Access Control for Content Distribution Network Assets

Klingsbo, Lukas

June 14, 2016

Abstract

This work presents a technique for implementing Copy-on-Write on top of a persistent storage. The method was developed from the needs of 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 security management system is tested and the technique is evaluated with load testing and model checking. The implementation is shown to be scalable for up to >6.000 simultaneous active users per node, but based on our experience a better solution is suggested in the conclusion.

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

Contents

1 Introduction

The development of large projects using static content usually involves the use of a Content Distribution Network to be able to scale to a larger user base. Commercial Content Distribution Networks are usually fairly easy to use. The content that is to be used in a project is uploaded and then distributed over the globe when a client is trying to access the content. For secret content this can be a problem and an inconvenience due to the lack of tools available to handle groups of content and their security settings, and that is what this thesis is about.

When looking at the game industry, secret content could be anything from a picture related to a yet unreleased game to an internal video only intended for employees. Some secret content needs to stay secret and some content needs to get its security restraints changed during time. For example when a game is in a closed alpha stage, the content should only be accessible by the people working on it. But when that game later moves into an open beta phase (open for public), the content needs to be publicly accessible.

As large projects most often are divided into parts, a concept called containers is introduced in this work. Containers can contain more containers and content, and when modifying the security settings for a container it is applied recursively to all its content and sub-containers.

This work examines a way of enforcing access control on content and groups of content, in the form of containers and snapshots. By being able to control the secrecy settings on a container instead of separately modifying the settings for each individual content it becomes easier to release content to be accessible to the public or different groups during the progression of the project. To manage different versions or releases of the project, the snapshot functionality can be used to handle access control by taking a snapshot with the correct security settings at each release.

A system was developed using the Copy-on-Write principle to improve scalability and make it a lighter operation to take snapshots of parts of the system.

The research question that this report tries to answer is how and whether it is practically feasible to use Copy-on-Write for a high-level storage system and how it can affect the scalability of the system.

2 Background

This section gives the reader some background of the company where the thesis was written and the implementation was made. It also gives some brief explanations of the core concepts which are used throughout this work.

2.1 About Uprise

Uprise (formerly known as ESN) is a company based in Uppsala, Sweden. It is an EA studio focusing on creating great gaming experiences, which means that they are not focused on the actual gameplay, in contrast to other EA studios like DICE. Historically Uprise has developed for example an application called BattleLog, which is a companion for the BattleField game series. What Uprise is currently working on is unfortunately classified.

2.2 Prior Work

The *Prior Work* section goes through the related work that has already been made within the area and which this thesis is based on.

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 do 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-Writes a complex tree structure can be built up, but optimisations can be done to simplify that structure [4].

2.2.2 Snapshots

Several filesystems [5] [6] use snapshot techniques to make it possible for a user or system to roll back the system's state, to a state that the filesystem had at the time of snapshot creation. This can be useful for example when having a stable system and then doing a system update, if the system does not respond well to

the update the filesystem can be rolled back to its previous stable state, provided that a snapshot was created before the update procedure.

This work has taken a lot inspiration from the way the BTRFS filesystem [5] handles snapshots, which is further explained in Section 4.4.3.1.

2.3 Related Terminology

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

2.3.1 Abbreviations

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.

GUID

Global Unique Identifier - 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.

JPF

Java Path Finder was developed by NASA and in 2005 they released it under an open source license, 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 deadlocks.

2.3.2 Terms

Eventual Consistency

Eventual consistency [7] is an informal guarantee that defines how the consistency of stored data can behave. In a system defined as using eventual consistency, access to stored data is not guaranteed to return the result of the latest modification. At some point, if modifications to the stored data stop, access to a data item will return the resulting data of the last modification of that item. It is often used in the context of database systems.

The effect of the lack of a strong consistency requirement is that persistent storages, especially distributed databases, can operate faster since there is no need for locks or transactions. This results in that operations on the same items can be made before earlier operations have propagated to all the distributed nodes.

A system using eventual consistency can end up in a state where there is a conflict that needs to be handled. If for example two concurrent operations performed on two different servers changes the same data item, a conflict of how the resulting item should look like will occur when synchronising these

servers. To solve the conflict the system needs to have a defined set of rules to be able to determine which operation that will take precedence, or how the concurrent operations are mixed, which is usually called conflict or sibling resolution.

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.

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. Snapshots are not effective against data corruption as snapshots usually still refer to unchanged data that is still a part of the system [8].

3 Model

This section describes how different operations can be expected to effect the data in the system. The model for this work also shows how the data can only be accessed or modified by authorized users and how the defined integrity of the data is always kept.

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* [9] and *Secure computer systems: A mathematical model* [10]. 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* [11], 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 however give a strong idea of the soundness of the design of the system.

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

3.3 Sets and Elements of the Model

In this section we show the elements of the model, identify the different sets and explain how they should be interpreted. This is done in the same way as in *Secure Computer Systems: Mathematical Foundations* [9] by Bell & LaPadula.

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; Metadata for files
U	$u_0 \dots u_n$	Users; registered users in the system
A	$A[u_0, c_0] \dots A[u_n, c_n]$	Boolean Access matrix; access rights

Set	Relation	Type
C	$\mathbf{C} \supseteq C \ni c$	$\mathbf{C} = P_{fin}(P_{fin}(C \cup M))$
M	$\mathbf{M} \supseteq M \ni m$	$\mathbf{M} = P_{fin}(M)$
F	$\mathbf{F} \supseteq F \ni f$	$\mathbf{F} = P_{fin}(F)$

The bold notation (**C**, **M**, **F**) refers to the sets containing all possible members of that type.

