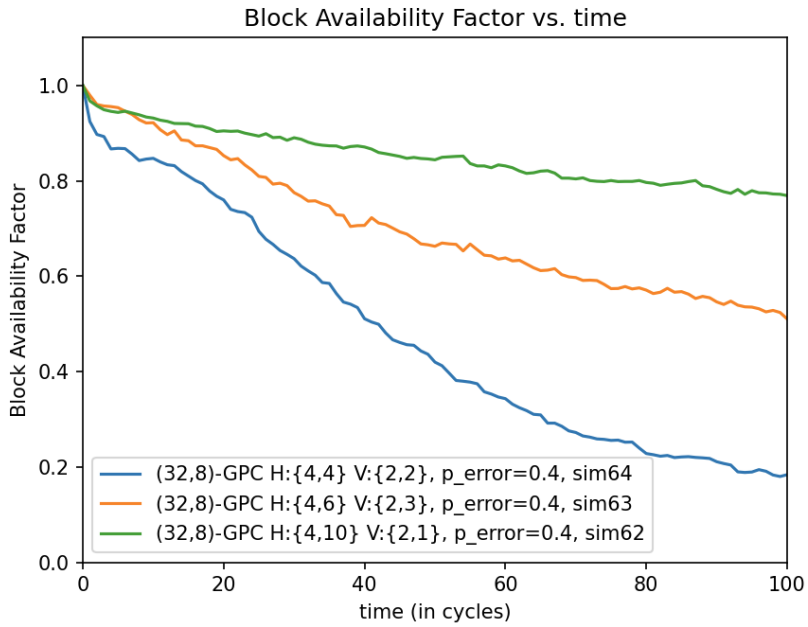
6.1 Lazy Heal



storage system	curve	blocks in stripe	system dimension	options	details
RS(32,32), p_error=0.3 view sim60		32 data 32 redundancy 64 total	100 cycles 100 stripes 6400 blocks		Adam 24-May-2020 2 views
RS(32,32), p_error=0.3 view sim59		32 data 32 redundancy 64 total	100 cycles 100 stripes 6400 blocks	**lazy heal threshold (horizontal)** 8	Adam 24-May-2020 2 views
RS(32,32), p_error=0.3 view sim58		32 data 32 redundancy 64 total	100 cycles 100 stripes 6400 blocks	**lazy heal threshold (horizontal)** 16	Adam 24-May-2020 2 views

Figure 6.1: In this comparison we can see the results of applying the lazy heal parameter to storage systems consisting of 100 stripes of RS(32,32) and p_error=0.3. In this particular configuration, with this probability of error, it is clear that a lazy heal threshold of 8 has little effect on the system's evolution, potentially sparing bandwidth and resources in the storage system. On the other hand, increasing the lazy heal threshold to 16 would be seriously detrimental to the survival of the system over the long term.

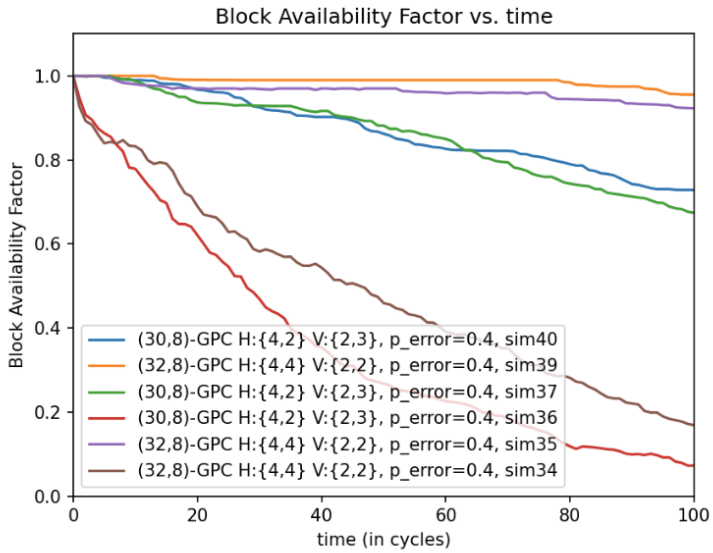
6.2 GPC Stripe Shape



storage system	curve	blocks in stripe	system dimension	options	details
(32,8)-GPC H:{4,4} V:{2,2}, p_error=0.4 view sim64		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 2	Adam 24-May-2020 2 views
(32,8)-GPC H:{4,6} V:{2,3}, p_error=0.4 view sim63		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 2	Adam 24-May-2020 2 views
(32,8)-GPC H:{4,10} V:{2,1}, p_error=0.4 view sim62		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 2	Adam 24-May-2020 2 views

Figure 6.2: In this comparison we can see the results of different (32,8)GPC stripe shapes, they have the exact same number of redundancy and data blocks in the system, and p_error=0.4 for all, but it is clear the performance of the (32,8)GPC H:{4,10} V:{2,1} stripe is superior to the other shapes for a system with these parameters.

6.3 Repair Cycles for GPC Overlapping Stripes



storage system	curve	blocks in stripe	system dimension	options	details
(30,8)-GPC H:{4,2} V:{2,3}, p_error=0.4 view sim40		8 data 22 redundancy 30 total	100 cycles 100 stripes 3000 blocks	**repair cycles** 3	Adam 24-May-2020 3 views
(32,8)-GPC H:{4,4} V:{2,2}, p_error=0.4 view sim39		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 3	Adam 24-May-2020 3 views
(30,8)-GPC H:{4,2} V:{2,3}, p_error=0.4 view sim37		8 data 22 redundancy 30 total	100 cycles 100 stripes 3000 blocks	**repair cycles** 2	Adam 24-May-2020 4 views
(30,8)-GPC H:{4,2} V:{2,3}, p_error=0.4 view sim36		8 data 22 redundancy 30 total	100 cycles 100 stripes 3000 blocks	**repair cycles** 1	Adam 24-May-2020 4 views
(32,8)-GPC H:{4,4} V:{2,2}, p_error=0.4 view sim35		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 2	Adam 24-May-2020 4 views
(32,8)-GPC H:{4,4} V:{2,2}, p_error=0.4 view sim34		8 data 24 redundancy 32 total	100 cycles 100 stripes 3200 blocks	**repair cycles** 1	Adam 24-May-2020 3 views

Figure 6.3: In this comparison we can see the results of two different overlapping Generalize Pyramid stripe shapes, (30,8)GPC H:4,2 V:2,3 and (32,8)-GPC H:4,4 V:2,2, 100 stripes and p_error=0.4 for all. The main focus here is the number of repair cycles, with both 2 and 3 cycles holding the system near full survival

Chapter 7

Conclusion

At the end of the project, a Python implementation of the FaiRDy module was achieved, as well as a web interface for users to run simulations and visualize simulations.

7.1 Python FaiRDy

The FaiRDy module previously created in Wolfram Language by Verónica Estrada Galiñanes[1], [2] was successfully implemented in Python. The package was realized using the NumPy library. The module can run erasure code simulations for Replication, Reed Solomon, and Generalized Pyramid Codes. This while letting the user choose among multiple customization setting: How many loops to run the simulation, the chance for blocks or storage locations to fail (even with chance changing between steps), storage location distribution, lazy heal thresholds, and number of heal cycles for Generalized Pyramid Codes. Some of the decisions and choices were done more in the moment, than out of actual planning

The general hierarchical structure of the erasure code package, has its benefits and drawbacks. The main reason for structuring it this way, was to avoid having to write code and define attributes for classes multiple times. The structure has the benefit of defining sanity checks, fault injection, and the simulation function at only one level of the hierarchy. Inherited attributes and functions also automatically have the same name across objects. Making changes and correcting errors in the code is therefore only needed to do at one location. One of the main drawbacks of this choice is making sure the functions work properly across inheritance, which can in some situations make it a bit harder to troubleshoot problems. If we were to start over and implement this again, a better plan should have been created to properly

map out features of erasure codes. This would have earlier lead us on the right path and the two rewrites that were done, could have been avoided.

All in all the package produced results which resemble Verónica Estrada Galiñanes' unpublished results. FaiRDy-Py was, as planned, published on Github allowing it to be spread to users who are interested in seeing the effect erasure code structures have on storage security. Hopefully it will encourage users to do their own tests, and perhaps create their own version, or even contribute to the FaiRDy project. The end goal for the Python package was met, even though there is always room for more features.

7.2 FaiRDy website

The FaiRDy-Py web framework successfully interfaces with the FaiRDy-Py package responsible for the computations of the simulations, and produces web-based plots that are comparable to the examples we received from the original FaiRDy module written in the Wolfram Language. In addition the simulations are stored in a database for future reference and analysis as specified in the project description. The accounts application reduces the risk of misuse of the system, and also makes it simpler for the user to view their own person history of simulations which have been ran.

While the scope of the web framework component and the time available for construction have not allowed exhaustive testing of the user interface and debugging, or extensive consideration of alternative architectures and tools, the end result is still a stable and useful product which will hopefully provide valuable insight to data storage professionals when designing new storage systems.

Nomenclature

BAF block availability factor

CA Cellular Automata

FaiRDy-Py Failure-Repair Dynamics Python package

GPC Generalized Pyramid Code

Rep Replication code

RS Reed Solomon Code

References

[1] V. Estrada-Galinanes, "A research playground for examining erasure codes", in *Proceedings of the 10th ACM International Systems and Storage Conference*, 2017, pp. 1–1.

[2] V. Estrada Galinanes, *Failure-repair dynamics study over a p2p system with an entangled storage layer*, Aug. 2018. DOI: 10.13140/RG.2.2.16003.14885.

[3] W. W. Peterson, "Error-correcting codes", *Scientific American*, vol. 206, no. 2, pp. 96–111, 1962.

[4] J. S. Plank, "A tutorial on reed–solomon coding for fault-tolerance in raid-like systems", *Software: Practice and Experience*, vol. 27, no. 9, pp. 995–1012, 1997.

[5] C. Huang, M. Chen, and J. Li, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems", *ACM Transactions on Storage (TOS)*, vol. 9, no. 1, pp. 1–28, 2013.

[6] J. L. Cheng Huang Minghua Chen, "Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems", *ACM Transactions on Storage*, vol. 9, no. 1, 2013. DOI: https://dl.acm.org/doi/abs/10.1145/2435204.2435207.

