Designing a Security Management Layer for CDN Assets

Lukas Klingsbo

February 23, 2016

Abstract

This work presents a technique for incorporating Copy-on-Write in a high level system on top of an unaware persistent storage. The system in this case is a security management system, but the technique and conclusions drawn are general enough to be applicable to any system with vaguely similar consistency requirements. The scalability of the system is tested and the technique is evaluated and the technique is shown to be quite efficient (scalable to >6000 simultaneous active users per node), but a better solution is also suggested in the conclusion.

Keywords: CDN, Copy-on-Write, Eventual Consistency, MongoDB, Persistent Storage, Scalability, Software Security

Contents

1 Introduction

Developing large projects containing static content usually involves using a Content Distribution Network to be able to scale to a larger user base. The commercial Content Distribution Networks are usually fairly easy to use, the content that is to be used in a project is usually simply uploaded and then distributed over the globe when the public requests it. For secret content this can be a problem and an inconvenience, and that is what this thesis is about. This work examines ways of enforcing virtual access control on content and groups of content, in the form of containers and snapshots. A system was developed to make the underlying theory work in practice.

The research question that this report tries to answers is how and whether it is practically feasible to use Copy-on-Write for a high-level system like the one that is implemented and how it can effect the systems scalability.

TODO: Clarify problem definition and research questions

2 Background

This section gives the reader some background of the company which the thesis was done at and explanation of the core concepts which are exerted.

2.1 About Uprise

Uprise (formerly known as ESN) is a company based in Uppsala, Sweden. It is an EA studio focussing on creating great gaming experiences, which means that they are not focussed on the actual gameplay, which other EA studios like DICE are. Currently Uprise has a lot of focus on developing companion apps and a new form of menu systems.

2.2 Prior Work

2.2.1 Copy-on-Write

This work relies heavily on the Copy-on-Write principle, which was founded and used in the Mach kernel [1], as it can be used to efficiently create snapshots and help solving concurrency problems that otherwise can occur.

Copy-on-Write is used in for example virtual memory management systems [2], snapshot algorithms and as an optimisation technique for objects and types in several programming languages [3].

Its principle is that when processes or nodes share data in between each other, the data is not copied until one of the processes makes changes to it. This is an optimisation as the processes does not have to send or copy all of the related data that is in memory, rather they only have to send pointers to the data. After many Copy-on-Write's a complex tree structure can be built up, but optimisations can be done to simplify that structure [4].

TODO: Polish and extend

2.3 Related Terminology

In this section concepts and abbreviations, that are recurring throughout the paper, which the reader needs to be familiar with are explained.

2.3.1 Abbreviations

2.3.1.1 JPF

Java Path Finder - It was developed by NASA and in 2005 they released it under an open source licence, which made more people contribute to the project. JPF is usually used for doing model checking of concurrent programs to easily find for example race conditions and dead locks.

2.3.1.2 CDN

Content Distribution/Delivery Network - Replicates content to several servers, usually spread out geographically. Once a request is made, the network serves content from the server closest to the requester.

2.3.1.3 GUID

Global Unique Identifier - Usually in distributed environments normal incremented identifiers can not be used as there can be insertions on several nodes at the same time. A GUID is generated by a function that makes it impossible or extremely unlikely that the same identifier will be generated and used again.

2.3.2 Terms

2.3.2.1 Snapshot

A snapshot is a way to record the full state of a system at a specific time. The term comes from photography where a photo can be seen as the state of what the photo is of, at a certain time. Snapshots should not be confused with a full copy of a system, or part of a system, as full copies can be used as backups meanwhile snapshots are not very effective means of backup in the case of data corruption. It is not effective against data corruption as snapshots usually still refer to unchanged data that is still a part of the system [5].

2.3.2.2 Eventual Consistency

The expression eventual consistency is a term used for describing how data is processed. It is often used in the context of database systems. Eventual consistency

means that stored data is guaranteed to be consistent in any one moment, but at some point the data will eventually converge to being consistent [6].

The effect of this is that persistent storages, especially distributed databases, can operate faster as there is no need for locks and transactions for any operations. As no locks or transactions are used, operations on the same entities can be done before earlier operations have propagated to all nodes. This can mean that the system ends up in a state where there is a conflict that needs to be handled, for example two operations that wants to change the same data at once. To solve this the system needs to have defined which operation that will take precedence and this is usually called conflict or sibling resolution.

2.3.2.3 Perius

Perius is the name of the implementation that was made during this project. It is an anagram of Uprise and also a type of butterfly.

3 Model

The model for this work should show how the data can not be accessed or modified by unauthorized users and how the defined integrity of the data is always kept in the Perius system.

There could also be another relevant part included in the model, to show that content can not be accessed by unauthorized viewers once the content is uploaded to a CDN. But as that should already have been thoroughly checked by the CDN providers this work can focus solely on the internal users and content in the management system.

3.1 Related work

The model for this paper is based on the work that was done by Bell, D Elliott, LaPadula and Leonard J in their papers *Secure computer systems: Mathematical foundations* [7] and *Secure computer systems: A mathematical model* [8]. In these papers the foundation was laid for how to model computer systems to be able to analyse the security of them. Furthermore this paper was also inspired by Biba, *Integrity considerations for secure computer systems* [9], where many of the points made by him was taken into consideration when inspecting that the integrity of the data was always sound.

3.2 Approach

As this work only presents an informal model of how the system is designed it can not be regarded as a proof for the actual implementation of the system to be flawless. The model should how ever give a strong idea of the soundness of the design of the system.

The Copy-on-Write system is only used for files as all of the other objects in the system is basically just meta data and not classed as important, from an data integrity point of view, as the actual files themselves.

3.3 Elements of the Model

Set	**Elements**	**Semantics**
C	$c_0 \dots c_n$	Containers; folders in the virtual file system
F	$f_0 \dots f_n$	Files; files, images, videos
M	$m_0 \dots m_n$	Content; Meta-data for files
U	$u_0 \dots u_n$	Users; registered users in the system
A	$A[u_0, c_0] \dots A[u_n, c_n]$	Access matrix; describes what containers users have access to

3.4 Access rights

The a notation is meant to symbolise an access token, and if a exists in the requested entity of the matrix the corresponding user to that entity has full access to the corresponding object.

$$\begin{aligned} u \in U \text{ can read } c \in C &\Leftrightarrow u \in A[u, c] \\ u \in U \text{ can write } c \in C &\Leftrightarrow u \in A[u, c] \text{ and readonly } \notin c \\ u \in U \text{ can delete } m \in c &\Leftrightarrow u \in A[u, c] \text{ and readonly } \notin c \\ u \nexists U \text{ can delete } f \in F & \\ \forall c \in C, \quad \exists u \in U \mid a \in A[u, c] & \end{aligned} \tag{1}$$

3.5 Data integrity

A computer system or subsystem is defined as possessing the property of integrity if it behaves consistently according to a defined standard. This implies that a subsystem possessing the property of integrity does not guarantee an absolute behaviour of the system, but rather that it performs according to what its creator intended [9].