Element	Type
c	$C \cup M$
m	$\{id : GUID, file : f\}$
f	File

Container

A container is what would normally be called a folder in popular speech and in the terms of filesystems. A container can contain content objects and more containers, this recursive definition results in containers representing a virtual tree structure.

File

A file is an actual file, in the same sense as in a file system. A file could be for example a video or image.

Content

A content object contains a pointer to a file and metadata about that file. The pointer to the file within the object can change to a different file, but the content object still keeps the same metadata as before.

User

A user is a registered user in the system.

Access Matrix

The access matrix contains what containers that all the users in the system have access to. If $A[u, c]$ is true in the access matrix, user u has access to container c.

3.4 Access rights

Access is only handled on containers and if a user has access to a container it also has access to all of that containers children. The $A[u, c]$ notation is symbolising an access token. If $A[u, c]$ is true in the access matrix A, user u has full access to the container object c and its children. If it is false or if the entry does not exist in the access matrix, the user u does not have any access rights for c.

$$\begin{aligned} A[u,c] &\Rightarrow u \text{ can read } c \\ A[u,c] \wedge \neg \text{ read-only(c)} &\Rightarrow u \text{ can write to } c \\ A[u,c] \wedge \neg \text{ read-only(c)} &\Rightarrow u \text{ can delete } c \\ A[u,c] \wedge f \in c &\not\Rightarrow u \text{ can delete } f \end{aligned} \tag{1}$$

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 [11].
The invariant for this model is the file. Which means that it should not be possible to change a file through out the cycle of operation without creating a new file object with a new identifier. This is to ensure that files can not lose their integrity by concurrent operations from different nodes or get mixed up when referred to from a content object.

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 of unintentional race conditions etc.

3.7 Integrity Threats

According to Biba et al. [11] 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. Another external source 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 that does not behave according to its specification.

In this work external threats are handled as threats that can occur from what has been exposed by the API (See Section 4.2) and internal threats as incorrect implementation. As the server and its system are assumed safe, malicious subsystems are not considered.

3.8 Consequences of Operations

In this section the preconditions and effects of different operations are stated in an informal mathematical manner, similar to how inference rules are composed. These rules should be followed by the implementation in Section 4.

Action	**Semantics**
u, read(o)	User u reads o
u, create(m, c)	User u creates m in c
u, update(m, f)	User u updates f with m
u, delete(o)	User u deletes o
u, copy(o)	User u creates a copy of o
u, snapshot(o)	User u takes a snapshot of o

To describe a content object m containing fileId f, the following syntax is used: $\{id = \ldots, file = f\} \in M$, the same syntax is also applied to different types of objects throughout the explanation of the model. The "fresh" keyword applied to id means that it is a new globally unique identifier.

To explain a change of a container the notation $C[c]$ is used when removing or adding a nested container from the set. $C[\cdot]$ is a C that can contain the meta variable $\cdot$ and $C[c]$ is $C[\cdot]$ where $\cdot$ has been replaced by c.

File Read

$$\frac{A[u, c]]}{(C[c \cup \{m\}], F) \xrightarrow{u, read(f)} (C[c \cup \{m\}], F)} \tag{2}$$

A user u with access rights to a parent container c of content m can read a file f in content m.

File Creation

$$\frac{A[u,c] \qquad m \notin c \qquad \neg readOnly(c)}{(C[c],F) \xrightarrow{u,create(m,c)} (C[c \cup \{m\}], F \cup \{m.file\})} \tag{3}$$

A user u with access rights to a non-readonly container c can create a content m, containing a file f, in container c.

File Update

$$\frac{A[u,c] \qquad m' = \{id = m.id, file = f'\} \qquad \neg readOnly(c)}{(C[c \cup \{m\}], F) \xrightarrow{u,update(m,f')} (C[c \cup \{m'\}], F \cup \{f'\})} \tag{4}$$

If a user u with access rights to a parent container c (which is not read only) of content m, updates a file f in content m, the original file stays intact as an effect of Copy-on-Write.

File Deletion

Files can not be deleted by any user, however all the references to the file can be deleted, see Consequence 8.

Content Update

$$\frac{A[u,c] \qquad \neg readOnly(c)}{(C[c \cup \{m\}], F) \xrightarrow{u,update(m,m')} (C[c \cup \{m'\}], F)} \tag{5}$$

If a user u with access rights to a parent container c of content m updates that content, the content is directly changed, without any Copy-on-Write.

Container Update

$$\frac{A[u,c] \qquad \neg readOnly(c)}{(C[c],F) \xrightarrow{u,update(c,c')} (C[c'],F)} \tag{6}$$

If a user u with access rights to a container c updates that container, the container is directly changed, without any Copy-on-Write.

Copy

$$\frac{A[u,c] \qquad m' = \{id = \text{"fresh"}, file = m.file\} \qquad \neg readOnly(c)}{(C[c \cup \{m\}],F) \xrightarrow{u,copy(m)} (C[c \cup \{m,m'\}],F)} \tag{7}$$

If a user u with access rights to the parent (source and destination) container c of content m copies that content, the new content m' is not equal to the old (because of its fresh ID), however they both contain the same file.