[7] E. W. Weisstein. (Mar. 2020). Cellular automaton, [Online]. Available: https://mathworld.wolfram.com/CellularAutomaton.html. (accessed: 15-Mar-2020), (from: MathWorld–A Wolfram Web Resource).

[8] S. Wolfram, "Statistical mechanics of cellular automata", *Reviews of modern physics*, vol. 55, no. 3, p. 601, 1983.

[9] M. Gardner, "The fantastic combinations of jhon conway's new solitaire game'life", *Sc. Am.*, vol. 223, pp. 20–123, 1970.

[10] W. Research. (May 2020). Wolfram language, [Online]. Available: https://www.wolfram.com/language/. (accessed: 24-May-2020).

[11] D. Robinson. (Sep. 2017). The incredible growth of python, [Online]. Available: https://stackoverflow.blog/2017/09/06/incredible-growth-python/. (Accessed 23-May-2020).

[12] t. s. q. c. TIOBE. (May 2020). Tiobe index for may 2020, [Online]. Available: https://www.tiobe.com/tiobe-index/. (Accessed 23-May-2020).

[13] python software foundation. (Jan. 2020). Pep 373 – python 2.7 release schedule, [Online]. Available: https://www.python.org/dev/peps/pep-0373/. (accessed: 24-May-2020).

[14] stack overflow inc. (May 2020). Stack overflow trends, [Online]. Available: https://insights.stackoverflow.com/trends?tags=java%2Cc%2Cc%2B%2B%2Cpython%2Cc%23%2Cvb.net%2Cjavascript%2Cassembly%2Cphp%2Cperl%2Cruby%2Cswift%2Cr%2Cobjective-c (Accessed 23-May-2020).

[15] the SciPy community. (May 2020). About numpy, [Online]. Available: https://numpy.org/. (Accessed: 20-May-2020).

[16] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. Jarrod Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. Carey, İ. Polat, Y. Feng, E. W. Moore, J. Vand erPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and S. 1. 0. Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python", *Nature Methods*, vol. 17, pp. 261–272, 2020. DOI: https://doi.org/10.1038/s41592-019-0686-2.

[17] T. S. community. (Feb. 2020). Routines, [Online]. Available: https://numpy.org/doc/1.18/reference/routines.html. (Accessed 23-May-2020).

[18] J. D. Hunter, "Matplotlib: A 2d graphics environment", *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[19] T. M. development team. (Oct. 2017). Usage, [Online]. Available: https://matplotlib.org/faq/usage_faq.html. (accessed 24-May-2020).

[20] T. jQuery Foundation. (May 2020). About jquery, [Online]. Available: https://learn.jquery.com/about-jquery/. (accessed: 10-May-2020).

[21] P. S. F. Chris Zwerschke. (Mar. 2020). Web frameworks for python, [Online]. Available: https://wiki.python.org/moin/WebFrameworks. (accessed: 07-Mar-2020).

[22] django software foundation. (Mar. 2020). Django documentation - models, [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/db/models/. (accessed: 09-Mar-2020).

[23] Wikipedia. (May 2020). Web template system, [Online]. Available: https://en.wikipedia.org/wiki/Web_template_system. (accessed: 10-May-2020).

[24] django software foundation. (May 2020). Django documentation - templates, [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/templates/. (accessed: 01-May-2020).

[25] D. software foundation. (Mar. 2020). Django documentation - django-admin and manage.py, runserver, [Online]. Available: https://docs.djangoproject.com/en/3.0/ref/django-admin/#runserver. (accessed: 15-Mar-2020).

[26] open-source. (May 2020). What is wsgi?, [Online]. Available: https://wsgi.readthedocs.io/en/latest/what.html. (accessed: 10-May-2020).

[27] python software foundation. (May 2020). Abc - abstract base classes, [Online]. Available: https://docs.python.org/3/library/abc.html. (accessed: 10-Mar-2020), (version: 3.8.3rc1).

[28] P. S. Foundation. (May 2020). Random — generate pseudo-random numbers, [Online]. Available: https://docs.python.org/3.8/library/random.html. (accessed: 21-Mar-2020), (version: 3.8.3rc1).

[29] python software foundation. (May 2020). Built-in functions, [Online]. Available: https://docs.python.org/3/library/functions.html. (accessed: 21-Mar-2020), (version: 3.8.3rc1).

[30] P. software foundation. (May 2020). Pickle — python object serialization, [Online]. Available: https://docs.python.org/3.8/library/pickle.html. (accessed: 24-May-2020), (version: 3.8.3).

[31] django software foundation. (May 2020). Documentation - applications, [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#extending-the-existing-user-model. (accessed: 01-May-2020).

[32] D. software foundation. (Mar. 2020). Django documentation - django.contrib.auth, [Online]. Available: https://docs.djangoproject.com/en/3.0/ref/contrib/auth/. (accessed: 09-Mar-2020).

[33] E. Commission. (Mar. 2020). Data protection in the eu, [Online]. Available: https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu_en. (accessed: 10-Mar-2020).

[34] django software foundation. (May 2020). Customizing authentication in django - extending the existing user model, [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/auth/customizing/#extending-the-existing-user-model. (accessed: 01-May-2020).

[35] P. software foundation. (May 2020). Secrets — generate secure random numbers for managing secrets, [Online]. Available: https://docs.python.org/3/library/secrets.html. (accessed: 10-May-2020), (version: 3.8.3).

[36] Google. (Mar. 2020). Google cloud guides - getting started with django, [Online]. Available: https://cloud.google.com/python/django. (accessed: 20-Mar-2020).

[37] python software foundation. (Mar. 2020). The python tutorial - 12. virtual environments and packages, [Online]. Available: https://docs.python.org/3/tutorial/venv.html. (accessed: 21-Mar-2020), (version: 3.8.3).

[38] django software foundation. (Mar. 2020). Django documentation - one-to-one relationships, [Online]. Available: https://docs.djangoproject.com/en/3.0/topics/db/examples/one_to_one. (accessed: 15-Mar-2020).

Appendix A

Guidance for Further Development

FaiRDy-Py is an open source project, and other developers are encouraged to pick up where we left off. In this section we will explain how to extend FaiRDy-Py with additional erasure codes. In addition to adding new erasure codes, other improvements to performance, interface, etc., are also desirable, but this will require more general familiarity with the Python programming language and associated libraries listed in the technologies Chapter, as well as the Python-based Django web framework.

A.1 fairdypy module

As stated in chapter 4 on the fairdypy package, the faridypy python package is structures in a hierarchical manner because of the fact that a Reed Solomon Code is a special case of Generalized Pyramid Code and Replication is a special case of Reed Solomon Code. The following section will be based on this implementation, and try to give some guidelines how to modify the already existing structure. This does by no means imply that this is the correct or best way to approach this problem, but it is in the end the way we chose. If you would like to contribute to the project, or make one of your own, feel free to try other approaches.

A.1.1 Adding classes

To add an erasure code class, you would first have to establish where in the current hierarchy, in figure 4.2, it would fit in. Say you have an erasure code which is not a special case of GPC, but it is on the same level. The Reed

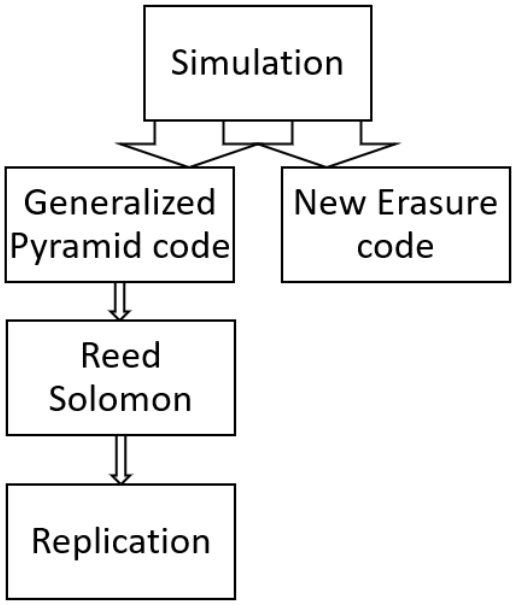


Figure A.1: Showing the hierarchy if a new erasure code, which does not inherit from GPC, nor does it succeed anything to other modules.

Solomon is also not a special case of this new erasure code. It would be placed parallel to the GPC, and below Simulation. It would inherit from Simulation, but it would not pass anything on to other Classes. In this case the class should be placed as shown in figure A.1. In this case the class would inherit from Simulation. The new class, when initiated, would have to initiate the Simulation class with `super.__init__()`, and all of the required inputs passed to super-initialization of the Simulation class. the functions for healing, and updating BAF would have to be implemented in the new class. This might also lead to some changes and modifications having to me made to the Simulation class, to have it fit with both the new class, and already existing classes.

A.1.2 Modifying or expanding classes

The generalized pyramid code which is implemented in FaiRDy-Py does only support straight horizontal and vertical Reed Solomon codes in a quadratic or rectangular structure. This does not have to be the case. One could change the class to be able to have erasure codes based on block indexes, and not by geometry. Doing this for GPC, would mean you would have to restructure healing-functions, to base it on indexes in the array, and also type of block referenced. Or you could add a separate healing function, and have it activate, instead of the standard ones, by setting a property on the object. The added functionality would also extend to already existing classes, and would have to be checked, or disabled in those classes.

Adding calculations during simulation

Currently all calculations are done in `loop_simulation`. The easiest way to add calculations during simulation this to the `loop_simulation` function in the Simulation class. One would also have to store the new values on the Simulation object, by creating a new attribute in the `__init__()` function. Before the loops start in `loop_simulation`, initialize a NumPy array the required size for loops to run. Then set its initial value. for each loop, then perform the calculation and add the value to the NumPy array by index. In the check for the BAF-limit, set the new variable to itself and index-slicing from index 0 to loops plus 2. This is to remove any unused indexes in the initialized NumPy array.

A.2 Django web project

As long as the implementation patterns in the fairdypy module are followed, it is relatively straightforward to also add a new erasure code to the Django web framework component of FaiRDy-Py. Only two of the fairdy application's Python modules need to be updated for a new erasure code, in addition to the documentation.

