UNIVERSITY OF TROMSØ UIT

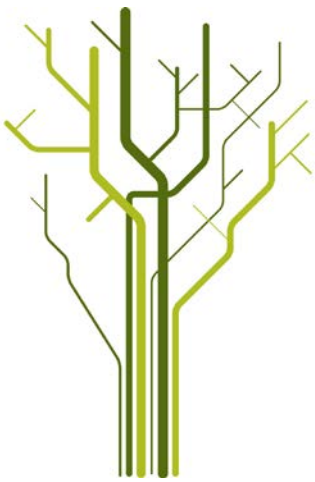
FACULTY OF SCIENCE AND TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE

A Video Storage Management System for Soccer Analytics

Roger Bruun Asp Hansen

INF-3981 Master's Thesis in Computer Science

December 2012



Abstract

Video is dominating consumer internet traffic. Restless internet users expect smooth playback and low latency when watching video content and vendors risk losing customers if this cannot be provided. Distributed storage systems specialized for delivering video content and for handling the high traffic this lead to, have been developed over many years.

This thesis look into building and deploying a distributed video storage to deliver video content to an enterprise sport analytics web interface using open source solutions. We look into properties such as horizontal scalability, fault tolerance, security and privacy to outline an architecture that can scale horizontally and deliver video content in an efficient way to a HTML5 web interface. Our conjecture is that developing such a distributed storage solution from scratch is not feasible, and that other, accessible, storage solutions should be evaluated. We ended up with deploying OpenStack Object Storage on our cluster, integrating it with the web front end for uploading, managing and accessing video content.

Our experiments test and evaluate the performance and boundaries of our system. We also discuss elasticity problems related to sudden spikes in interest, and solutions in the context of privacy and economic issues.

The final deployed video storage system has replaced an old centralized approach that previously has been used. It is currently in production delivering video content to a web interface that is used for soccer analytics by our sport partner Tromsø IL. A case study has been made to observe how they have been using our systems for the last year.

Acknowledgement

I would like to thank my supervisor Professor Dag Johansen for excellent feedback, advices and ideas, during all stages of this thesis. No one have inspired and motivated me more than you. I would also thank my co-advisor Krister Mikalsen for his involvement during the past year. I am looking forward co-operating more with you in the future.

Further I would like to thank Agnar Christensen and the staff and athletes from Tromsø IL for letting us record videos and using the systems we build for analyzing matches and practices. Your time, feedback and evaluation have been of great value.

I would like to thank all the other members from the iAD group. Thank you Magnus Stenhaug for your involvement building the cellular application and being present at close to every home match this season recording. Thanks to Håvard Johansen and Robert Pettersen for helping me out with re-configuring the old rocks-cluster so I could use it in my thesis.

Special thanks go out to my friend Kim Sørensen for his valued friendship. Thank you for, from time to time, sticking up with my nonsense. Our discussions and co-operation during these years have been invaluable. I look forward working with you in the future.

Finally, I would like to thank my girlfriend, my friends and family for their feedback and support.

Table of Contents

List of Figures

List of Tables

1. Introduction

Video dominates consumer internet traffic. It is forecasted to be 55% of all customer internet traffic in 2016 [1], which will result in an astonishing 1.2 million minutes of video content crossing the network *each second*. 12% of this traffic is estimated to be short form videos that are defined to generally be shorter than 7 minutes. The popularity of short form videos are estimated to grow faster at a compound annual growth rate of 34% than for long term videos, which only will grow at a rate of 22%. Mobile videos are forecasted to grow by the highest rate of 90% and end up for almost 17% of all consumer internet traffic in 2016, only surpassed by long form videos that alone will stand for 38%.

Social networking and video sharing sites is the dominant portal for uploading videos [2] and 25% of users browsing videos online share or repost them on social sites where many people can watch [3]. By sharing videos on social networks, such as Facebook, sudden increased load (peaks) can quickly occur on the original site where the video is hosted. Facebook, YouTube and Vimeo are leading vendors in multimedia hosting, and have for many years developed and built solid foundation for their back-end solutions to support extreme load while still providing a good user experience. Content distribution networks are also built to distribute and cache data to geographically dispersed locations close to users to provide low latency and to avoid multimedia-requests ending up in the same centralized location.