3.6 Initial Assumptions

To create an integrity model, some initial assumptions have to be made about what the correct behaviour of the system is, which the model then can be shown to follow. In this work unintentional behaviour as the result of data modification is the main concern, which could be used for sabotage or simply be the effect unintentional unfortunate race conditions etc.

3.7 Integrity Threats

According to Biba et. al [9] one can consider two threat sources, namely subsystem external and subsystem internal. The external sources could be another system calling the subsystem with faulty data or trying to make inaccurate calls to program functions, it could also be somebody trying to tamper with the exposed functions of the program. Threats that are internal could be a malicious part of the subsystem or simply an incorrect part of the subsystem, which does not behave according to specification.

In this work external threats are handled as threats that can occur from what has been exposed by the API (See Section 4.5.5) and internal threats as incor-

rect implementation. As the server and its system are assumed, safe malicious subsystems are not considered.

3.8 Consequences of modification

In this section the consequences of different operations are stated in an informal mathematical manner. These rules are what should be expected to be followed by the implementation in Section 4.

Action	**Semantics**
u, read(o)	User u reads object o
u, write(o)	User u writes object o
u, copy(o)	User u copies object o and the copy gets a new ID
u, change(o)	User u locally changes object o
u, modify(o', o)	User u globally modifies object o based on object o'
u, delete(o)	User u deletes object o
u, snapshot(o)	User u takes a snapshot of object o

$$\begin{aligned} &m' = u, read(m) \wedge \\ &x = f \in m' \wedge \\ &f' = u, change(x) \wedge \\ &u, write(f') \wedge \\ &u, modify(f', m) \Rightarrow f \in F \wedge f' \in F \end{aligned} \tag{2}$$

If a user wants to update a file in a content, the file is copied and the original is intact.

$$\begin{aligned} &m' = u, read(m) \wedge \\ &m'' = u, change(m') \wedge \\ &m''' = u, modify(m'', m) \Rightarrow \\ &m''' \in M \wedge m \notin M \end{aligned} \tag{3}$$

If a user reads content and then writes to it, the content is directly changed.

$$\begin{aligned} &c' = u, read(c) \wedge \\ &c'' = u, change(c') \wedge \\ &c''' = u, modify(c'', c) \Rightarrow \\ &c''' \in C \wedge c \notin C \end{aligned} \tag{4}$$

If a user reads a container and then writes to it, the container is directly changed.

$$\begin{aligned} &f' = f \in m \\ &m' = u, copy(m) \Rightarrow \\ &m' \neq m \wedge \\ &f' \in m' \wedge f' \in m \end{aligned} \tag{5}$$

If a user copies a content, the new content is not equals the old (because of its new ID), however they both refer to the same file.

$$\begin{aligned} s = u, snapshot(c') \Rightarrow \\ \forall c \in c', u, copy(c) \in s \wedge \\ \forall m \in c', u, copy(m) \in s \wedge \\ (\forall f \in c' \Rightarrow f \in s) \end{aligned} \tag{6}$$

If a user creates a snapshot of a container the full container tree is re-created with new ids and its content still refers to the same files.

$$\begin{aligned} f' = f \in (u, read(c)), \\ u, delete(c) \Rightarrow \\ c \notin C \wedge f' \in F \end{aligned} \tag{7}$$

If a user deletes content, the file referred to in the content is not deleted.

3.9 Interaction of Multiple Accesses

TODO: Rename consequence?
As a result of consequence 3 and 4 in Section 3.8 there can be race conditions where the last write wins. This could be solved by locks or transactions, but as updates are not based on each other, the trade-off for this inconsistency to scalability and speed is intentional. However as can also be seen in consequence 2, if a file is removed from a content that file is not removed from the set of files, even though its modification process is not the last one to write to the content. This is necessary as a file can be referenced from several content and the removal process can not atomically ensure that the file is not referred to by another content. This results in a lot of files in the system which are not referenced from anywhere. This can be solved by either making each file keep track of how many times it is referred to and also make it possible to check this number and delete the file in an atomic fashion.

An easier approach would be to have a garbage collector which locks parts of the system momentarily meanwhile it removes files which are not referenced.

There is also the chance to have content and containers that are not referenced from anywhere and result in being garbage in the system. This could happen as a result of consequence 7 in combination with any of the other consequences listed in Section 3.8. The basic idea is that an entity is deleted meanwhile another is created or modified to have that entity as its parent and thus creating a separate tree that can not be reached from the root node of the project. These type of inconveniences should also be cleaned up by the garbage collector.

3.10 JPF

TODO: Not sure where to put this section

JPF was used to test the idea and state transitioning of the model. It's a very simplified version of the real system that still contains all the important Copy-on-Write core concepts and the assumptions that have been made in the model. This simplified version was then used to automatically test for unwanted quirks. It is not a proof that the model works, but it is very exhaustive in its testing.

3.10.1 Entities

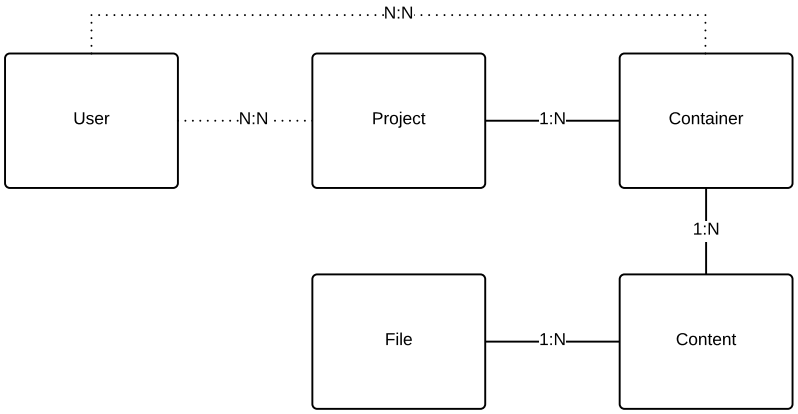

Figure 1: High Level Entity Relationships

Content
Content is meta data about a file and is stored in a container, it is a form of virtual file. The content can refer to for example an image, video or binary blob.