A.2.1 /web/fairdy/models.py

Class *Codes*

The *Codes* class is a subclass of the *Simulation* class located in the fairdy app's models.py file. This class is referred to in several different parts of the web framework, including to generate the lists of erasure codes in the

top navigation bar and the main simulation form. The class inherits from Django's models.TextChoices class and works similar to *enum* from Python's standard library.[22] Adding a new erasure code to this class is straightforward, simply define the name in uppercase with underscores, and the value with an uppercase abbreviation, enclosed in quotes, that is 10 characters or less. It is the value that is stored in the database in field *code_type* for the simulation.

```
# Erasure Codes which are currently installed on FaiRDy web
    framework
class Codes(models.TextChoices):
    REPLICATION = 'REP'
    REED_SOLOMON = 'RS'
    GENERALIZED_PYRAMID = 'GP'
```

Erasure Codes Requiring Additional Fields

Some erasure codes will require additional fields which are not already defined in the *Simulation* model or its corresponding *fairdy_simulation* database table. Of the three erasure codes we have already implemented in FaiRDy-Py, only generalized pyramid codes have so far required more fields than are defined in the main simulation model. Because the fields in the simulation model are meant to be universal for all erasure codes, we have chosen to store the extra fields using separate models with one-to-one relationships to the primary simulation model. Django offers a *OneToOneField* class for exactly this reason. [38]

To define additional fields for a new erasure code using Django's OneToOneField, create a new model class in models.py which has the necessary extra fields, as well as a field defined as a OneToOneField type with the Simulation model as its first parameter.

```
class NewErasureCodeSimulation(models.Model):
    sim = models.OneToOneField(
        Simulation,
        on_delete=models.CASCADE
    )
    extra_field = models.PositiveIntegerField(
        'necessary for running sims with new erasure code',
    )
```

After completing the new one-to-one model, the following commands are ran using Django's manage.py module.

```
$ python manage.py makemigrations fairdy
$ python manage.py migrate
```

The first command reads the models.py file for the fairdy application and creates a SQL script with instructions to update the database in line with modifications to the models. The second command runs the SQL script created in step one against the database.

A.2.2 Forms

If you have not defined a new one-to-one model to store additional fields for a new erasure code that will be set by user input, you can skip to /web/static/js/form.js part of this Forms section, otherwise it will also be necessary to define a Django form class for your new one-to-one model.

/web/fairdy/forms.py

For both the main simulation form, and the additional form for the generalized pyramid code, we have chosen to define our forms by inheriting from Django's *forms.ModelForm* class. By inheriting from this ModelForm class Django automatically creates a form object for the fields you choose, based on the field definitions that are already in models.py.

```

class PyramidSimulationForm(forms.ModelForm):
    class Meta:
        model = PyramidSimulation
        fields = ['num_repair_cycles', 'k_horizontal',
            'm_horizontal', 'k_vertical', 'm_vertical',
            'lazy_heal_threshold_ver']
```

/web/fairdy/models.py

After the new form class is defined it must also be enabled in the *run_simulation(request)* function back in models.py, as we have done for generalized pyramid codes. When the user has a GET request to the run_simulation function, all forms are initialized and sent, for a POST request the main simulation form is validated, and one-to-one forms are validated as needed based on the selected erasure code. In an attempt to keep our code a little cleaner we have defined a separate method to handle the initialization of a pyramid code simulation when necessary, and called this from the main run_simulation(request) function.

```
def initialize_gp(request, web_sim, sim_form, gpc_form):
    if gpc_form.is_valid():
        gpc_sim = gpc_form.save(commit=False)
        gpc_sim.simulation = web_sim
```

```
        gpc_sim.save()
    else:
        messages.error(request, gpc_form.non_field_errors())
        return render(
            request,
            'form.html',
            {
                'base_form': sim_form,
                'gpc_form': gpc_form
            }
        )
```

The helper function being called in the main function:

```
def run_simulation(request):

```

⋮

```
    if web_sim.code_type ==
        Simulation.Codes.GENERALIZED_PYRAMID:

        initialize_gp(request, web_sim, sim_form, gpc_form)

```

⋮

/web/fairdy/templates/forms.html

Any additional fields that have been added for the new erasure code will also need to be added to this html file. As you can see in the section for pyramid codes:

```
<!--GPC FIELDS-->
<div class="gpc-fields">
    <div class="form-group row">
        <label class="col-12 col-sm-9 col-form-label pl-0" for="{{ gpc_form.k_horizontal.name }}">{{ gpc_form.k_horizontal.label }}:</label>
        <input type="number" min=0 max=10000 class="col-6 col-sm-3 form-control" data-toggle="popover"
               data-content="<img src='{% static 'fairdy/gpc_k_hor.svg' %}'></img>"
               name="{{ gpc_form.k_horizontal.name }}" id="{{ gpc_form.k_horizontal.auto_id }}">
        {{ gpc_form.k_horizontal.errors  }}
    </div>

```

⋮

```
</div>
```

The special fields are wrapped in a *div* element with a CSS class *gpc-fields* which we have been defined to allow easy hiding and showing of GPC fields based on the selected erasure code at the top of the form. This behavior is defined in the *form.js* file described in the following subsection, and the main function in this file, *refreshFields()*, is set to run any time the active fields or selection boxes are modified in an inline script in the HTML page. The HTML for the k_horizonatal special GPC input also shows we have defined the popover for this field in the HTML instead of the JavaScript, since these fields will be hidden for other code types, so the popover is never changed on these elements.

```
$(document).ready(function () {
    refreshFields();

    //open the optional parameters section when the + button
   is clicked
    $('#toggleOptionsButton').on('click', function () {
        $('#option-fields').toggle();
    });

    //refresh the disabled fields whenever a non-disabled
    //field is changed or a new code type is selected
    $('select').change(refreshFields);
    $('input:not([readonly])').change(refreshFields);

    //enable the popovers on the input fields
    $(function () {
        $('[data-toggle="popover"]')
            .popover({
                trigger: 'focus',
                placement: 'bottom',
                html:true
            });
    });
});
```

/web/static/js/form.js

The web framework's main form is highly dependent on JavaScript to provide a responsive interface which prevents user input and automatically calculates fields that may not be necessary for one erasure code, but require user input for another. All of this behavior is defined in form.js kept in the /web/static/js folder. The principle function in this file is called refreshFields(), and

inside this function it will be necessary to add the new erasure code to the switch statement.

```
switch(code) {
    case 'REP':
        refreshREP();
        break;
    case 'RS':
        refreshRS();
        break;
    case 'GP':
        refreshGP();
        break;
    case '':
        $('.gpc-fields').hide();
        break;
}
```

As you can see, it will also be necessary to define a new function for erasure code that is being added. For example, the following JavaScript function prepares the form to accept input for a simulation using Reed Solomon erasure codes:

```
function refreshRS() {
    $('.gpc-fields').hide();

    //handle m_value field
    let m_val = parseInt($('#id_m_value').val());
    $('#id_m_value').attr(
        'data-content',
        'The number of equally sized redundancy blocks
            that each stripe will contain.');

    //handle k_value field
    let k_val = parseInt($('#id_k_value').val())
    $('#id_m_value').attr(
        'data-content',
        'The number of equally sized data blocks that
            each stripe will contain.');

    //handle n_value field
    let n_val = m_val + k_val;
    $('#id_n_value').val(n_val);
    $('#id_n_value').attr('readonly', true);
}
```

First the special fields for GPC codes are hidden, then we read in the values of the k_value and m_value fields, as well as set the content of the popover for these fields with tips for the field that are applicable to the selected erasure

code. Finally, the n_value field is marked as *readonly*, disabling user input, and the value is set to be the sum of k_value and m_value.

A.2.3 /web/fairdy/views.py

function *run_module_simulation*

This function is the main interface between the fairdypy module, which actually runs the simulations, and the web framework which stores and displays them. If you are implementing a new erasure code without any additional fields, for views.py it should be sufficient simply to add the new codes to the *standard_codes* list.

```
def run_module_simulation(web_sim):
    # STANDARD CODES
    standard_codes = [
        Simulation.Codes.REED_SOLOMON,
        Simulation.Codes.REPLICATION
    ]
    if web_sim.code_type in standard_codes:
        module_sim = fairdypy.Reed_solomon(
            word_blocks=web_sim.k_value,
            extra_blocks=web_sim.m_value,
            num_of_stripes=web_sim.num_stripes,
            lazy_heal_threshold=web_sim.lazy_heal_threshold,
            storage_fault_mode=web_sim.storage_location_mode,
            num_of_storage=web_sim.num_storage_locations
                if web_sim.storage_location_mode else None,
        )
```

If the new erasure code does require additional fields, a custom interface will likely need to be defined as we have done for generalized pyramid codes.

A.2.4 /web/fairdy/templates/help/

This folder contains the 3 main HTML help files for the *fairdy* application:

- getting_started.html
- run_simulation.html
- compare_download.html

Any new erasure code that is added will need to be described in getting_started.html, in addition to this, there should also be a section in run_simulation.html which explains in detail how to fill out the form to run a storage system simulation with the new erasure code.

A.2.5 Adding fields to class Simulation

Instead of adding a field to a new Django one-to-one model, it should also be possible to add a new field to the Simulation model itself, especially if the field could be considered common for simulations with many different erasure codes. In this case, the procedure will be similar to adding a new one-to-one model, the modifications will just take place across already created models and forms.

Appendix B

Website documentation

Below, you can find the documentation from the FaiRDy website. it includes an introduction to the principles behind FaiRDy, as well as how to use the website and how to create and manage your user.

Getting Started

FaiRDy is a lightweight approach to simulate the failure-repair dynamics of storage systems and explore the effects of different redundancy strategies and parameters on system degradation. FaiRDy is based on cellular automata, the blocks of storage system data are represented as cells, which can either be available or unavailable. The simulations are run in two-part cycles, in the first half of the cycle, cells are randomly marked as unavailable according to the specified probability of failure. In the second half of the cycle, the unavailable blocks are repaired where possible according to rules of the erasure code.

To evaluate the performance of a storage system over time FaiRDy computes a *Block Availability Factor (BAF)* at the end of every cycle, which is the proportion of blocks in the system which were available at the end of the cycle.

Here you will find a quick overview of key concepts central to FaiRDy and this website.

Account Registration

FaiRDy requires registration of an account before running simulations. This enables you to view a history of simulations you have run, and also to prevent the misuse of the university hardware this website is hosted on. We will never share your details with any third party, only your username is publicly visible, and this may be changed at any time. You may delete your account, and all information associated with it, at any time. The simulations you have ran will remain in our database, but will no longer be associated with any user account.