Content Deletion

$$\frac{A[u,c] \qquad \neg readOnly(c)}{(C[c \cup \{m\}],F) \xrightarrow{u,delete(m)} (C[c \cup \{\}],F)} \tag{8}$$

If a user u with access rights to a parent container c of content m deletes that content, the file f referred to in the content is not deleted.

Snapshot

$$\begin{aligned} &snapshot(c) = \{snapshot(c') \mid c' \in c\} \cup \{snapshot(m) \mid m \in c\} \\ &snapshot(id = i, file = f) = \{id = \text{"fresh"}, file = f\} \end{aligned} \tag{9}$$

The snapshot operation makes a deep-copy by traversing the full tree or sub-tree of containers and content that is to be taken a snapshot of. As the snapshot

operation is performed upon a tree of objects it results in two new sets (C' and M') which are then added to C and M respectively, the referred files in the content remain the same as before the snapshot.

$$\frac{A[u,c] \qquad \neg readOnly(c)}{(C[c],F) \xrightarrow{u,copy(c)} (C[c,snapshot(c)],F)} \tag{10}$$

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

3.9 Interaction of Multiple Accesses

As a result of consequence 5 and 6 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 in favour of scalability and speed is intentional. However, as can also be seen in consequence 4, if a file is removed from a content that file is not removed from the set of files, even ough its modification process is 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, these containers and content are defined as garbage. This could happen as a result of consequence 8 in combination with any of the other consequences listed in Section 3.8. The basic idea is that an entity is deleted while another is created or modified to have that entity as its parent, thus creating a separate tree that can not be reached from the root container. These entities should be removed, for example by a garbage collector, or made sure to be deleted in the instance when they are no longer referenced.

3.10 JPF

JPF was used to test the idea and state transitions of the model. It is a very simplified version of the real system that still contains all the important Copy-on-Write 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

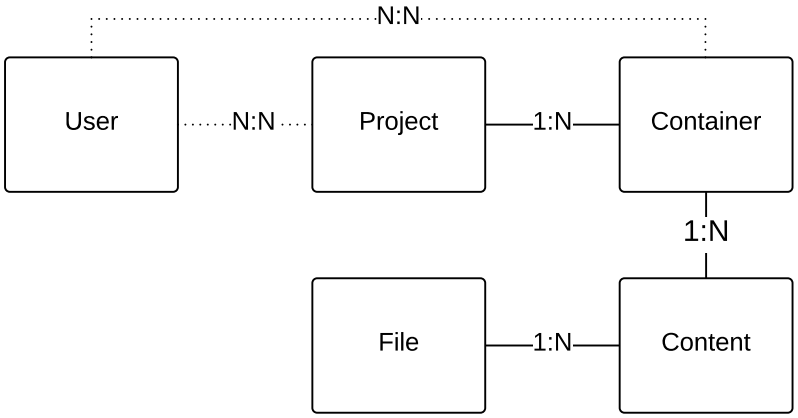


Figure 1: High-Level Entity Relationships

All the entities in Figure 1 is described in Section 3.3, except for projects, which is explained below.

A **Project** is what is created to contain all content and containers related to a real project. Files and metadata can only be changed within a project and the system can contain several projects that have their own virtual content and containers completely disjoint from the other projects.

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 hash function 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. The ID's are randomly generated by the ObjectId class to minimise or completely avoid collisions. Furthermore no locking or transactions were used and the threads were running fully concurrently and without any sleep statements.