Project
A project is what is created to contain all content and containers related to a real project. Files can be changed within a project and the system can contain several projects and their virtual content are completely disjoint.

Container
A container is a virtual folder within a project which can contain content and other containers.

Snapshot
A snapshot is a read-only container from the state which the container the was in when the snapshot was created. A snapshot can not be updated and can only be deleted from the root of the snapshot. Snapshots are by default stored as siblings to the container which they were made from, but they can be contained by any container.

File
A file refers to an actual physical file. Files are stored in the database to make backup, deployment and migration easier.

User
A user is the structure that handles people who have been granted access to the system. Access to the system is handled by a separate service, like LDAP.

3.10.2 Execution

Java path finder was used to show that the model and plan of how to build the system was sound. The model was built in Java with the objective of being as reduced and simple as possible, without loosing any of the cases that needed to be covered by the model checker. As the users are mainly going to be handled by external systems they were not included in the model.

Each collection in the persistent storage was emulated by using the built-in ConcurrentHashMap type. Each client was represented by a thread and each action taken by the client was randomised. The id hashes which MongoDB is using for each entity was imported from the mongo-java-driver-2.13.3 and each object had its own id, generated in the same fashion as the real implementation is using, randomly generated by the ObjectId class to minimise collisions that is. Furthermore no locking or transactions were used and the threads were running fully concurrently, without any sleep statements.

ConcurrentHashMap had to be used in instead of the normal HashMap, as the normal HashMaps can't be iterated over concurrently.

JPF checked each permutation of states that the threads can end up in, the result of the run can be seen in Listing 1.

Listing 1: Results of JPF run

```
elapsed time:       14:26:53
states:             new=160853259,
                    visited=451102505,
                    backtracked=611955764,
                    end=21640
search:             maxDepth=380,
                    constraints=0
choice generators:  thread=160853255
                    (signal=0,
                    lock=3603938,
                    sharedRef=146989208,
                    threadApi=3,
                    reschedule=10260106),
                    data=0

heap:               new=676056850,
                    released=435060996,
                    maxLive=655,
                    gcCycles=523950061

instructions:       11917045758
max memory:         6256MB
loaded code:        classes=111,
                    methods=2179
```

3.11 Findings

TODO: Add more findings

3.11.1 Garbage Collection

For the system to reach the decided eventual consistency [10], garbage collection will be needed. As of the conclusion in Section 3.9 from Section 3.8 there will be

objects in the system that can not be reached and should therefore be cleaned up for them not to cause negative performance effects on the system.

3.11.2 Conflict Resolution

If the persistent storage would be distributed the different database nodes would have to have rules set up for conflict resolution as this model does not consider what to do if the same meta data is modified on the nodes before they have time to synchronise. The easiest way to handle this is to have the clocks on the nodes synchronised and timestamp each change and when synchronising the nodes only consider the newest data if there is a conflict. This will have the same result as concurrent modification with a single database node with this model, namely last write wins [11].

4 Implementation

This chapter deals with the second part of this work, namely the implementation of the Perius system based which is based on the model described in Section 3. As this work is not intended as an instructions manual, more information about how to deploy and use Perius can be found on `http://perius.se`. This chapter's focus is instead on how the decisions for the implementation were made, how well the system would scale and the resulting security implications.

4.1 Background

4.1.1 The current system

Today a system called battlebinary [12] is used for managing and uploading files, mostly images, to content delivery networks. The current system does not make use of the security features that the CDNs are offering, instead it uses a form of security by obscurity. When a file is uploaded to a CDN it is open for the public, but its filename is composed out of its original filename concatenated with a part of the MD5 hash of the content of the file, which makes it an extremely hard process to access the file on the CDN without access to the original file or a reference to the URI.

In the current system you can only upload a file once as there will be a collision in the upload otherwise, as the old and the new file will have the same MD5 hash, which is by design as duplicate content just wastes space on the CDN. The problem with this is that the current interface does not handle it very well as one file can not be shown or uploaded in two different places in the virtual filesystem.

4.1.2 Problem description

As the current system does not offer proper security measurements, is lacking a lot of features that is needed and does not scale very well, a new system should be developed. This work is about examining a way of implementing Copy-on-Write in a high level system like this, which should solve the scalability problem and make it possible to implement wanted features like snapshots, cloning and concurrent modifications of content.

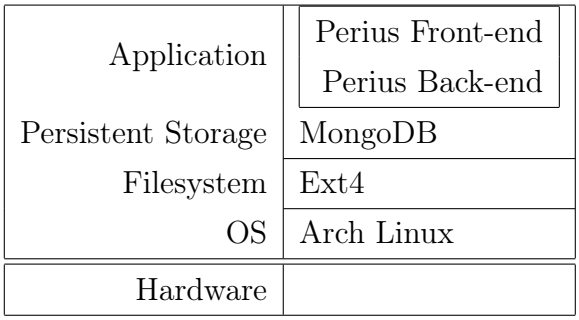

Figure 2: Software Stack

To clarify what is meant by Perius being high level software the stack in Figure 2 can be examined. Theoretically the Copy-on-Write features could be implemented in any of the separate parts of the stack to support the features that Perius has to achieve, in this work only the second highest layer is considered, the Perius back-end.

4.2 Related Technologies

4.2.1 React

React is a JavaScript library for building user interfaces. React uses both its own virtual DOM and the browser's, this makes it able to efficiently update dynamic web pages after a change of state through comparing the old virtual DOM with the resulting virtual DOM after the state change and then only update the browser's DOM according to the delta between the virtual DOMs [13]. React can be seen as the system for handling views in front-ends implementing a MVC (Model-View-Controller) architecture.

4.2.2 Reflux

Reflux [14] is an idea and a simple library of how to structure your application. It features a unidirectional dataflow (see Figure 3) which makes it more suitable, than for example Flux [15], when using a functional reactive programming style.

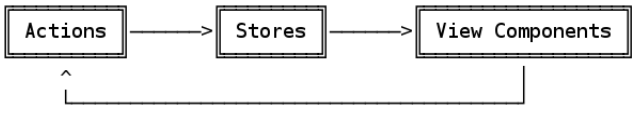

Figure 3: Reflux unidirectional dataflow

4.2.3 Scala

Scala is a multi-paradigm programming language. It most commonly runs on the JVM and compared to Java it supports most functional programming features at the same time as it supports object oriented programming [16].