Cellular Automata

A cellular automaton (pl. cellular automata, abbrev. CA) is a collection of boolean cells, also referred to as blocks in fairdy-py, which together form a system. This system is then evolved through a set of discrete time cycles, with the state of a block in a given cycle determined by its neighbors' states in the previous cycle.

In FaiRDy the CA represents a digital storage system, and the individual cells represent blocks of data contained within this system. To begin with, all data blocks in the system are considered available, however in the first step of each time cycle blocks are randomly marked as unavailable by a fault- injection module, using a pre-specified probability of failure, to simulate real world events which lead to data loss. In the second step of time cycle, an attempt is made to repair the failures introduced in the first step, whether or not this is successful is determined by a set of rules specific to the erasure code being modelled.

Erasure Codes

An erasure code encodes redundancy into a stream of data, this in turn ensures the data is recoverable even if certain blocks in the stream are erased or otherwise unavailable. Optimal erasure codes enable recovery of data up to the point that the number of erased blocks equals the number of redundancy blocks.

In FaiRDy we define a *stripe* in terms of the dimensions of the erasure code to be investigated. After the stripe is defined, a specified amount of stripes are concatenated to form a complete storage system model which is then cycled between fault-injection and repair to gain insight into the storage system's properties.

---

n-way replication

N-way replication is the most simple way to store data redundantly, and in FaiRDy a n-way replication stripe consists of a single data block, and n complete replicas.

Repair Rules:
A n-way replication stripe is considered repairable if at least one copy of the data is not erased.

example: a two-way replication stripe with 3 blocks total: 1 for data, and 2 for copies of the data

Reed-Solomon codes

Reed-Solomon (RS) codes are optimal erasure codes, such codes are also called maximum distance separable (MDS) codes. RS codes tolerate a number of block erasures equal to the number of redundancy blocks in the stripe. In real storage systems the contents of the redundancy blocks are computed and decoded using advanced linear algebra, this is however ignored in FaiRDy and a stripe will be repaired as long as enough active blocks remain to allow this.

Repair Rules:
A Reed-Solomon stripe is considered repairable as long as the number of erased blocks does not exceed the number of redundancy blocks.

RS(5,4)

example: a Reed-Solomon stripe with 9 blocks total: 5 for data, and 4 for redundancy

Generalized Pyramid codes

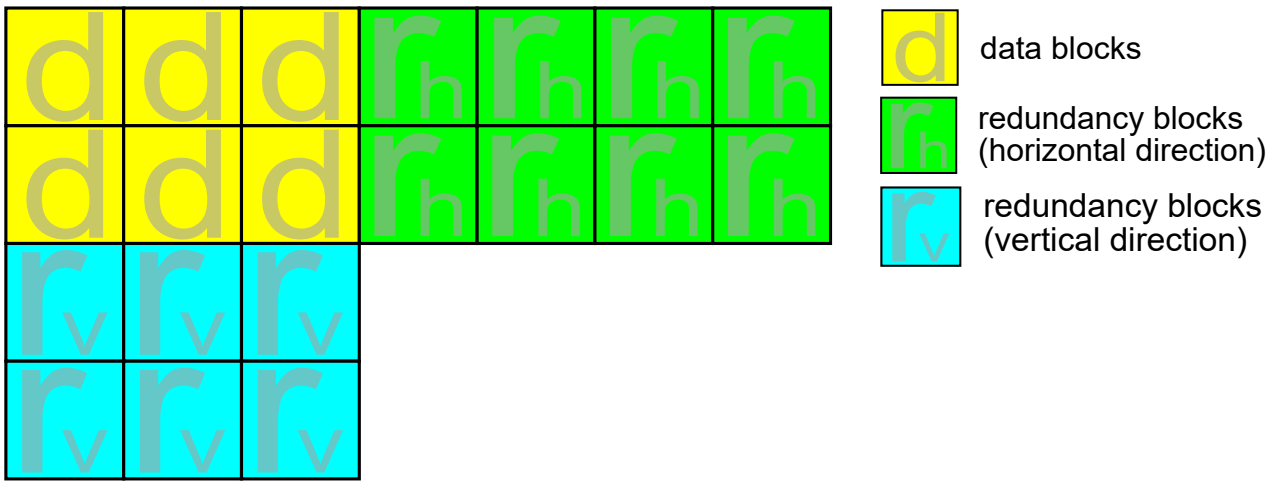
The Generalized Pyramid codes implemented by FaiRDy are composed of maximum distance separable (MDS) codes, such as Reed-Solomon stripes, where data blocks belong to two stripes. one in the vertical direction and one in the horizontal direction. This provides an extra layer of protection since each data block is a member of two different Reed-Solomon stripes, with each providing a possibility for repair. Two different Generalized Pyramid code stripe constructions are possible in FaiRDy, a standard non-overlapping with the blocks arranged in an L-shaped polygon, and an overlapping construction with the blocks arrange in the shape of a rectangle. The overlapping construction adds an additional dimension of redundancy, with the overlapping redundancy blocks computed using the vertically redundant as data input blocks.

Repair Rules:
In a Generalized Pyramid stripe, the vertical and horizontal Reed-Solomon stripes are repaired individually in accordance with the rules for this erasure code. If all blocks in theGeneralized Pyramid stripe are able to be repaired in this way, the stripe is considered completely repaired.

non-overlapping

The number of horizontal stripes is equal to the number of DATA blocks in the vertical stripes, and correspondingly, the number of vertical stripes is equal to the number of DATA blocks in horizontal stripes.

GPC(20,6) H:{3, 4} V:{2, 2}

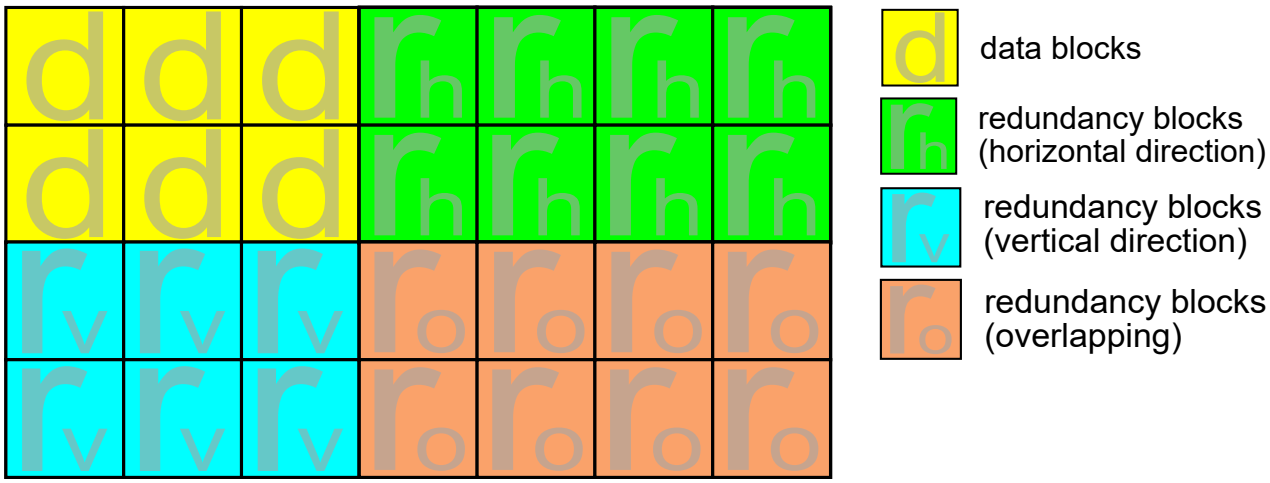


example: a non-overlapping Generalized Pyramid stripe with 20 blocks total: 6 for data, and 14 for redundancy

overlapping

The number of horizontal stripes is equal to the TOTAL number of blocks in the vertical stripe, and correspondingly, the number of vertical stripes is equal to the TOTAL number of blocks in the horizontal stripe. The overlapping redundancy blocks are computed using the vertical redundancy blocks as the input data blocks, and thereby provide second degree redundancy.

GPC(28,6) H:{3, 4} V:{2, 2}



an overlapping Generalized Pyramid stripe with 28 blocks total: 6 for data, and 22 for redundancy

How Simulations Work

Simulations are carried out in a dedicated Python module that can also be used on the command line to simulate the life cycles of storage systems. This module uses numpy arrays to store the state of the individual blocks and stripes, and cycles through fault-injection and fault-repair steps to simulate the evolution of the storage system.

References

- Cellular Automaton, from MathWorld--A Wolfram Web Resource
- Cellular Automaton, on Wikipedia
- Erasure Code, on Wikipedia
- Reed-Solomon Codes, on Wikipedia

Running Simulations

Not sure what we're talking about here? Check out the Getting Started page first.

In order to run storage system life cycle simulations on this website you must register an account and validate your email address. In addition, there is a generous limit on the cumulative amount of computational resources your simulations may consume over the course of a month. See here for more details.

Here you will find a field-by-field explanation for the form used to create and run simulations.

Define Stripe

In this section you define the shape of the stripe that will be used to build your system. After choosing an erasure code for your simulation the relevant input fields will be activated, and others may be calculated automatically based on a combination of the erasure code's properties and your input in other fields.

FaiRDy assumes that each block in a stripe, whether it contains data, copies, or computed redundancy, occupies the same amount of space on disk as the others, whether that be 1 kilobyte or 1 terabyte.

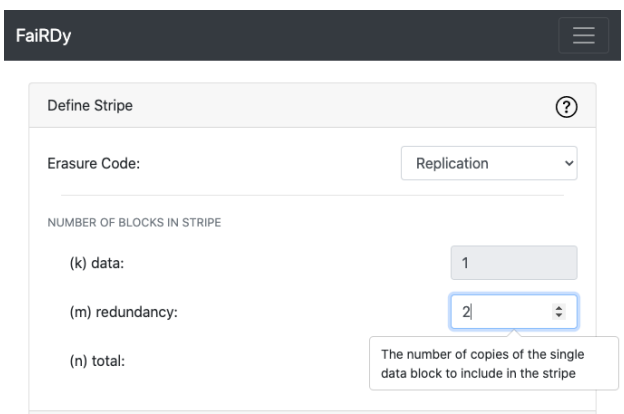
---

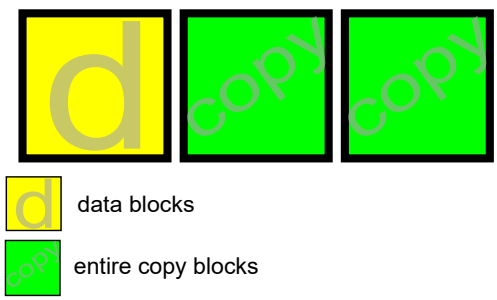
n-way Replication

An n-way replication stripe has only one single data block, so the only field necessary to define is the number of times this one data block is replicated within the stripe. The stripe will remain recoverable so long as at least one block containing either the original data, or a copy, remain available.