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

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

Listing 1: Results of JPF run

```
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

This section summarises the findings from the work on the model.

3.11.1 Garbage Collection

For the system to reach a clean state and eventual consistency [12], garbage collection or reference counters will be needed. The conclusion of Section 3.9 is that there will be objects in the system that are unreferenced and can not be reached and they should therefore be cleaned up to not cause negative performance impact 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 metadata 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 [13].

3.11.3 JPF Results

Although the JPF model checking results did not return any errors or showed any possible misbehaviours of the model, it should not be considered proven. The results however strongly indicate that there are no major flaws in it. But as the model will be more complicated once fully implemented as an application there might be flaws in the implementation details.

4 Implementation

This chapter deals with the second part of this work, namely the implementation of the Perius system. The implementation 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

Section 4.1 explains the problem at hand, in relation to the system currently used.

4.1.1 The current system

Today a system called Battlebinary [14] is used by Uprise 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 available to the public, but since its identifier is composed out of its original filename concatenated with a part of the MD5 hash of the content of the file. This makes it a very computationally heavy process to access the file on the CDN without access to the original file or a reference to the URI, as the identifier would have to be brute forced.

In the current system you can only upload a file once, since there will be an identifier collision. This is due to the old and the new file having 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, since one file can not be shown or uploaded in two different places in Battlebinary's virtual filesystem.

4.1.2 Problem description

The current system does not offer proper security measurements as its mechanism for keeping private content out of the hands of the public is based on security by obscurity, which is further explained in Section 4.1.1.

It is also lacking several features that are needed and does not scale very well, hence a new system is to be developed. Features that needs to be implemented are for example snapshots, cloning and proper support of concurrent modifications of content.

This work is about examining a way of implementing Copy-on-Write in a high-level system, which should solve the scalability problem and make it possible to implement wanted features.

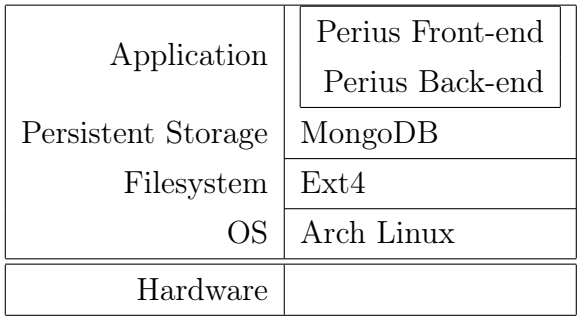


Figure 2: Software Stack

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 part of the stack, in this work only the second highest layer is considered, the Perius back-end.

4.2 Architecture and Technologies used

This section introduces the different technologies used in the implementation. The section also contains motivations and explanations why certain technologies were chosen and briefly outlines the software architecture.

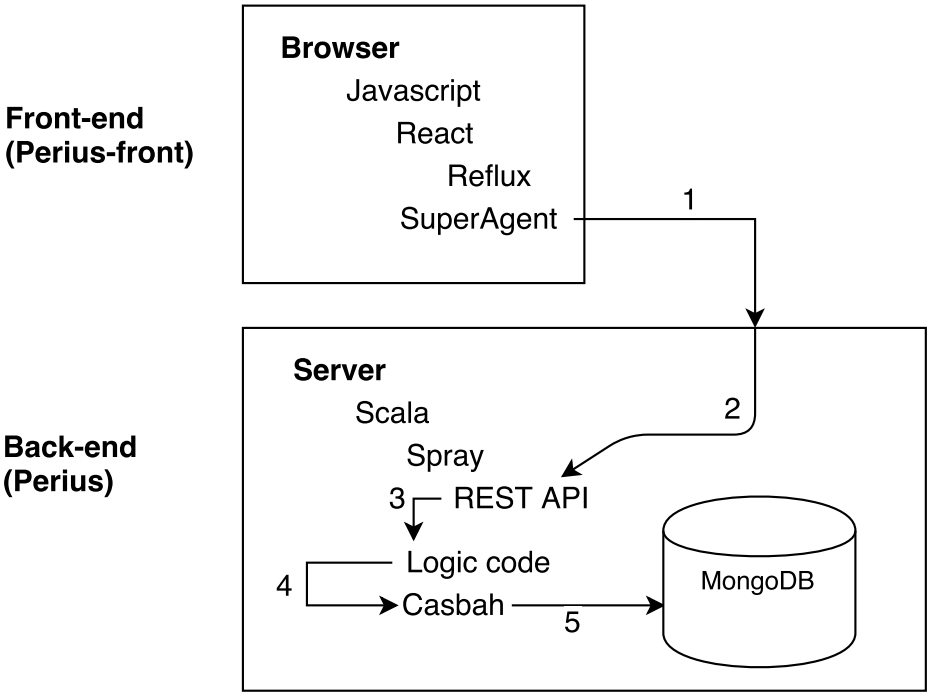


Figure 3: Technology Stack

Figure 3 shows how the different technologies used are related. In the Figure, the technologies that are indented are built on top of its parent. The arrows that can be seen is the flow of data after a request from the front-end has been made. The arrows are unidirectional, but the response follows the same path back to the requester.

Javascript is a programming language mainly focussed on running code in web browsers [15]. In this implementation the ECMAScript 6 (ES6) language specification is used, which adds a lot of syntactic sugar compared to what usually is called normal JavaScript. An example of such syntactic sugar is lambda expressions. React and Reflux are both built with JavaScript in mind and is used on top of JavaScript in this implementation.

SuperAgent is a small HTTP request library which can be used from client-side Javascript or as a node.js module [16]. It is used to form the requests to the REST API from the front-end, arrow 1 and 2 in Figure 3.

React is a JavaScript library for building user interfaces. React uses both its own virtual DOM and the browser's normal DOM, 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 [17]. React can be seen as the system for handling views in front-ends implementing a MVC-like (Model-View-Controller) architecture. In this work React is used as a part of building the front-end.

Reflux is an idea and a simple library of how to structure your application [18]. It features a unidirectional dataflow (see Figure 4) which makes it more suitable, than for example Flux [19], when using a functional reactive programming style. Reflux is used together with React to define a structure of the flow of data and state changes.

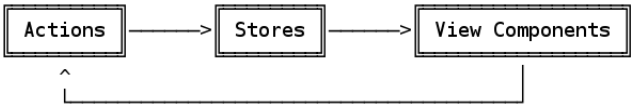


Figure 4: Reflux unidirectional dataflow

Scala is a multi-paradigm programming language. It most commonly runs on the JVM and unlike Java it supports most functional programming features at the same time as it supports object-oriented programming [20]. Scala is used to program the back-end and the Spray toolkit is used to simplify the API.

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 [21].

In this work a RESTful API was implemented and used for back-end $\Leftrightarrow$ front-end communication, arrow 1 and 2 in Figure 3.