4.2.4 REST

REST stands for representational state transfer, it is an architectural idea for writing stateless services. These services usually use URIs to identify specific resources and HTTP to modify or query these resources [17].

4.2.5 MongoDB

MongoDB is a document-oriented database which means that it does not have the concept of rows as normal relational databases has. Instead each entity in the database is stored as a document which is not fixed to a predefined table structure [18]. MongoDB lacks the support for joins to improve its possibility to scale, which can be a big down side to some applications containing the need for such logic.

4.3 Methods for determining implementation details

This chapter introduces the different methods used to determine how the new system should be implemented, which DBMS it should use and how the estimation of long term scaling was done.

4.4 Snapshot functionality

TODO: Structure to compare snapshot systems and conclude how Perius snapshot system was designed

4.4.1 Copy-on-Write

To efficiently create snapshots of a system Copy-on-Write can be used to make it possible to create snapshots in O(1) [19], this is due to the fact that to create a snapshot in a system using Copy-on-Write you only need to reference the current nodes in the tree and make sure that they are not removed, see Figure 4.

As the persistent storage, used in this implementation (Section 4.5.4), does not implement transactions or locks a lot of different problems can occur when several clients are working on the same data set at the same time. Such problems could be

race conditions and determining the happened-before relation. In this work this problem is solved by implementing Copy-on-Write. TODO: Move last paragraph

4.4.2 Full Copy

Full copy or deep copy, as opposed to copy-on-write, is a copy where everything is copied directly and not only when an object is changed. This is easier to implement but is in most cases more inefficient as more disk space will have to be used and if used with for example certain tree structures the part of the tree that needs to be copied will have to be traversed.

4.4.3 Comparison of Copy-on-Write system implementations

4.4.3.1 BTRFS

Btrfs is a B-tree file system for Linux which makes use of Copy-on-Write to make it able to do efficient writeable snapshots and clones. It also supports cloning of subtrees without having to actually copy the whole subtree, this is due to the Copy-on-Write effect. As several nodes in the tree can refer to the same node each node keeps track of how many parents it has by a reference counter so that the node can be deallocated once the node does not have any parents any more. The reference counter is not stored in the nodes themselves but rather in a separate data structure so that a nodes counter can be modified without modifying the node itself and therefore eludes the Copy-on-Write that would have to occur.

(a) Tree T_p (b) T_p cloned to T_q

Figure 4: Cloning mechanism of Btrfs [19]

4.4.3.2 Mach kernel

In the mid 80's when the development of the Mach kernel started, there was problems with that physically copying memory was too slow. Too minimise the copying of memory, Copy-on-Write was implemented. It was implemented so that virtual copy operations could be done and so that tasks could share read-write memory [20].

TODO: Insert more systems here and add a comparison of the systems

4.4.4 Snapshot functionality of Perius

In Perius snapshots and clones are not taken in the fashion which Btrfs uses, which can be seen in Figure 4. As Perius does not have the tree structure pre-built and each node is instead stored in a flat storage space, such operation would be too computationally expensive as trees would have to be merged when collisions occur, due to the non-blocking nature of the application. Instead this implementation makes a full copy of the meta-data of the tree, but still refers to the same binary files until they are changed, which results in the creation of a new node.

TODO: Relate section more to comparison

4.5 Resulting system

4.5.1 Perius

Perius is the implementation that was done to solve the problem at hand at Uprise. Perius has a back-end written in Scala and a front-end written in Javascript (ES6), but they are both interchangeable. The back-end has a REST API running, which is how the front-end communicates with the back-end.