Parameters

(m) redundancy	the number of complete copies of your single data block in each stripe





example: defining a two-way replication stripe with 3 blocks total: 1 for data, and 2 for copies of the data

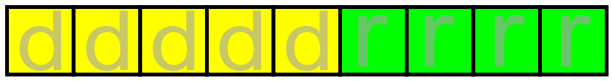
---

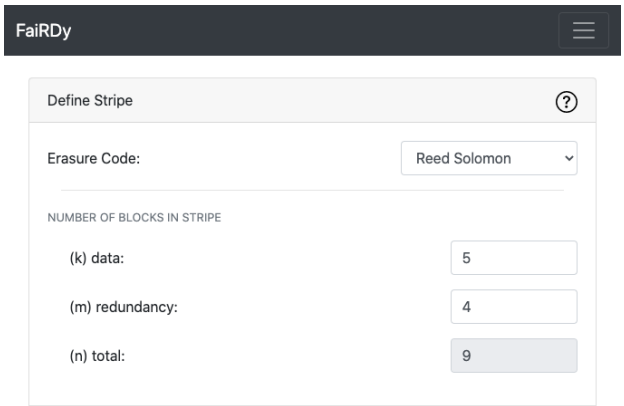
Reed-Solomon Codes

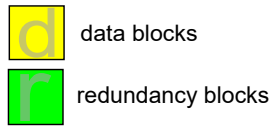
A Reed-Solomon stripe has user-defined numbers of both data and redundancy blocks, the total number of blocks in the stripe is disabled for input and automatically calculated as the sum of the first two numbers. The stripe will remain recoverable so long as the number of erased blocks does not exceed the number of redundancy blocks.

Parameters

(k) data	the number of complete copies of your single data block in each stripe
(m) redundancy	the number of complete copies of your single data block in each stripe





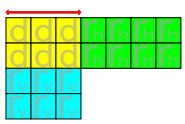
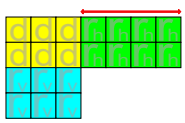
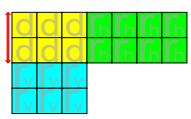
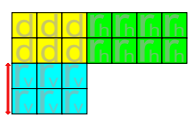


example: defining a RS(5,4) stripe with 9 blocks total: 5 for data, and 4 for redundancy

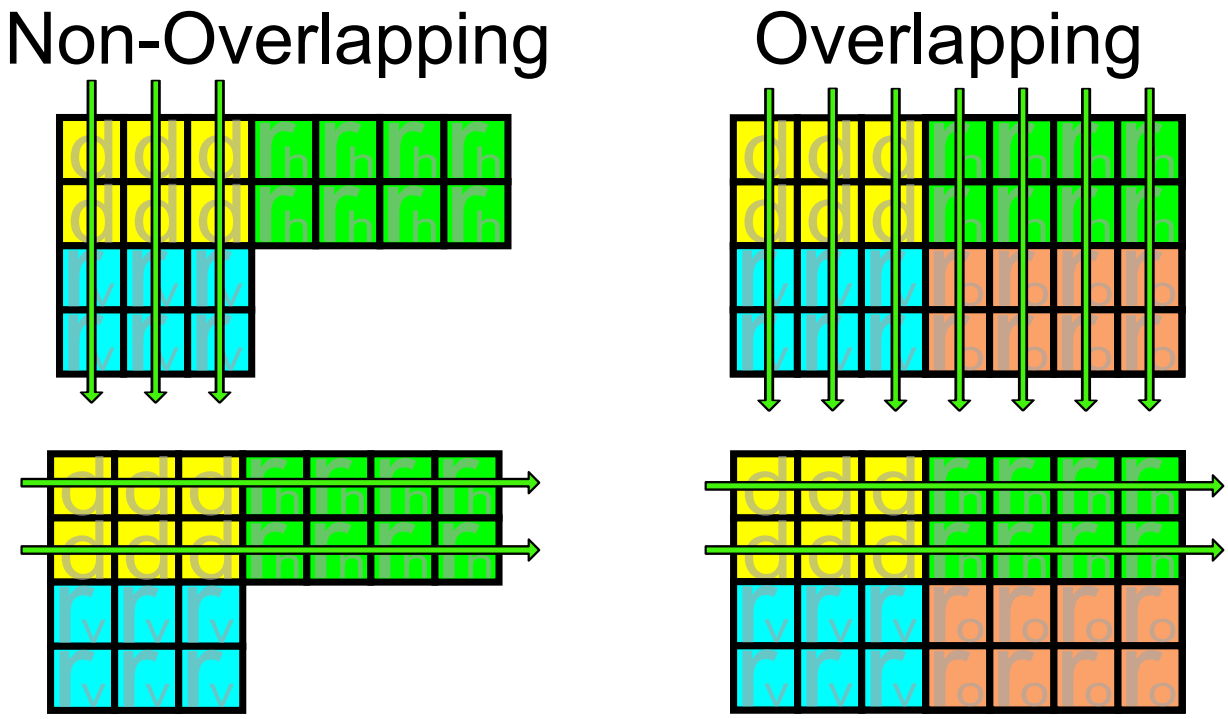
Generalized Pyramid Codes

A Generalized Pyramid stripe has a user-defined total number of blocks per stripe. The number of both redundancy and data blocks per stripe is disabled for input and calculated automatically based on the additional horizontal and vertical parameters. In FaiRDy, these additional parameters define up to two different subordinate Reed-Solomon stripe shapes which are used to compose the stripe's pyramid structure. The form will attempt to calculate whether your stripe is overlapping or not, based on the provided parameters.

Parameters

(n) total		the total number of blocks in the stripe, including data, copy, and computed redundancy blocks
(k_horizontal) data		the number of data blocks in the horizontal direction
(m_horizontal) redundancy		the number of redundancy blocks in the horizontal direction
(k_vertical) data		the number of data blocks in the vertical direction
(m_vertical) redundancy		the number of redundancy blocks in the vertical direction

Repair Steps per Cycle



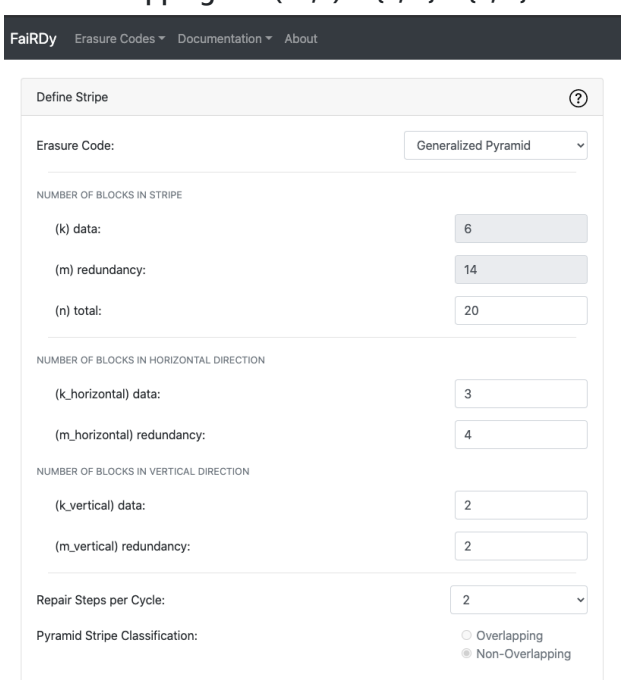
The number of repair steps completed in a single cycle.

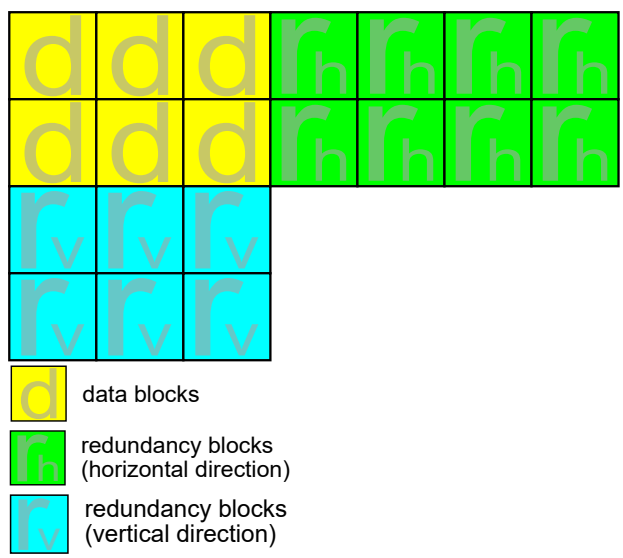
A single repair step heals Reed-Solomon sub-stripes first vertically, and then horizontally. You may choose to repeat the repair step up to three times in an individual cycle.

Pyramid Stripe Classification

This field is always closed for user input and attempts to determine whether the parameters you have entered can be interpreted as an overlapping or non-overlapping Generalized Pyramid stripe.