REST was chosen as all operations that needs to be performed can be defined as basic CRUD operations 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].

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.

Spray [24] is an open-source toolkit written in Scala and for Scala. Spray is mainly used to make it easier to build REST/HTTP APIs.

Casbah [24] is a Scala toolkit and a java driver wrapper used to interface with MongoDB instances in a more Scala oriented way.

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 that is not fixed to a predefined table structure [25]. MongoDB lacks support for join operations to improve its possibility to scale, which can be a big disadvantage for applications having the need for such logic. MongoDB is used as a persistent storage for the back-end to store all the projects and nodes in the tree structures and also the files uploaded to the system. The Scala logic code uses a layer called Casbah [24] to interface with MongoDB, arrow 4 and 5 in Figure 3.

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 then it is very easy to distribute MongoDB and, if needed, the system can easily be migrated to Reactive Mongo. Reactive Mongo is an asynchronous and non-blocking driver for MongoDB and can therefore make the system scale even further [26].

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 storing files directly in the database rather than in the filesystem is that backups of can be done through a database backup which will contain both the meta-data stored in MongoDB and the files which results in that no separate files needs to be backed up.

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 a 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 directly.

4.3 Methods for determining implementation details

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

As this work was not mainly about the system architecture, but rather about evaluating the effects of using Copy-on-Write, the decisions taken for the different

technologies used were simply made from what the author was comfortable with and expected would have little impact on the evaluation. With that said there are no comparisons of for example different databases, protocols or API solutions in this paper.

The long-term scaling was done by load testing the back-end with Wrk and Apache Bench. With the help of these tools, the computationally expensive operations could be found and it could be determined what the upper bound for how many active users the system could handle on a node with similar hardware specifications.

A low timeout threshold was used to make sure that the system could handle operations faster than they queued up. This was to be certain that the system could sustain the number of active users during a longer time frame without having a queue that could grow beyond a manageable size.

4.4 Copy-on-Write

This section is about how Copy-on-Write was used in Perius and how its implementation of it differs from other systems that incorporate Copy-on-Write.

4.4.1 Snapshot Functionality

The Copy-on-Write concept is not only well-suited for consistency requirements, its basis also gives the possibility of creating snapshots.

To efficiently create snapshots of a system, Copy-on-Write can be used to make it possible to create snapshots in O(1) [5]. 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 5.

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 Examples 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 node's counter can be modified without modifying the node itself and therefore avoids the Copy-on-Write that would have to occur.

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

Figure 5: Cloning mechanism of Btrfs [5]

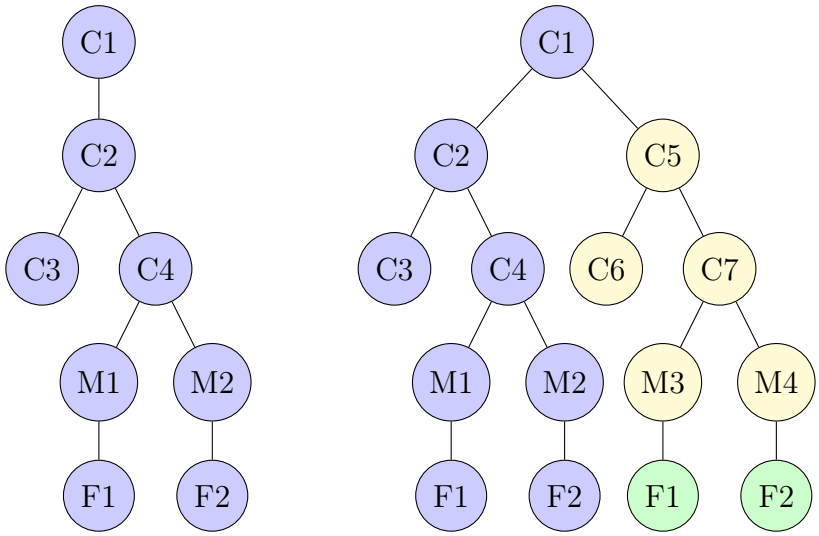
4.4.3.2 Mach kernel

In the mid 80's when the development of the Mach kernel started, there was problems with that physically copying memory was too slow. To minimise the copying of memory, Copy-on-Write was implemented to enable the use of virtual copy operations and to create the possibility of tasks sharing read-write memory [27].

4.4.4 Perius

As the persistent storage used in this implementation (Section 4.2) does not implement transactions or locks, several different problems can occur when multiple clients are working on the same data set at the same time. Such problems could be race conditions or determining the happened-before relation [28]. Race conditions are problematic due to that when several users or nodes modify data they can start in one state and expect the end state of the data to be the same as if they would have been alone modifying the data, but instead if another user modifies the data concurrently the changes that the first user made can be overwritten. In this work the race condition problem is solved by implementing Copy-on-Write on data with high consistency requirements. The happened-before relation is a

problem of ordering events, if it is possible to know in which order concurrent events happened it is easier to know what the outcome of the events should be. The happened-before relation is not fully solved in Perius, but suggestions of how to solve it is given in Section 3.11.2.



(a) Original Tree (b) Tree with C2 Snapshot

Figure 6: Snapshot Explanation
C - Container, M - Content, F - File

In Perius, snapshots and clones are not taken in the fashion which Btrfs uses, which can be seen in Figure 5. 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 metadata of the tree, but still refers to the same binary files until the metadata is changed to point to another file.