TODO: Picture of the newest front-end

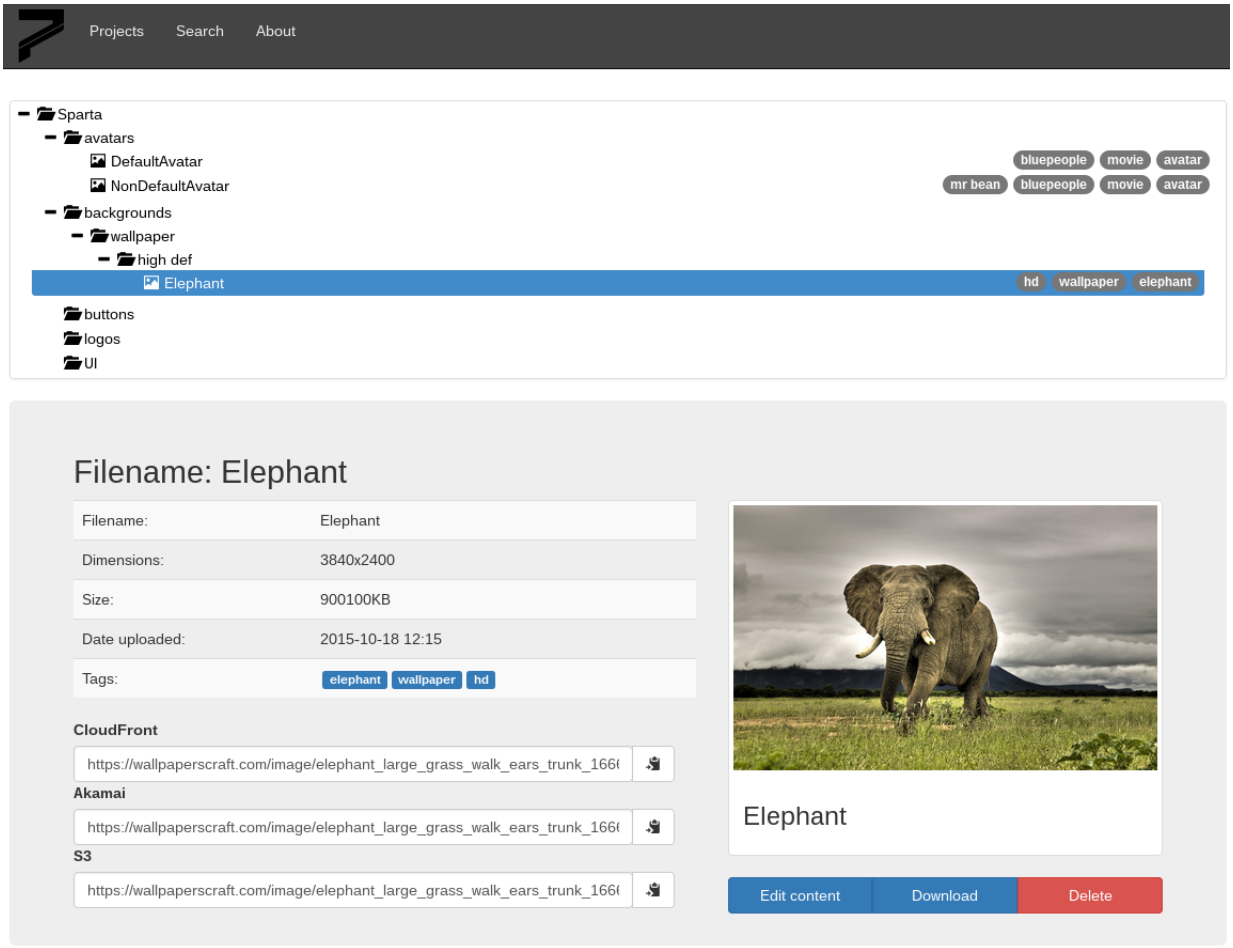

Figure 5: Front-end [19]

The service features a virtual file structure over the assets that has been stored, snapshots, security management of whole containers as well as individual files, audit and auth logging, multi project support and a modular design for persistent storage.

The front-end is written in ES6 with React and Reflux, and the styling is done with the help of Bootstrap.

4.5.2 Run-time Complexity of Operations

The runtime complexities of the operations in this section are quite intuitive, that is why they are only explained, and not proven to fulfil the complexities which are claimed.

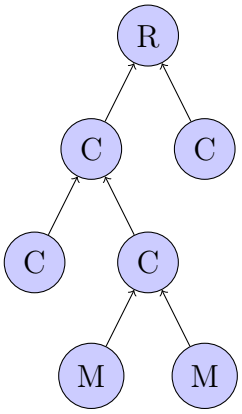

Figure 6: Project Tree
R - RootContainer, C - Container, M - Content

In the user interface the stored data is always represented and worked on as a tree, or a forest if all projects are considered. For a very small project that tree could look like the example in Figure 6. This is however not what the actual storage of the data looks like.

Type	Id	Parent Id
RootContainer	1	1
Container	2	1
Container	3	1
Container	4	2
Container	5	2
Content	6	5
Content	7	5

Figure 7: Simplified Representation of the Actual Storage

The actual storage of the data is represented by documents in a flat document storage, which is in this case MongoDB. A very simplified version of how the data is stored can be seen in Figure 7. One important point that differs in the example and in the actual storage is that GUIDs are used for identifiers and not incremented numbers, as that would not work in distributed environments.

As the data is not stored as a tree, but rather as separate documents loosely referring to each other, the run-time complexity is not obvious and that is what is discussed in this section.

GET single entity
When the back-end is handed a GET request it looks at what type it is and returns the document with the correct id from the corresponding collection. As the collections are hash indexed on id this operation can be made in constant time, O(1) and thus $\theta(1)$.

GET project tree
When requesting a project the full project tree with all entities except files has to be built as the entities not are stored as a tree in MongoDB. The tree is built layer by layer by fetching content with the same parent id, starting from the root of the project. Parent id's are also indexed by a hash index which makes it possible to get all nodes with the same parent id in constant time. This has to be done for every level with a different parent in the tree which makes the upper bound O(n) if all of the subtrees only have one node each. The tight bound can not be calculated as the trees balance is not determined by an algorithm but rather by a user, but from observation one could argue that the trees usually have several nodes for each parent node in the tree, which then would give a logarithmic tight bound, $\theta(log(n))$.

POST/Create entity
Creating a new entity in the system is a constant time operation for containers and content as they only have to be inserted as new documents in MongoDB, which is a O(1) operation [**?**]. Even though the insertion itself has constant run-time the index needs to be updated or some times even rebuilt, which can slow the time before the document is accessible in constant time. When inserting files they need to be chunked in to pieces and inserted into GridFS, this results in n/s insertions where n is the size of the file and s is the chunk size. The insertion of files can then be regarded as O(n) if the size of the file is regarded, but if n is regarded as the number of files in the system the upper bound is constant as the insertion for files is made in the same way as for other documents.

Snapshot creation
In the creation of a snapshot all meta data in the tree that the snapshot is taken of is duplicated, meanwhile the files remain the same, this results in n insertions where n is the number of containers and content entities in the subtree. A snapshot insertions runtime is therefore $\theta(n)$ and O(n).

Delete entity
Deleting an entity is a constant time operation when that entity does not have any children. When the entity has children all those children has to be deleted too

which will be done in the same manner as the project tree is built and therefore results in the same runtime, O(n) as an upper bound, where n is the number of entities in the subtree.

PUT/Update entity
The update operation in MongoDB completely replaces a document with another in regards to their unique identifier. This could be seen as a deletion of the old document and an insertion of the new document, but with the same id, which makes it unnecessary to update the index for the unique id (other indices might have to be updated though). As both deletion and insertion are constant time operations, update is also O(1).

4.5.3 Copy-on-Write

TODO: Rename section and move

This implementation is far from as efficient as the other Copy-on-Write systems described in Section 4.4.1 in most aspects, but more efficient in some. As the implementation is built upon MongoDB as persistent storage and not a pure tree structure, single nodes can be fetched in O(1) but when querying for subtrees they need to be built first, which takes O(log(n)), where n is the number of nodes in the subtree.

4.5.4 Persistent storage

4.5.4.1 MongoDB

MongoDB was chosen as the persistent storage because of its quick lookups and because of its internal storage format called BSON, which is very similar to JSON which the API is using. As the formats are similar, the process of marshalling and unmarshalling becomes quite easy between the core code, MongoDB instance and REST interface. The second reason was that if the system needs to scale in the future it is very easy to distribute MongoDB and if needed the system can easily be migrated to Reactive Mongo, which is an asynchronous and non-blocking driver for MongoDB and can therefore make the system scale even further [21].

All files are also stored directly in MongoDB with the help of GridFS. GridFS chunks the files according to the size limit of MongoDB objects, which is currently 4MB. The advantage of this is that backups of the Perius state is easily done through a database backup, no separate files needs to be backed up. Another advantage that is given by this is that you can retrieve specific ranges of a file, although that advantage is not needed in the Perius implementation. For scalability this could be used to retrieve different parts of a file from different servers,

normal load balancing would probably work better in a system like Perius where no extremely large files are expected to be stored.