Non-Overlapping GPC(20,6) H:{3, 4} V:{2, 2}

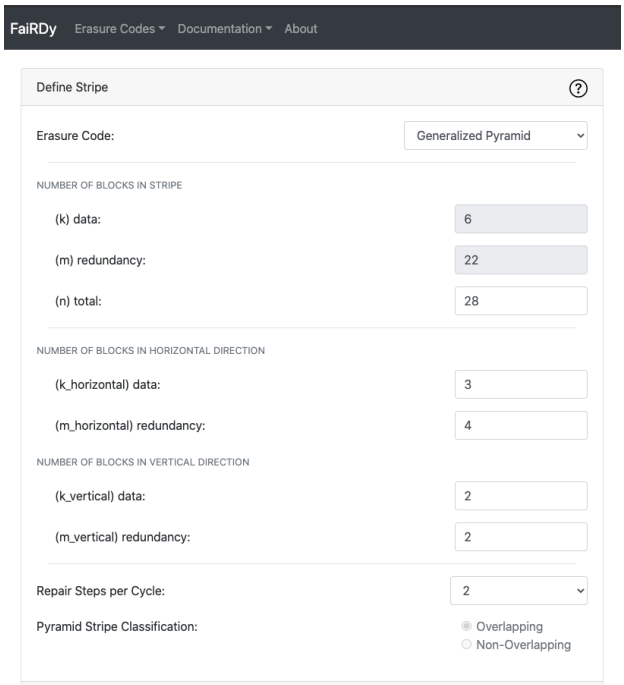


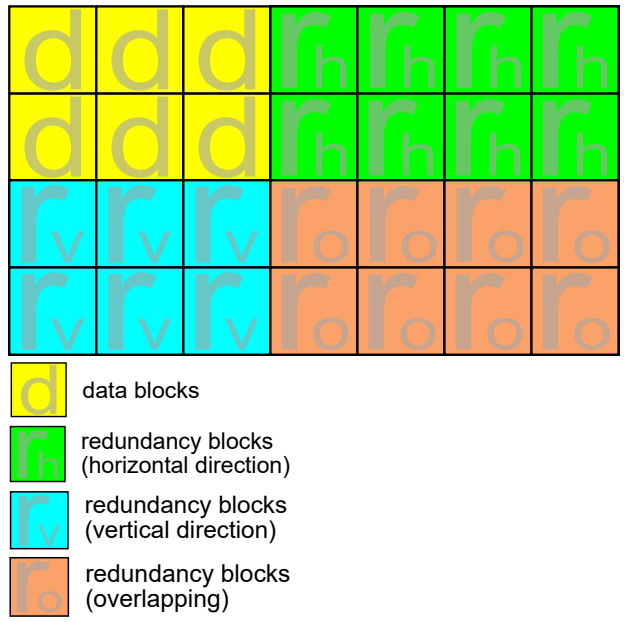


example: defining a GPC(20,6) H:{3, 4} V:{2, 2} stripe with 20 blocks total: 6 for data, and 14 for redundancy

---

Overlapping GPC(28,6) H:{3, 4} V:{2, 2}



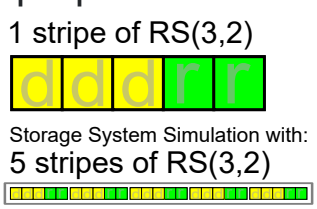


example: defining a GPC(28,6) H:{3, 4} V:{2, 2} stripe with 28 blocks total: 6 for data, and 22 for redundancy

Define Storage System and Simulation Parameters

In this section you define additional parameters required to run your simulation. In FaiRDy a *storage system* consists of a user-defined number of stripes concatenated together. This system is then evolved through a specified number of cycles, with each cycle consisting of both a failure-injection step, and a subsequent repair step, during which the stripe is healed if the erasure-code-specific criteria are met.

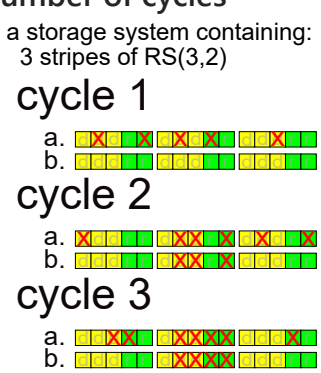
Stripes per simulation



This field is the number of times the stripe you have defined is repeated before being concatenated into a storage system. The total number of blocks in your system will be the number of blocks per stripe multiplied by the number of stripes.

TIP: choose a number which will give your system more than 1,000 total blocks. With fewer blocks than this in your system you may notice larger variations between simulations of identical systems because of different random seeds. (we do not set random seeds in FaiRDy-Py, on this website they are provided by the server's OS)

Number of cycles

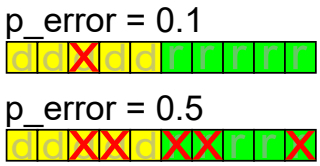


The number of two-part cycles, each with fault-injection and repair, your storage system is evolved through.

TIP: 100 cycles are usually sufficient for most codes to get a feel for the evolution of the system over time.

(p_error) probability of block failure

The probability an available block in the system fails in the fault

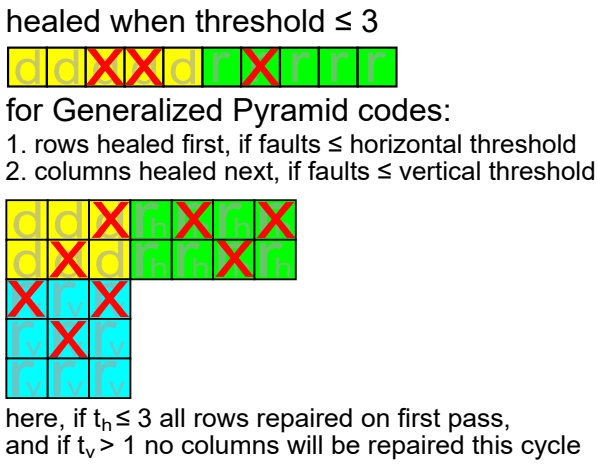


injection step of an individual cycle, when an advanced storage location mode is enabled, the probability an individual storage location fails.

Optional Parameters

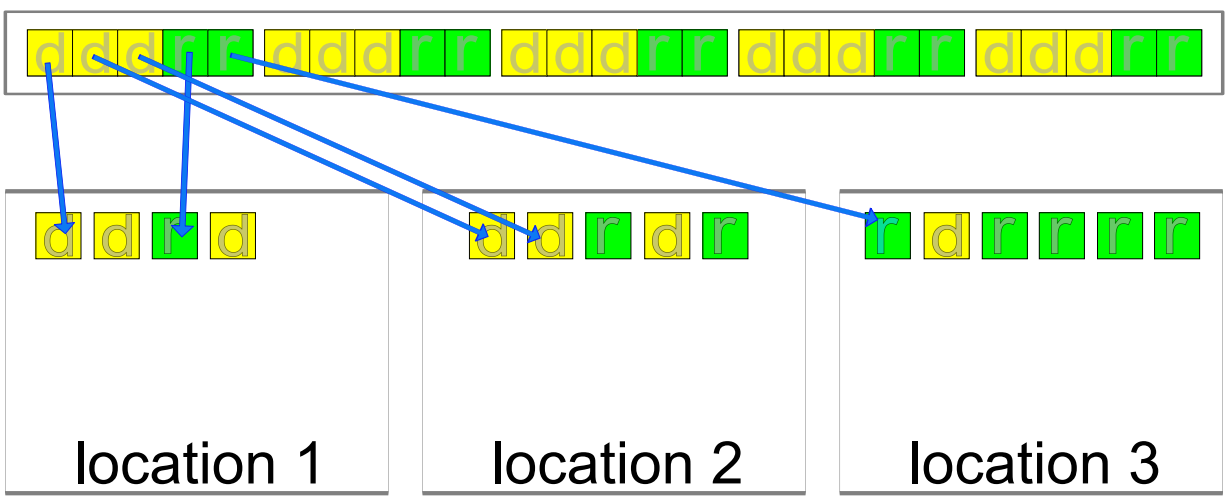
In this section you will choose supplementary advanced parameters for your simulation.

Lazy Heal Threshold



Lazy heal delays the healing of a stripe, or sub-stripe for Generalized Pyramid systems, until the specified number of blocks are first marked as unavailable. In real-world storage systems delaying repairs in certain situations can save bandwidth and other resources, while impacting the rate of system degradation only minimally.

Storage Location Mode



When you select a storage location mode, the blocks which form your system will be assigned to an individual storage location independent of their stripe. Then, during the failure-injection step, failures are applied to the storage locations instead of individual blocks. When a storage location fails, all block within it do so also, but repairs are still conducted according to the stripe which the block belongs to.

Options

Random	blocks are randomly assigned a storage location
Equal Shuffle	blocks are distributed as equally as possible using modulo with the block's sequence number in the system
Hash Block Index	similar to Equal Shuffle, but the block's sequence number is first converted to a string and put through a hash function before modulo is applied to assign the storage location, it is meant to more realistically simulate real-world storage system behavior than Equal Shuffle

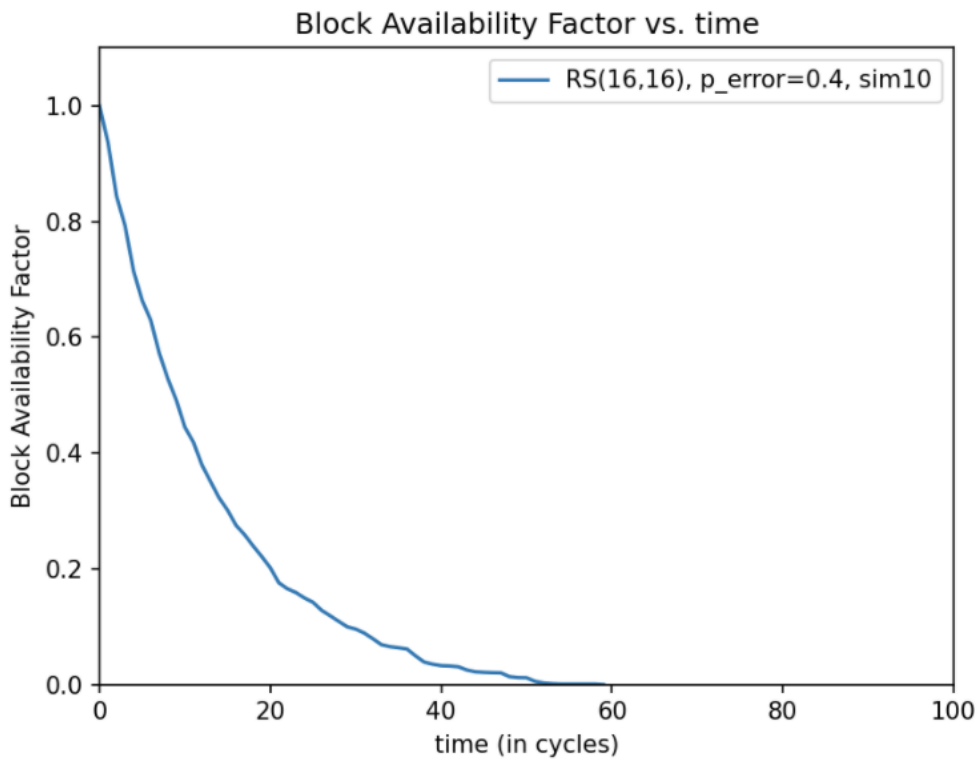
View, Compare, and Download Simulations

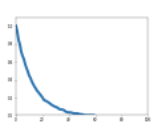
In addition to running and storing individual storage system life cycle simulations, you are also able to compare the results of several simulations simultaneously and download the raw simulation data.