Figure 6 shows how a snapshot is performed on C2 and stored in C1, and how the resulting tree would look like. In Figure 6b all of the yellow nodes are copies of metadata that were in the left tree, for example C5 and C2 contain the same data (except for their newly generated IDs). The green nodes refer to files that have not been copied but are still referenced in the copied metadata, this is due to the fact that a snapshot is not a write to the files and they are therefore not copied but remain intact.

4.5 Resulting system

The *Resulting System* section explains what the implementation looks like and how its different operations work, in the terms of efficiency.

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 independently replaceable. The back-end has a REST API running, which is how the front-end communicates with the back-end.

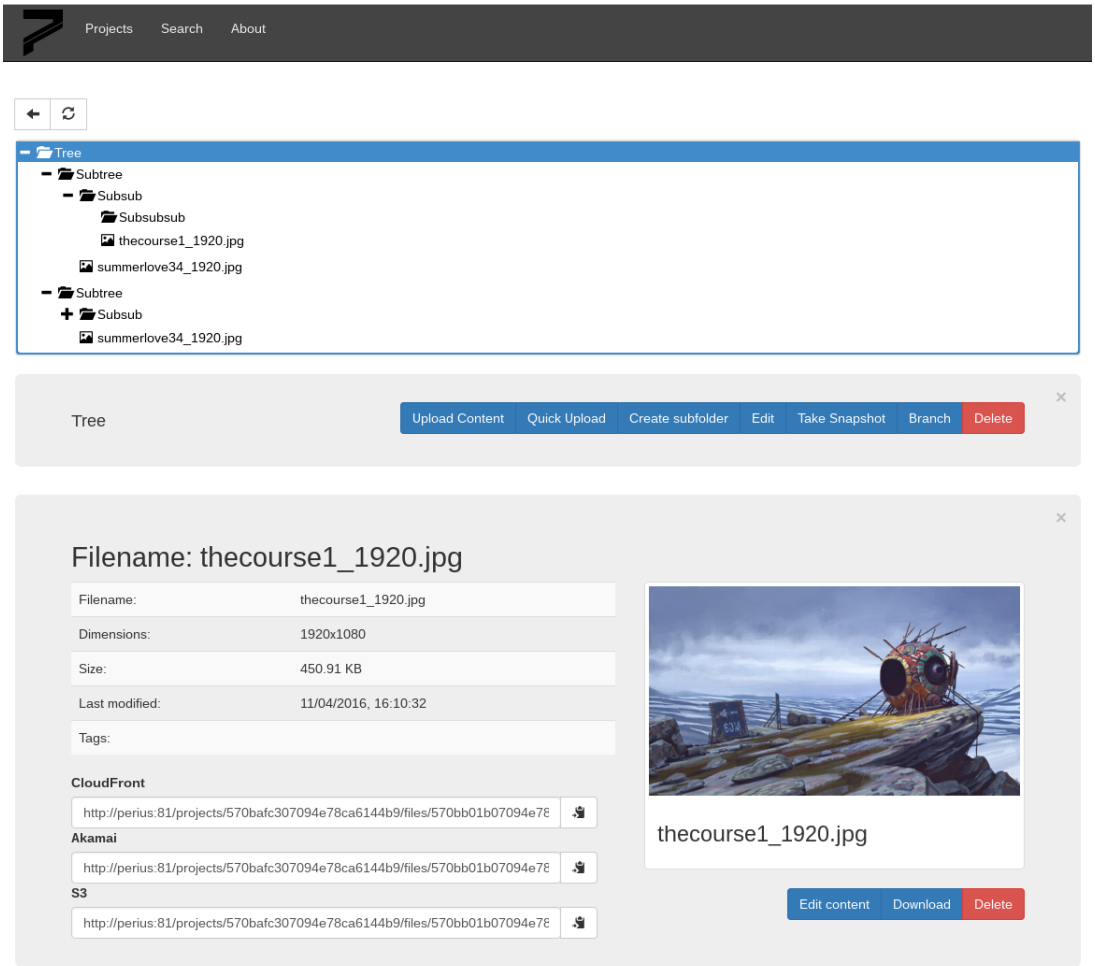


Figure 7: Front-end [5]

The service features a virtual file structure over the assets and metadata that have been stored, as can be seen in Figure 7. It also features snapshots, security

management of whole containers as well as individual files, audit and auth logging, multiple separate projects and a design which allows for a replaceable 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 Entities of the Implementation

This section explains the different types of entities, and their interaction, that are handled by Perius. They are explained similarly to the model in Section 3.3, but with more implementation-specific details.

Content
: Content is metadata 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 and metadata can only be changed within a project and the system can contain several projects that have their virtual content and containers completely disjoint. Each project is handled as a separate database in MongoDB.

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 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.

4.5.3 Runtime 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.

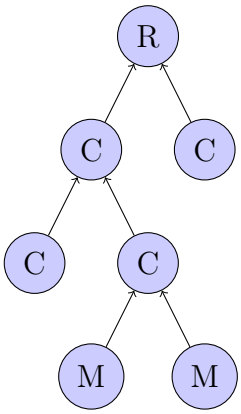


Figure 8: Project Tree
R - RootContainer, C - Container, M - Content

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 8.

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 9: Simplified Representation of the Actual Storage

A very simplified version of how the data is stored can be seen in Figure 9, however in reality it is not quite so table-like. The implementation is using GUIDs, instead of incremental numbers as shown in the figure, to work correctly in a distributed environments.

As seen above, the data is not stored as a tree (see Example 9), but rather as separate documents loosely referring to each other. How the different operations perform their actions and what implications these actions have on the run-time complexities are not obvious and that is what is discussed in this section.

Variable	Explanation
n	All content+containers
c	All containers
m	All content
f	All files
s	Size of a file
k	Chunk size

Figure 10: Variables used in run-time complexities

GET single entity

When the back-end is handed a GET request it looks at what type of entity that is requested 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 since 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 each parent. Since all of the containers have to be processed when building the tree, the runtime complexity will thus be $\Theta(c)/$

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 [29]. Even though the insertion itself has constant run-time the index needs to be updated or sometimes even rebuilt, which can slow down

the time before the document is accessible in constant time. When inserting files they need to be chunked into pieces and inserted into GridFS, this results in s/k insertions. The insertion of files can then be regarded as $\Theta(s)$ if the size of the file is regarded, but if k is larger than the size of the files inserted, 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 metadata in the tree that the snapshot is taken of is duplicated, meanwhile the files remain the same, this results in t insertions where t is the number of containers and content entities in the subtree. A snapshots insertion runtime is therefore $\Theta(n)$.

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, $\Theta(n)$ for containers and $\Theta(1)$ for content.

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 a new one, 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, if for example the parent of the document is changed. As both deletion and insertion are constant time operations, update is also $O(1)$.

4.5.4 Copy-on-Write

This implementation is far from as efficient as the other Copy-on-Write systems described in Section 4.4 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 arguable (See Section 4.5.3) takes $\theta(log(n))$, where n is the number of nodes in the subtree.

4.6 Load Testing

Wrk and Apache Bench was used to load test the back-end. Wrk is a multi-threaded benchmarking tool for HTTP which can create large loads and Apache Bench has similar capabilities, but is better suited for file uploads than Wrk. The testing was done locally on a virtual server having 6 CPU cores and 48GB of RAM. The results that will be analysed are the throughput of each request performed without breaking the set timeout 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 the bottleneck of the back-end is, three different types of *GET requests* are performed. Most responses are JSON structures (except for the ones in Listing 2 and Listing 4) which can be seen in the Response data part of the listings.

The typical requests performed 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

Response data:
200 OK
```