Sports, and especially soccer, appeals to a large audience in the world today. Peak interest in videos containing spectacular goals or acrobatic movement can be anticipated to reach out to fans all over the world in a very short amount of time due to sharing through social media, such as Facebook and Twitter. Video streaming sites must anticipate that massive load might occur, and must ensure that these videos stay available even if nodes fail. Failure is the norm, and being available is crucial for keeping users around.

There are many techniques that can be applied to store data in an available and scalable fashion. Data can be deployed in public cloud storage solutions or developers can deploy cheap private cloud storage solutions on their own clusters using open source storage technologies. Using public cloud vendors for archiving large amounts of data that are accessed little to maintain availability quickly gets expensive [4]. Buying storage devices to store data in a private cluster is cheaper than archiving data over a long period of time in a public cloud storage solution. In a private cloud solution where each hardware component is specialized for this purpose, archiving videos in a fault tolerant and available way can be relative cheap. Public cloud solutions do provide a more elastic scale and *unlimited* amount of storage.

1.1 Problem Definition

"This thesis shall design, deploy and evaluate a video storage management system for efficiently storing, managing, securing and disseminate video content on a heterogeneous intra-cloud platform. Focus will be to provide horizontal scalability within a private cluster and fault tolerance in a system that efficiently serves video files to a sport analytics web interface in a secure manner. Video format and encoding must be evaluated to ensure that playback quality is satisfying."

1.2 Interpretation, Scope and Limitations

This thesis will explore efficient and fault tolerant ways to store and disseminate captured video-snippets in an intra-cloud storage network with heterogeneous hardware. We conjecture that developing such a distributed storage solution from scratch is not feasible, and that we should research and evaluate other, accessible, storage solutions and how they fit into our scenario. Deploying an intra-cloud solution provide us the elasticity to transparently grow the storage capacity by adding more servers to the storage cluster while still maintaining the system available.

The video storage system will be designed from a set of functional and non-functional properties desired by users of the system. Different existing storage solution that can relate to these properties will be evaluated, and we will deploy one of these onto our intra-cloud platform. Re-organization of the underlying network infrastructure needs to be considered. Our deployed video storage will be benchmarked and evaluated according to the desired properties.

Dependent on the delivery platform of our users, efficient and compatible ways of encoding the video files need to be evaluated. One of our main goals is to deploy a production ready distributed video storage solution that supports authentication and/or authorization of applications and users. This solution should be fault tolerant in the sense that it support failing nodes, still keeping the system available. It should also be horizontally scalable in the sense that nodes can be added to the solution to improve performance. We mainly look at an enterprise solution, but should also discuss how we can handle sudden peaks of interest in public scenarios. Videos should be stored in a redundant manner; such that no data is lost if a node fails.

We are limited to a cluster of nodes interconnected through a single private Gigabit switch, and nodes that need to be connected to public network can be linked to different single public Gigabit switch.

1.3 Methodology

The final report [5] of the ACM Task Force on the Core of Computer Science divides the discipline of computing into three major paradigms. These are theory, abstraction and design. A short summary is given below.

Theory is the first paradigm and is rooted in mathematics, and is approached by studying object, define problems and propose theorems. It seeks to prove these theorems in order to determine new relationships among the objects in order to progress in computing.

Abstraction is rooted in experimental scientific method, and seeks to investigate a phenomenon in computing by forming a hypothesis, constructing models or simulations to challenge the hypothesis, and finally analyzing the results.

The last paradigm, design, is rooted in engineering. It seeks to construct a system to solve a given problem by defining the requirements and specifications of the system. The system should then be designed and implemented according to the requirements and specifications. When a system is built, it should be tested.

For this thesis, the design process seems to be the most appropriate one to use. We have stated a problem, its requirements and specifications, and need to build a prototype to solve it. The testing of the system will be done by end-users evaluating the quality, as well as measurements to quantify the performance and its limits.

1.4 Context

This thesis is a part of the Information Access Disruption (iAD) project. The iAD project is partially founded by the Research Council of Norway as a Center for Research-based Innovation (SFI) and is directed by Microsoft in collaboration with Accenture, Cornell University, Dublin City University, BI Norwegian School of Management, as well as some of the major Norwegian universities: Tromsø (UiT), Trondheim (NTNU) and Oslo (UiO).