The disadvantage of using the GridFS approach is that when using a non-distributed database it is slower to read and write to the database than reading or writing directly to the filesystem. Another disadvantage is that to access the files it is needed to go through the database layer in some way, instead of accessing the filesystem.

4.5.5 API

In this work a RESTful API was implemented and used for back-end ⇔ front-end communication.

REST was chosen as only basic CRUD operations needs to be performed and because the BSON format which is used in MongoDB is almost identical [22] to the standardised JSON format which is usually used by RESTful services [23].

4.5.5.1 REST Endpoints

For the front-end to communicate with the back-end, a RESTful service is implemented. The following endpoints were configured:

- projects

  GET - list all projects

  POST - create new project

- projects/{id}

  GET - get specific project

  PUT - update existing project

  DELETE - delete existing project

- projects/{id}/content

  POST - create new content in a specific project

- projects/{id}/content/{id}

  GET - get specific content in a specific project

  PUT - update existing content in a specific project

  DELETE - delete existing content in a specific project

- projects/{id}/snapshots

  POST - create new snapshot in a specific project

- projects/{id}/containers

  POST - create new container in a specific project

- projects/{id}/containers/{id}

  GET - get specific container in a specific project

  PUT - update existing container

  DELETE - delete existing container

As can be seen several expected endpoints are missing, this is intentional as the operations missing can be performed in a more efficient way. Such endpoint is for example *GET projects/{id}/containers* as all containers exist in *GET projects/{id}* and the interface should present a file structure where both content and containers are shown.

4.6 Load Testing

Wrk was used to load test the back-end. It is a multi-threaded benchmarking tool for HTTP which can create large loads. The testing was done locally on a virtual server having 6 CPU cores and 48GB of RAM. The results that will be analysed are latency and throughput of the server as that will give a fair judgement of how well the system can perform.

4.6.1 GET requests

To be able to determine what is the bottleneck of the back-end three different types of *GET requests* are performed. The first one tries to maximise the number of requests that spray-can (HTTP server) can handle with the given specifications and therefore the server just returns 200 (OK). The second request requests the simplest API call which involves getting all currently stored projects from the MongoDB instance. As this involves the server answering with some payload, the third request is simply the same amount of static payload as the second request but without involving any connections to MongoDB.

The typical requests with Wrk in this work will look similar to this:

```
wrk -t100 -c100 -d10s http://perius:8000/ok
```

where -t100 means that it uses 100 threads, -c100 that it emulates 100 clients requesting over and over and -d100s is the amount of time that the load test will run, in this case 100 seconds.

Listing 2: Result of OK requests

```
wrk -t100 -c100 -d10s http://perius:8000/ok

Running 2m test @ http://perius:8000/ok
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.22ms    3.20ms 166.78ms   97.75%
    Req/Sec     1.04k   152.50     2.25k    79.18%
  10345746 requests in 1.67m, 1.69GB read
Requests/sec: 103353.85
Transfer/sec:     17.25MB
```

Listing 3: Result of MongoDB requests

```
wrk -t100 -c100 -d10s http://perius:8000/projects

Running 2m test @ http://perius:8000/projects
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    14.52ms   13.34ms 375.51ms   99.31%
    Req/Sec    72.85      6.91   232.00     60.20%
  726351 requests in 1.67m, 196.03MB read
Requests/sec:   7256.38
Transfer/sec:      1.96MB
```

Listing 4: Result of static text requests

```
wrk -t100 -c100 -d10s http://perius:8000/static

Running 2m test @ http://perius:8000/static
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     6.42ms   28.99ms 290.37ms   96.26%
    Req/Sec     0.94k   203.30     2.00k    83.83%
  9199737 requests in 1.67m, 2.19GB read
Requests/sec:  91902.48
Transfer/sec:     22.44MB
```

Listing 2 shows that the HTTP server can answer about 100K requests/s when not involving any payload other than status code 200 (OK). When comparing that to the request which involved the server responding with a small (100 Byte) payload (Listing 4 one can see that it is about 10% slower to send some more data, ~90K requests/s. However when comparing that to the requests that needed database access it is obvious that the HTTP server can handle all the load that it needs to, as Listing 3 shows that to fetch all documents in a collection (in this case only one) is dramatically slower, the throughput was only ~7000 requests/s.

These are very simple requests, to evaluate how much load the application can handle in its current state a more complex but common request has to be analysed. The most complicated and still common request that system is receiving is requests of a full project tree. This request is computationally expensive as the tree is not stored directly in the database but has to be built from the id and parent id of each container and content, a simulated project tree of similar size to the ones stored in the current management system reaches around 25KB in uncompressed

size.

Listing 5: Result of project tree requests

```
wrk -t10 -c10 -d10s http://perius:8000/projects/56a5f...

Running 10s test @ http://perius:8000/projects/56a5f...
  10 threads and 10 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   247.54ms   21.64ms 323.78ms   96.76%
    Req/Sec     3.80      1.22    20.00     99.00%
  401 requests in 10.07s, 9.67MB read
Requests/sec:     39.82
Transfer/sec:      0.96MB
```

The result in Listing 5 shows an average of 40 requests/s, which means this is the real bottleneck in the application. There are two solutions to fix this bottleneck, as the project tree wont be modified nearly as often as it is requested the first solution would be to cache the results of the tree requests and invalidate those caches when the tree is modified. The second solution would be to make a more computationally feasible way of building and fetching the whole tree from the database. This is left to implement if the need comes for it in the future. Even though this massive bottleneck is present, it wont be a problem in a small scale production environment as the tree is fetched approximately once every 30 seconds by active users, which means that the application still could support at least 1200 very active users.

Listing 6: Result of cached project tree requests

```
wrk -t100 -c100 -d100s
  http://perius:8000/projects/cached/56a5f...

Running 2m test @
  http://perius:8000/projects/cached/56a5f...
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     1.46ms    3.62ms 180.68ms   97.87%
    Req/Sec   838.58    120.95     1.74k    79.12%
  8343565 requests in 1.67m, 17.03GB read
Requests/sec:  83352.77
Transfer/sec:    174.17MB
```

With caching turned on, on a single back-end, the application can theoretically

support several million active users which are not posting any content, this is simulated in Listing 6.

Another simple optimisation that was added was indices on the parent ids, which are heavily queried when building the project trees.