The first request (Listing 2) 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). 100 000 requests per second is well over what

we need to serve and with a 1.22ms latency we know that without practically any payload the system is responding fast.

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

Response data:
[{
  "id": "565cdae1898b798008105fd47",
  "name": Test Project,
  "readOnly": false,
  "lastModified": 1456320024643
}]
```

The second request (Listing 3) performs the simplest API call which involves getting all currently stored projects from the MongoDB instance. This request does not yield any extra computation in the back-end, apart from converting the MongoDB format to the API's JSON result format. From these requests it can clearly be deduced that the round trip time to fetch a document in the database is adding some latency, from the average of 1.22ms to 14.52ms with the database request.

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

Response data:
Static test data for WRK not involving MongoDB
or filesystem access.
Lorem ipsum dolor sit amet, consectetur adipiscing elit.
```

As the second request (Listing 3) involves the server answering with some payload that is generated from the persistent storage, the third request (Listing 4) is simply the same amount of static payload as the second request but without involving any connections to MongoDB. This was done to be able to estimate what impact the round trip time to MongoDB and the conversions to the response format had on the throughput. This test still gave an average latency that was larger than for the smaller payload in Listing 2, but still half of the latency compared to the request involving a MongoDB call.

Listing 5: Result of project tree requests

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

Running 10s test @ http://perius:8000/projects/56...
  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

Response data:
{
  "parent": "",
  "name": "Test Project",
  "readOnly": false,
  "lastModified": 1456320024643,
  "id": "56cdae1898b798008105fd47",
  "containers": [{
    "parent": "56cdae1898b798008105fd47",
    "name": "Upload stuff",
    "readOnly": false,
    "lastModified": 0,
    "id": "56cdae2298b798008105fd48",
    "containers": [],
    "content": [{
      "fileId": "56cdae2d98b798008105fd49",
      "parent": "56cdae2298b798008105fd48",
      "size": 35245,
      "readOnly": false,
      "lastModified": 1456320045336,
      "tags": [],
      "cdnLinks": [],
      "id": "56cdae2d98b798008105fd4b",
      "filename": "newtree.png",
      "dimensions": [1088, 369],
      "originalId": "56cdae2d98b798008105fd4b"
    }, ...]
  }],
  "content": [{
    ...
  }]
}
```

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. Since the project tree will not 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 algorithm for building and fetching the whole tree from the database. This is will not be discussed further, but left as future work. Even though this massive bottleneck is present, it will not be a problem in a small scale production environment where 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

Response data:
Same as in Listing 5
```

With caching turned on, on a single back-end, the application can theoretically support several million active users that are not posting any content, this is simulated in Listing 6.

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

Response data:
Same as in Listing 5
```

As a project tree is built by querying documents parent ID an optimisation that could be done is 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 timeout, 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 active clients that one node can handle will be >60000 clients at the same time, without taking the web server 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

Response data:
{
  "id": "56cdae2298b798008105fd48",
  "parent": "56cdae1898b798008105fd47",
  "name": "Upload",
  "readOnly": false,
  "lastModified": 1456320045336,
}
```

As can be seen in Listing 8 the server can handle around 4000 container POST requests per second. A problem that was discovered during this test 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 tree from the web server it timed out the request since it took too long for the back-end to build the tree. In this work the web server will not be configured to accept longer data processing time, due to the low probability of trees as large as this being 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 filesystem 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 is 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

This section declares for the software security features that were used in Perius, in the terms of how the system and its assets are protected internally and externally.

4.7.1 Authentication

The authentication of users in the system is currently being handled by LDAP [30], since Perius mainly will be deployed in internal networks with an existing LDAP infrastructure. 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 have perform which actions. In the event that somebody, consciously or unconsciously, are leaking private CDN content the audit trail can then 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 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 [31]. 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 the later paragraph "IP range restriction") and inclusion of the base64 version of the policy in the URL [32]. 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 [33].