Both the compare and download features are controlled by the checkboxes and buttons on the main simulation index page. Simply check the boxes you wish to compare or download, and click the appropriate button. Your selections will be stored as you navigate between different page numbers.

View

After a simulation is finished processing you will be redirected to a page where you can view the results. This page includes a plot of the Block Availability Factor throughout the evolution of your system, as well as details of the system itself.



storage system	curve	blocks in stripe	system dimension	options	details
RS(16,16), p_error=0.4 view sim10		16 data 16 redundancy 32 total	100 cycles 100 stripes 3200 blocks		Adam 23-May-2020

The evolution of a storage system shown in terms of BAF during each cycle

Compare

To view and compare the results of several simulations simultaneously, check the box for the relevant simulations followed by the "Compare" Button.

FaiRDy Erasure Codes Documentation About Adam

Simulations of Storage System Life Cycles

New

showing systems using Reed Solomon erasure codes

storage system	curve	blocks in stripe	system dimension	options	details
☐ RS(16,16), p_error=0.4 view sim10		16 data 16 redundancy 32 total	100 cycles 100 stripes 3200 blocks		Adam 23-May-2020
☑ RS(32,32), p_error=0.3 view sim2		32 data 32 redundancy 64 total	100 cycles 1000 stripes 64000 blocks	**lazy heal threshold (horizontal)** 16	Adam 22-May-2020
☑ RS(32,32), p_error=0.3 view sim1		32 data 32 redundancy 64 total	100 cycles 1000 stripes 64000 blocks		Adam 22-May-2020

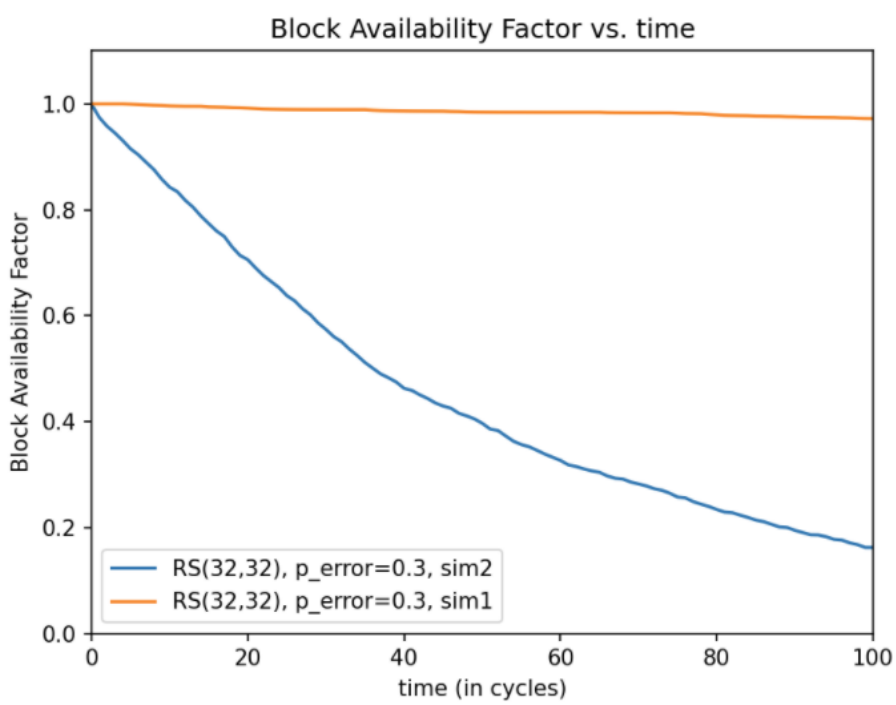
Previous 1 Next

Page 1 of 1

Download Compare

Selecting two Reed-Solomon systems for comparison

FaiRDy



storage system	curve	blocks in stripe	system dimension	options	details
RS(32,32), p_error=0.3 view sim2		32 data 32 redundancy 64 total	100 cycles 1000 stripes 64000 blocks	**lazy heal threshold (horizontal)** 16	Adam 22-May-2020
RS(32,32), p_error=0.3 view sim1		32 data 32 redundancy 64 total	100 cycles 1000 stripes 64000 blocks		Adam 22-May-2020

Comparing these two simulations with otherwise identical specifications we see that the lazy heal threshold of 16 has a serious negative effect on the evolution of the system over time.

Download

You can download the results of any simulation in a comma separated values (CSV) file to analyze or plot the data in software of your choice. The data includes the parameters used to define the storage system and simulation, along with 3 pieces of data for each of the simulation's cycles: block availability factor, number of blocks faulted this cycle, and number of blocks healed this cycle.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V
1	id	code_typ	n_valu	k_val	m_valu	num_strip	num_cyc	p_error	laz	sto	nu	k_horiz	m_hori	k_vert	m_ver	lazy_h	num_re	cycle 0	cycle 1	cycle 2	cycle 3	cycle 4
2	7	GP	32	8	24	100	100	0.4	1		#	4	4	2	2	1	2	1	0.9319	0.8903	0.8697	0.8538
3	faulted_blocks																	0	1293	1207	1139	1130
4	healed_blocks																	0	1075	1074	1073	1079
5	6	GP	32	8	24	100	100	0.3	1		#	4	4	2	2	1	2	1	0.9631	0.9609	0.9547	0.96
6	faulted_blocks																	0	992	910	920	878
7	healed_blocks																	0	874	903	900	895
8																						

example: downloaded simulation data shown in Microsoft Excel

How to create an account on this website.

FaiRDy requires registration of an account before running simulations. This enables you to view a history of simulations you have run, and also to prevent the misuse of the university hardware this website is hosted on. We will never share your details with any third party, only your username is publicly visible, and this may be changed at any time. You may delete your account, and all information associated with it, at any time. The simulations you have ran will remain in our database, but will no longer be associated with any user account.

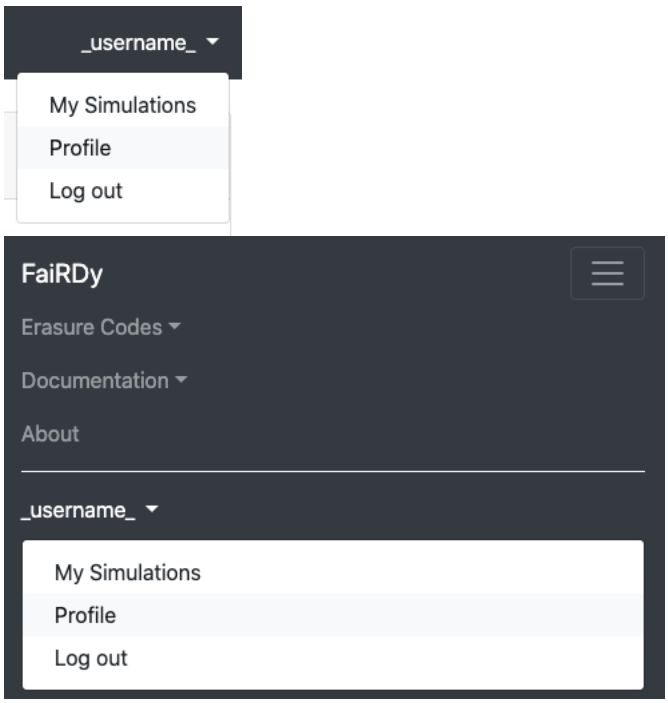
1. **Go to the registration page.**
2. **Choose a username**- You may use your email, a pseudonym, or anything you wish as a username, but remember it will be publicly visible on the website on the simulations you run. You can change your username at a later date from your profile page.
   - must be 150 characters or fewer
   - must be composed of alphanumeric characters, and _ @ + . -
3. **Choose a password**- enter twice to verify.
   - must be at least 9 characters long
   - must not be too similar to username or email
   - must not be in a list of the 20,000 most common passwords
   - must not be entirely numeric
4. **Confirm you agree with terms and conditions:**
   - This website uses cookies to optimize user experience by storing information about your session.
   - This website will store your username and email, as well as timestamps associated with registration, login, use, etc.
   - The simulations you run will be publicly visible with your username, your email will only be visible to website administrators.
   - You may delete your account at any time by going to your profile and following the 'Delete Account' link.
   - After you delete your account, all simulations you have run will remain in our database, but without any reference to your former account or personal data.
5. **Follow the link in your email to verify your account.** If you have not received a link:
   - Check your spam folder
   - Resend the email confirmation from your profile page when logged in
   - Try using another email address, this can be updated on your profile page when logged in.
6. **Start simulating storage system life cycles!**
   - You can only run simulations when you are logged in.
   - The number of simulations you can run within a 30 day period is limited, check your profile page for the current status.

Manage your FaiRDy account

- User Menu
- Profile Page
  - Remaining Credits
  - Email Validation
- Delete Account
- Forgot Password

User Menu

The user menu is located in the upper right corner of the page. Before logging in, Sign Up and Log In links are shown. After logging in, the username is displayed with an icon to open a drop-down menu.