Listing 7: Result of indexed project tree requests

```
wrk -t100 -c100 -d100s
  http://perius:8000/projects/56a5f...

Running 10s test @ http://perius:8000/projects/56a...
  100 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency    47.27ms    3.03ms  86.54ms   93.79%
    Req/Sec    21.12      3.50    60.00     87.51%
  21271 requests in 10.10s, 9.76MB read
Requests/sec:   2105.82
Transfer/sec:      0.97MB
```

As the project tree is built by querying documents parent id an optimisation that could be done was to create an index of the documents parent ids. This drastically improved the size of the tree that the server was able to handle within the time out, going from being able to handle 4000 documents in one tree to at least 50000 documents, which the browser instead will have problems displaying, represented in the UI, without lag.

After adding indices, the building of the project tree was able to finish 50 times faster and the server was able to serve 2100 requests/s, see Listing 7. If using the same approximated average that was used in connection to Listing 5 (30 requests/s), the maximum number of clients that one node can handle will be >60000, without taking the web servers limits or content posting into consideration.

4.6.2 POST requests

Two different POST requests will be tested, creation of containers and creation of content together with file uploads.

Listing 8: Result of container creation

```
wrk -c24 -t12 -d4s -s post.lua
  http://perius:8000/projects/56ab.../containers

Running 4s test @
  http://perius:8000/projects/56ab.../containers
  12 threads and 24 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency     5.92ms    1.25ms  19.58ms   91.67%
    Req/Sec   339.86     76.68     1.28k    95.65%
  16357 requests in 4.10s, 5.04MB read
Requests/sec:   3990.12
Transfer/sec:      1.23MB
```

As can be seen in Listing 8 the server can handle around 4000 container POST requests per second. The problem that this test resulted in was not the amount of containers that could be posted, but rather loading all those containers in the interface or API afterwards. When requesting the project that the containers were posted to the web server timed out the request, as it took too long for the back-end to build the project tree. In this work, the web server wont be configured to accept longer data processing time, as trees as large as this for a single project wont be used in production.

File uploads was not supported by WRK and therefore Apache Bench was used. As there is a lot of difference depending on the hard drive and file system when benchmarking uploads which are greater in size, which images usually are, no real conclusions can be drawn from this benchmark. It could perform about 200 POSTS/second with an image of 200KB, but it's unsure if the server could keep up that pace forever. If following the same example as earlier in the section with one POST per user every 30 seconds that would make the system able to serve 6000 active users.

4.7 Security of the system

4.7.1 Authorization

The authorization of users in the system is currently being handled by LDAP [24], as Perius is mainly focussed at being deployed in internal networks which usually has an LDAP service enabled. LDAP also makes it easier for the user to login as no separate account is needed for Perius and thus the user can use the same account as for all the LDAP connected services on the internal network.

4.7.2 Audit logs

Audit logs are used in the system to keep track of which users that perform which actions. In the event that somebody, consciously or unconsciously, are leaking private CDN content from within the organisation using Perius the audit trail can be used to find which LDAP account that was responsible for the leak.

4.7.3 CDN Connections

When uploading files to the CDN networks, their TLS/SSL protected APIs are used to ensure that no data leaks through packet sniffing etc. The files are grouped corresponding to their parent containers in Perius, which makes it possible to handle the security settings for all files in a group at once, without having to manually traverse through the tree in Perius.

4.7.3.1 Serving private content

There are three different ways, that this implementation make use of, to make sure that the CDNs serve content in a private manner. The first two are signed URLs and signed cookies, and the third is a mixture between one of the first two together with a range of IP addresses. [25] These mechanisms are needed, as explained in Section 4.1.1, to ensure that assets are not leaked if an attacker manages to find out the URL for the asset.

Signed URLs

Signed URLs work by writing a policy statement that specifies the restrictions that should apply for the asset the URL is referring to. There are two different types of these policies; canned and custom. In the canned policy there is only an option to specify the date for when the URL is no longer valid. In the custom policy it is possible to also specify the date when the asset should be made available, ranges of IP addresses (which is discussed more in a later paragraph) and inclusion of the base64 version of the policy in the URL [26]. For custom policies it is also possible

to reuse the policy and have it refer to multiple assets.

Signed cookies
Signed cookies work very similarly to signed URLs, the same type of canned and custom policies can be set, except for the policy to include the base64 version of the policy in the URL. Signed cookies are to be used when the URL should remain static even though the policies change [27].

IP range restriction
The IP range restriction is presented as a third option in Perius, even though it in fact is just a part of a Signed Cookie or URL policy. The IP range restriction makes it possible to restrict which IP addresses that can access the asset, this option can be fitting when for example an office or companies internet access is based on a set of public static IP addresses and thus restricts the content from ever being accessed outside of that controlled network.

4.8 Findings

4.8.1 Scalability

When using the ReactiveMongo driver [21], which is asynchronous and non-blocking, the application has no limits of how much load and users it can handle as the hardware and nodes can be scaled up linearly when needed. With Cashbah [28], which is used with the current implementation, it is harder to scale to the enormous amounts of load which ReactiveMongo can support as Casbah is synchronous and has blocking IO. For this work the kind of scalability which is offered by ReactiveMongo is not needed as the load will not reach the peak, as discussed in Section 4.6.1, for what Casbah can handle on a single server.

4.8.2 Security

The greatest security improvement of Perius compared to its predecessor Battlebinary is not the security of the system itself but rather how Perius handles the contents security settings on the CDN providers. Perius uses several of the security features for private content that the CDN providers offer (see Section 4.7.3.1) to make sure that the content only is accessible from where it should be. In Battlebinary the content was secured by adding a part of the hash of the file (see Section 4.1.1) to the filename that it was uploaded with, which resulted in a URL on the CDN that was not guessable but if the link somehow leaked the content that the link referred to would be open to anyone.

5 Discussion

5.1 Persistent Storage

More research could have been done in the choosing of persistent storage, as mentioned in *"NoSQL: Moving from MapReduce Batch Jobs to Event-Driven Data Collection"* [10] many applications that choose NoSQL databases as their persistent storage actually don't need it. This is most likely true for Perius too, it would have been efficient enough generating the project trees from an indexed SQL database with foreign keys. On the other hand it was quite nice having the BSON documents for insertion as they were so similar the accepted JSON format from the REST service. As Perius most common operations involve modifying the project tree, a graph database should have been researched too.