The iAD group in Tromsø focuses on researching fundamental structures and concepts for large-scale information access applications, and is currently building and evaluating information access runtime systems for private and public computing environment.

Current undergoing research is within

- Evidence-based technologies in the sport domain with Muithu [6] and Bagadus [7].
- The cloud software stack with Vortex [8] (virtualization), Codecaps [9] (security), Suorgi [10] (overlay) and Balava [11] (federation).
- Big data analytics with Cogset [12] and Oivos [13]

This thesis is a part of the Muithu project. With Muithu we explore this with videos extracted by a high level notational system used for sport analytics with the local Norwegian premier league soccer team Tromsø IL (TIL) as a well-established partner.

TIL is one of the top soccer clubs in Norway, and have for the last year been using and testing prototypes of the Muithu system during practices and matches with up to six recording cameras, contributing with statistics, notational metadata and video recordings to this thesis. Part of this thesis is to deploy a fault tolerant and available storage solution for video snippets recorded by Muithu.

Another iAD project that in many ways is a predecessor and inspiration to the Muithu system is Davvi [14]. Davvi had a more social and entertainment focus, as annotations and metadata in soccer context was used to provide search and recommendation service to provide a personalized, topic-based user experience. Muithu is currently considered as a tool to annotate and post analyze athlete performances in form of video snippets, but in the future Muithu might end up with an additional social and entertainment focus.

By proposing and deploying a private cloud storage solution, we also look at this thesis in the context of current undergoing cloud projects, such as Balava [11], which focuses on building extensible run-time systems that support resource elasticity in private cloud environments in combination with public clouds, to timely handle peak resource demands. Balava is a cloud federation system that is inspired by the security and privacy concerns that enterprises face when considering deploying computation or data into public cloud offerings.

1.5 Outline

The rest of this thesis is structured as follow. Chapter 2 contains relevant background information and related work on topics within databases and storage. We outline requirements and both functional and non-functional properties that the video storage desires, as well as outlining architecture that adhere to these properties in chapter 3. Chapter 4 gives insight in how the final solution in this thesis have been designed, deployed and integrated with the already existing Muithu web application. Chapter 5 gives us a case study, experiments and evaluation before we conclude this thesis in chapter 6.

2. Background

The development and scaling of database management systems (DBMSM) have for many years relied on vertical scaling by buying new hardware for a single server to support more load on the information system. Many techniques and new systems have in the latter years been designed to provide good horizontal scalability for simple read/write database operations in distributed environments. In heavy read-only environments, data replication can be used to address large workloads. On the contrary, in write-intensive environments, data replication will lead to high overhead as consistency demand that each replica is updated to the newest version at each read-write query.

Even though data management is more complex in a distributed storage approach, there are many advantages. Özsu and Valduriez [15] try to distill these advantages into four fundamentals:

1. *Transparent management of distributed and replicated data*: A DBMS should ideally be distribution transparent, meaning that it should hide details about data locality, replication of data and data fragmentation.
2. *Reliable access to data through distributed transactions*: Distributed DBMS systems should improve reliability as components are replicated across physical nodes, thereby eliminating single point of failures. In a centralized environment, failure of a single node will make the whole system unavailable.
3. *Improved performance*: A distributed DBMS fragments the conceptual database, enabling data to be stored within close proximity to its points of use, enabling reduction of remote access delays involved in wide area networks. Fragmentation of data also decreases CPU and I/O intensity on nodes, compared to centralized solutions.
4. *Easier system expansion*: In a distributed environment, it is much easier to increase the database size, as expansion usually can be handled by adding processing and storage power to the network. An important aspect of scaling the system out (adding more nodes), is that it often is much less expensive economic-wise, than scaling up a single node.

2.1 Transaction Processing

The traditional transaction processing model is characterized by *atomicity (A), consistency (C), isolation (I)* and *durability (D)*, which also is known as *ACID* properties.

- **Atomicity** guarantees that a transaction is either fully executed or not executed at all. If some part of the transaction fails, the whole transaction should fail.
- **Consistency** ensures that the database will not end up in an invalid state after the transaction has been executed, and that it is brought from one valid state to another.
- **Isolation** is the property that guarantees that concurrent transactions are executed isolated from each other, and that the result of executing a set of concurrent transactions will result in the same state as it would executing the same transactions serially.