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 a restricted groups internet access is based on a set of public static IP addresses.

4.8 Findings

This section presents the relevant findings derived from the implementation of Perius and the testing of it.

4.8.1 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 content 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 Perius it is possible to have three different modes for a CDN asset; public to all, only accessible from certain IP ranges or only accessible by clients using a signed URL or cookie. 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.

4.8.2 Scalability

Perius uses MongoDB as its persistent storage and all computationally expensive operations involve doing calls to it. Scala currently has two popular drivers for interaction with MongoDB, namely Casbah and ReactiveMongo.

When using the ReactiveMongo driver [26], which is asynchronous and non-blocking, the application has no limits of how much load or how many users it can handle as the hardware and nodes can be scaled up linearly when needed. With Cashbah [24], which is used in the current implementation, it is harder to scale to the enormous amounts of load that 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.

4.8.3 Load Testing

Section 4.6, specifically 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 was analysed. The most complicated and still common request that the 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.

From the load testing a bottleneck of the application was found, namely building the project trees. Three solutions for this was presented and two were tested. The first was to cache the results of computationally heavy requests. The second was to simply make the operations more computationally efficient, which was not tried. The third presented solution was to add indices to keys that were often queried in MongoDB. The third solution yielded the best results and made Perius being able to serve 2100 requests/s, which involved building the project tree. A combination of solution one and three could possibly make Perius able to serve even more requests per second.

5 Discussion

This chapter contains informal discussions about topics that were not covered by the rest of the report but that are still significant enough to mention.

5.1 Persistent Storage

More research could have been done in the choosing of persistent storage, as mentioned in many applications that choose NoSQL databases as their persistent storage actually do not need it [12]. 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 to the accepted JSON format from the REST service. As Perius most common operations involve modifying project trees, a graph database should have been researched too.

5.2 Copy-on-Write's effect on Scalability

Since no comparison with solutions that were not Copy-on-Write based were made, no scientific conclusions can be drawn from what 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 since 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, in some sense, be fully aware of the state of the other nodes. Sharding [34] could be used to minimise the communication of state between the nodes, but that results in less fault tolerance and possibly higher loads on some nodes than others due to that the data most likely will not be accessed uniformly.

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 Scalability

Since GridFS was used to store the files an unexplored option for making it easier to scale is to use GridFS feature of retrieving specific ranges of a file. For scalability this could be used to retrieve different parts of a file from different servers, but normal load balancing would probably work better in a system like Perius where no extremely large files are expected to be stored.

5.4 Branching

Branching was also implemented in Perius, but to not make the report too broad it was not included. The branching worked by the user selecting a container that the branch should be based on and a container in which the branch should be placed. The user could then choose which containers and content that it wanted to update with the origin and which objects that should be deep copied to the new branch.

5.5 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.

5.6 Access Control

Full object-level access control was not implemented according to the model described in Section 3. The implementation only checks whether a user should have access to the system as a whole, and does not set, nor check, specific access rights to contents or containers.

6 Summary

This section covers the conclusions that I draw from this work and how I plan to extend the implementation in the future.

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 where last write wins [13] on meta data, but in all real world observations no users have noticed it. The theoretical inconvenience for the user was a trade-off made so that the application could scale almost endlessly, especially if the issue with building the project trees is solved (as mentioned in Section 4.6).

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 persistent storage logic would have to be largely rewritten for every existing application that is 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. This approach could efficiently 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 Copy-on-Write and conflict resolution. This could be an advantage when deploying Perius in environments with high requirements for the number of users that the system should be able to handle. It would also be well suited if the level of consistency needs to be fully tweakable to make it possible to trade off consistency for scalability.

6.2 Future work

6.2.1 Access Control

As access control was not fully implemented (See Section 5.6) as planned in the model, the implementation will need to be extended in the future.

6.2.2 Front-end Refactorization

Currently the implementation fetches the full project tree every time a change is made. This results in increased latency, bandwidth usage and server load. We propose a refactorization of the front-end where it updates the state according to the REST response of each modification to the project tree instead of fetching the whole tree again.

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] 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.

[6] O. Rodeh and A. Teperman, "ZFS-a scalable distributed file system using object disks," in *Mass Storage Systems and Technologies, 2003.(MSST 2003). Proceedings. 20th IEEE/11th NASA Goddard Conference on*, pp. 207–218, IEEE, 2003.

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

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

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

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

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

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

[13] 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.

[14] "GitHub - battlelog/battlebinary," Uprise, September 2015. `https://github.dice.se/battlelog/battlebinary`, accessed 2015-10-15.

[15] D. Flanagan, *JavaScript: the definitive guide.* " O'Reilly Media, Inc.", 2006.

[16] A. Mardan, "REST API," in *Pro Express. js*, pp. 249–261, Springer, 2014.

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

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

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

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

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

[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. Alexandre, *Scala for Java Developers.* Packt, 2014.

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

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

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

[28] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[29] "Write operation performance," MongoDB, January 2016. `https://docs.mongodb.org/manual/core/write-performance/`, accessed 2016-02-19.

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

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

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

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

[34] R. Cattell, "Scalable SQL and NoSQL data stores," *ACM SIGMOD Record*, vol. 39, no. 4, pp. 12–27, 2011.