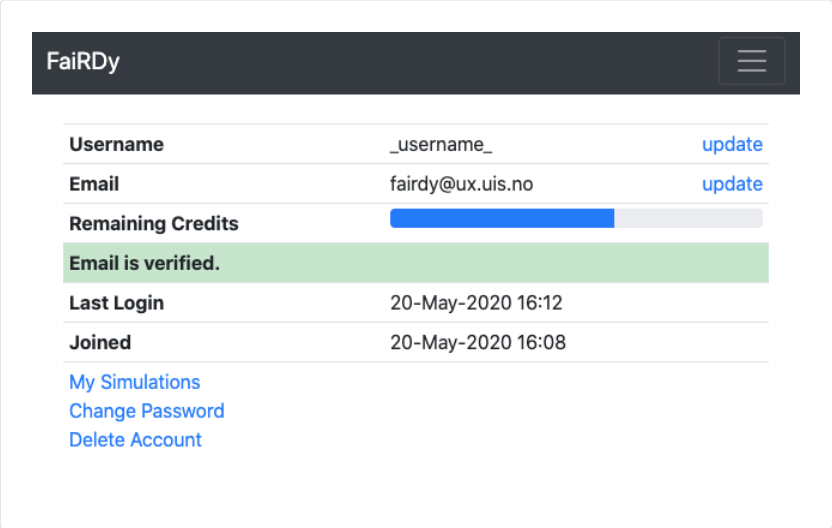


the user menu on large and small screens

My Simulations	shows a history of all simulations run with your account
Profile	the main page for viewing and modifying account details
Log out	log out of website and return to home page

Profile Page

When logged in, you can access your profile page from the user menu. The profile page shows key details about your account and contains links to update your username, email, and password. If you wish to delete your account, you will find the link at the bottom of your profile page.



Remaining Credits

While FaiRDy is a lightweight framework for characterizing the reliability of storage systems over their lifecycle, it still requires a certain amount of compute and memory to process the simulations. For this reason and because the server is hosted on hardware at the University of Stavanger, new accounts have a *Block Cycle Limit*, which limits the amount of simulations within a month. Block Cycle Limit is calculated as the size of the storage system (in blocks) multiplied by the number of cycles. New accounts have a default Block Cycle Limit of 500 million, your remaining credits are shown with a progress bar on your profile page. Considering that most systems reveal few additional insights after 100 cycles, the default limit allows you to run 100 simulations with up to 50,000 total blocks in each system. This should be sufficient for most cases to casually explore the effects of erasure code parameters on storage system life cycles, but please contact us if you would like to request an increase in your limit.

Email Validation

You will not be able to run simulation until you verify your email address by clicking a link we email to you. If you have not received a validation email, check your spam folder, or use the links on your profile page to send a new confirmation or change your email address. Note that changing your email will require you validate the new email address before running any simulations.

Delete Account

If you wish to delete your FaiRDy account we will remove all of your personal information from our database. Either follow the link in your profile page, or click here. The simulations you have ran with your account will remain in our database, but they will no longer be associated with your former username. You must enter your password to delete your account, if you have forgotten your password, first reset your password here, then follow the delete link from your profile page after validating your email.

Forgot Password

If you have forgotten your password, go to the forgot password page and enter your username. You will receive a validation email, and upon following the link it contains you will be able to reset the password for your account.

FaiRDy

Forgot Password

Enter your username below, and follow the link in your email to reset your password.

Username

Submit

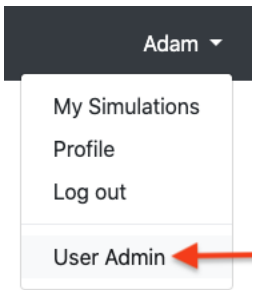
Register? Log in?

Manage FaiRDy Users

This help page, and the user management page are hidden from all accounts which are not marked "is staff".

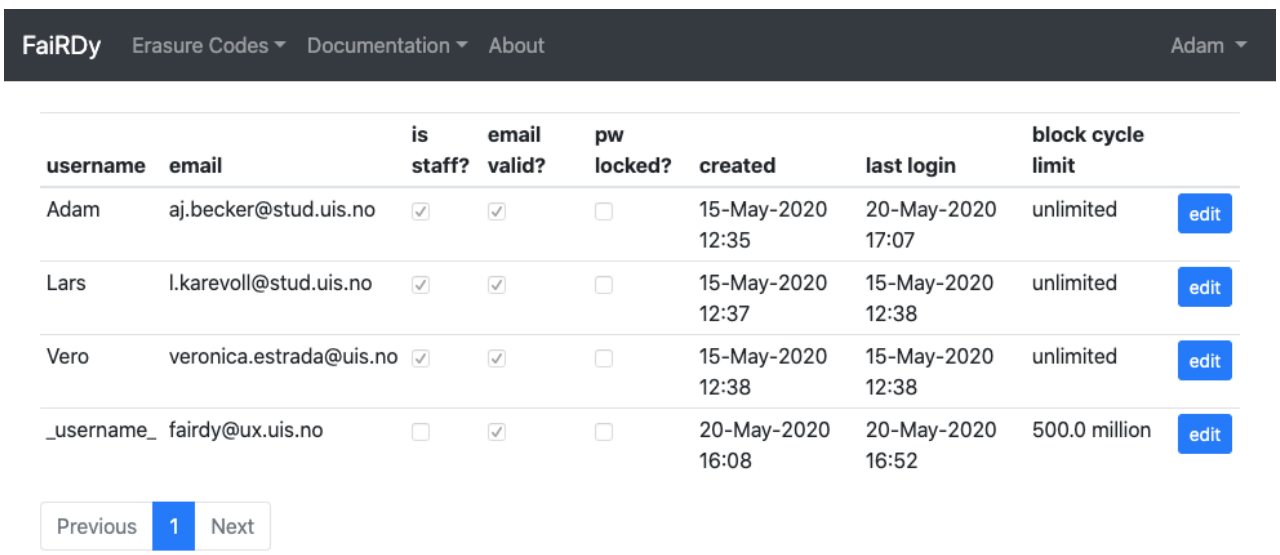
Accessing Page

You can access the user management page by clicking here or by clicking the link in the user menu, which is only visible for accounts marked as "is staff".



User Index Page

From this page it is possible to view all users who have registered for this website and various details about their account.



username	email	is staff?	email valid?	pw locked?	created	last login	block cycle limit	
Adam	aj.becker@stud.uis.no	✓	✓		15-May-2020 12:35	20-May-2020 17:07	unlimited	edit
Lars	l.karevoll@stud.uis.no	✓	✓		15-May-2020 12:37	15-May-2020 12:38	unlimited	edit
Vero	veronica.estrada@uis.no	✓	✓		15-May-2020 12:38	15-May-2020 12:38	unlimited	edit
username	fairdy@ux.uis.no		✓		20-May-2020 16:08	20-May-2020 16:52	500.0 million	edit

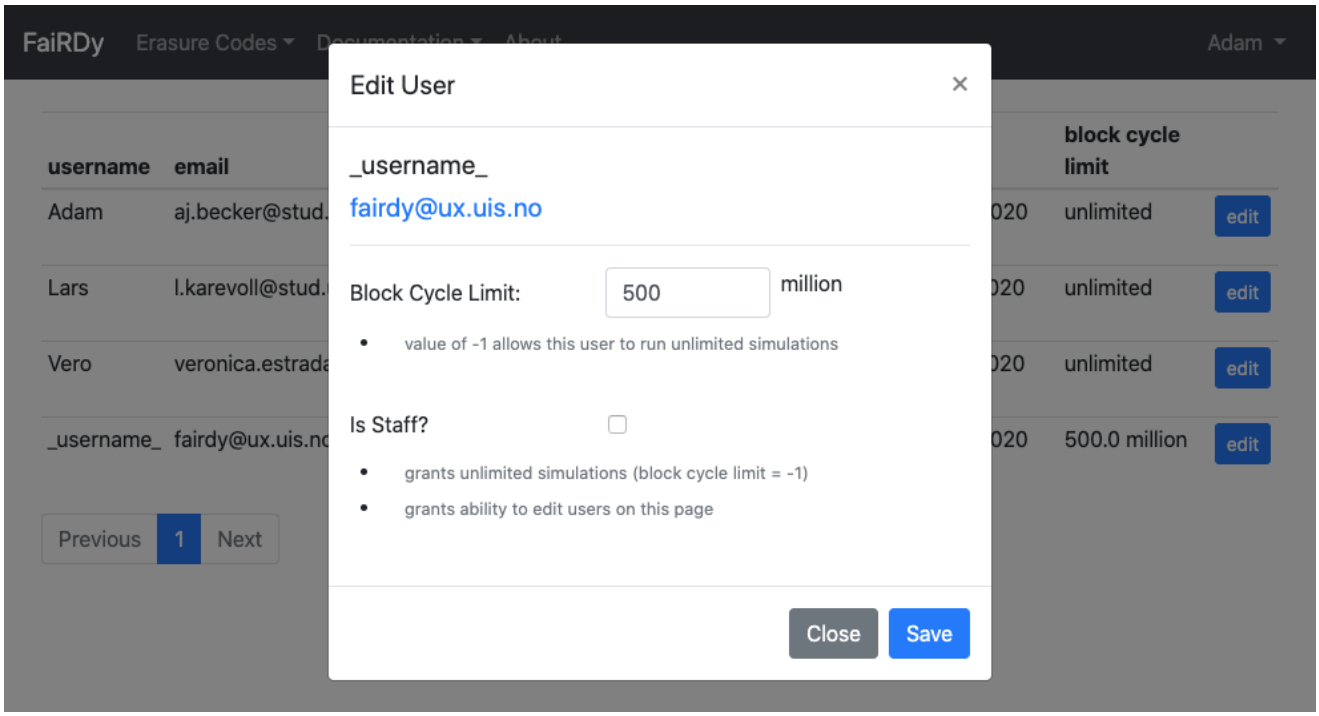
is staff?	determines whether the user as access to this user administration page
email valid?	whether the user has followed the linke in their email and validated the account
pw locked?	the user has attempted to reset their password (forgot password), and the process has not been completed
block cycle limit	limit on the number of simulations the user can run on a one month rolling basis

Block Cycle Limit

All new accounts have a *Block Cycle Limit*, which limits the amount of simulations the account can run on a one month rolling basis. Block Cycle Limit is calculated as the size of the storage system (in blocks) multiplied by the number of cycles. New accounts have a default Block Cycle Limit of 500 million. The default limit allows up to 100 simulations with up to 50,000 total blocks in each system, 100 cycles each. **Enter -1 to give a user an unlimited block cycle limit.** This is an integer field with a max. value of 2,147,483,647. Enter your choice in millions, it will be automatically multiplied by the web framework.

Edit User Dialog

To access the edit user dialog simply click the 'edit' button on the user record.



From within the dialog it is possible to modify the user's block cycle limit and 'is staff' setting.