5.2 Copy-on-Write's effect on Scalability

As no comparison with solutions that were not Copy-on-Write based were made, no scientific conclusions can be drawn from how its effect was on the scalability of the system. However some intuitively drawn conclusions can be suggested. If comparing to a system which uses a SQL database the Copy-on-write system would have an easier time to scale on the width as no global locks or transactions has to be used, which with high probability would slow down the system, especially when widely distributed. The gain of having the locks or transactions would be that the system would always be in a fully consistent state. The loss would be that all of the database nodes would need to constantly have to be fully aware of the state of the other nodes.

A more interesting solution would be to not implement any Copy-on-Write system and simply use a persistent storage that scales with eventual consistency. That implementation would most likely be more efficient than the current solution, with MongoDB as a persistent storage and the Copy-on-Write handling in the back-end code, as a lot of optimisations could be done in the database engine instead.

5.3 GET/POST Estimation

The estimation of 30 seconds/action that was made in Section 4.6 was only based on observation, no logging of a system used in production (as no such system existed) could be made to back up that estimation. In the future those numbers can easily be recalculated, when there are enough metrics from a fully deployed version of Perius.

6 Summary

6.1 Conclusions

The goal of this thesis was to see whether it was feasible to use Copy-on-Write in a high level application (See Figure 2). As the implementation (see Section 4) made as a part of this thesis project is already replacing its predecessor the simple conclusion is that it definitely is possible. In the beginning of the project thoughts were on having every element being operated upon in a copy-on-write fashion but this was later narrowed down to only have the most important part, the files, as Copy-on-Write. This was due to that the conclusion that it did not matter if the other elements were resolved in a last-write-wins manner when modified.

If this application would be distributed with several persistent storage nodes like MongoDB, the application would not always be in a global consistent state as there would not be any global locks. This could theoretically cause some inconvenience for the user but in all real world tests no users have noticed it. The theoretical inconvenience for the user was a trade-off made so that the application could scale on the width almost endlessly, especially if the issue with building the project trees is (as mentioned in Section 4.6) solved.

The positive effects of deploying Perius instead of its predecessor was not only about scaling, it also features a more secure way of handling private assets (Section 4.1.1, 4.7.3.1) and reduces the administration needed to control the security settings of large groups of assets (Section 4.7.3) at the same time.

The conclusion that can be drawn from implementing Copy-on-Write as high up in the stack as it was in Perius is that the code would have to be largely rewritten for every application that wants to use a similar approach. The solution to this could be to use Copy-on-Write in the persistent storage instead of in the actual application, which also efficiently could handle conflict resolution and every improvement made to it could be shared for every system that uses the same type of system architecture and underlying persistent storage. On the other hand, by having all the Copy-on-Write specific code in the back-end instead of the persistent storage the choices of storage becomes wider as the database does not have to implement the named things. This could be positive when deploying Perius in environments with high requirements for the number of users that the system should be able to handle as the level of consistency needed could be fully tweaked to make a trade off for how many users the application would work for.

6.2 Future work

6.2.1 Access Control

Full access control was not implemented according to the model described in 3, it was only implemented to check whether a user should have access to the system as a whole or not, the implementation did not set or check any specific access rights to certain contents or containers.

6.2.2 Front-end Refactorization

The application could be made substantially more efficient by rewriting the front-end to update itself according to the REST response after a modification of the project tree, instead of re-fetching the full project tree every time a change is made. The front-end should also be fixed to properly follow the Reflux streamlining.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," 1986.

[2] J. M. Smith and G. Q. Maguire Jr, "Effects of copy-on-write memory management on the response time of unix fork operations," *Computing Systems*, vol. 1, no. 3, pp. 255–278, 1988.

[3] R. G. White, "Copy-on-write objects for c++," *The C Users Journal*, 1991.

[4] F. J. T. Fábrega, F. Javier, and J. D. Guttman, "Copy on write," 1995.

[5] J. Guthrie, "Method and system for taking a data snapshot," July 8 2003. US Patent App. 10/616,411.

[6] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[7] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations," tech. rep., DTIC Document, 1973.

[8] L. J. LaPadula and D. E. Bell, "Secure computer systems: A mathematical model," tech. rep., Technical Report 2547, 1996.

[9] K. J. Biba, "Integrity considerations for secure computer systems," tech. rep., DTIC Document, 1977.

[10] L. Klingsbo, "Nosql: Moving from mapreduce batch jobs to event-driven data collection," 2015.

[11] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, pp. 180–209, 1979.

[12] "Github - battlelog/battlebinary," Uprise, September 2015. `http://curl.haxx.se/docs/faq.html#What_is_cURL`, accessed 2015-10-15.

[13] A. React, "Javascript library for building user interfaces," 2014.

[14] "Github - refluxjs," Reflux, October 2015. `https://github.com/reflux/refluxjs`, accessed 2015-11-24.

[15] C. Gackenheimer, "Introducing flux: An application architecture for react," in *Introduction to React*, pp. 87–106, Springer, 2015.

[16] M. Odersky, P. Altherr, V. Cremet, B. Emir, S. Micheloud, N. Mihaylov, M. Schinz, E. Stenman, and M. Zenger, "The scala language specification," 2004.

[17] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[18] K. Chodorow, *MongoDB: the definitive guide*. " O'Reilly Media, Inc.", 2013.

[19] O. Rodeh, J. Bacik, and C. Mason, "Btrfs: The linux b-tree filesystem," *ACM Transactions on Storage (TOS)*, vol. 9, no. 3, p. 9, 2013.

[20] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young, "Mach: A new kernel foundation for unix development," 1986.

[21] R. Haddock, "Intelligent internet system with adaptive user interface providing one-step access to knowledge," Mar. 14 2014. US Patent App. 14/212,654.

[22] D. Tomaszuk, "Document-oriented triple store based on rdf/json," *Studies in Logic, Grammar and Rhetoric,(22 (35))*, p. 130, 2010.

[23] L. Richardson and S. Ruby, *RESTful web services*. " O'Reilly Media, Inc.", 2008.

[24] T. A. Howes, M. C. Smith, and G. S. Good, *Understanding and deploying LDAP directory services*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[25] "Serving private content through cloudfront," Amazon, September 2015. `http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/PrivateContent.html`, accessed 2016-02-03.

[26] "Using signed urls," Amazon, September 2015. `http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/private-content-signed-urls.html`, accessed 2016-02-03.

[27] "Using signed cookies," Amazon, September 2015. `http://docs.aws.amazon.com/AmazonCloudFront/latest/DeveloperGuide/private-content-signed-cookies.html`, accessed 2016-02-03.

[28] T. Alexandre, *Scala for Java Developers*. Packt, 2014.