- **Durability** guarantees that once a transaction is committed, it will remain committed despite failure to the system.

Relation Database Management Systems (RDBMS) fully adapt these properties. Such a strict transaction processing model is usually only needed in certain use cases, such as databases in banks or stock markets that always have to provide the correct, same and up-to-date data [16].

The CAP theorem [17] asserts that any networked shared-data system can have only two of the three desirable properties: Consistency, Availability and Partitioning Tolerance. This theorem (or dilemma) makes it particularly hard to scale RDBMS while sustaining the desired ACID properties. To maintain the strict consistency, only either availability or partition tolerance can be prioritized. In distributed RDBMSs, two-phase commit protocols are generally used to ensure consistency and atomicity in transactions.

In many large scale applications priority of availability and partitioning tolerance often have economic justifications, as unavailability of the system can lead to financial losses. A more relaxed consistency model is called for in these scenarios. This way, all readers will receive *some* data, but not necessarily the last written data. Usually, these systems adapt the BASE (Basically Available, Soft state, Eventual consistent") properties. This means that an application works basically all the time (Basically Available), is not necessarily consistent all the time (Soft state) but it is guaranteed that without any new updates to an object, eventually all read requests on that object will return the same data (Eventual consistent) [16].

2.2 Database Scalability

Centralized relational database systems have for many years been the traditional approach to store and access data. Locating data in a centralized physical location drastically decreases complexity regarding problems such as consistency, availability and partitioning tolerance. A single server quickly becomes a single point of failure and expensive to scale. Scaling up single node RDBMSs (vertical scaling) is done by adding more processor, memory and external storage. Alternatively the RDBMs can usually be scaled out by replicating the entire database on multiple nodes. If the database often is update, maintaining ACID properties can become a challenge. Despite this, there are RDBMS out there that can scale to a rather acceptable scale such as Azure SQL that scales to 100s of SQL databases horizontally [18].

Distribution of the data across multiple physical locations (nodes) interrelated over a computer network is a possible solution to increase the performance and fault tolerance of a storage solution. As data now no longer reside in a single physical location, new, more complex, problems occur. These problems are often related to the CAP theorem.

NoSQL (Not only SQL) databases are a more modern approach to solve the loss in performance when guaranteeing ACID properties on large scale distributed database management systems. By introducing a database model without relations (key/value) and a simple interface (such as put/get), scalability problems are simplified by reducing the complexity of the data model. NoSQL database systems are developed to manage large volumes of data that do not

necessarily follow a fixed schema, where data is partitioned among multiple servers. To maintain good availability and performance, NoSQL databases do not guarantee ACID properties for their transactions. On the contrary, it is assured that the state of the database is following the BASE properties. Dynamo [19] fundamental NoSQL data storage that came up with the concept of using consistent hashing [20] in their partitioning and replication scheme, which has been adopted by many of the modern NoSQL databases [21].

2.2.1 Dynamo Distribution Technologies

Amazons Dynamo is a highly available key-value store, and its distributed technologies are widely used in many of the large-scale distributed storage solutions today. These are some of the fundamental techniques introduced in the Dynamo paper:

Partitioning and Replication

To be able to scale incrementally, Dynamo provides a mechanism to dynamically partition the data over a set of nodes in the system. This partitioning scheme relies on consistent hashing to distribute load and responsibility across multiple nodes. The consistent hashing technique treats the output from the hash function as a ring, where the largest hash value wraps around to the smallest. Nodes in the Dynamo ring are located at a random value within this hash range, as illustrated in Figure 1. We define a key range to be the region of hash values between two nodes. A node has responsibility of data items identified by a key which hashes down to a value residing in the key range between itself and its predecessor (first node counter-clockwise in the ring). This node is called a coordinator node for the respective data item.

Data is replicated N times throughout the ring to provide high availability and durability. The coordinator node has the responsibility of replicating the data N-1 times on its clockwise successor nodes. By following such a replication scheme, each node has the responsibility of items located in a key range between itself and its N^{th} predecessor. The list of nodes that are responsible of storing a given key is called the keys *preference list*.

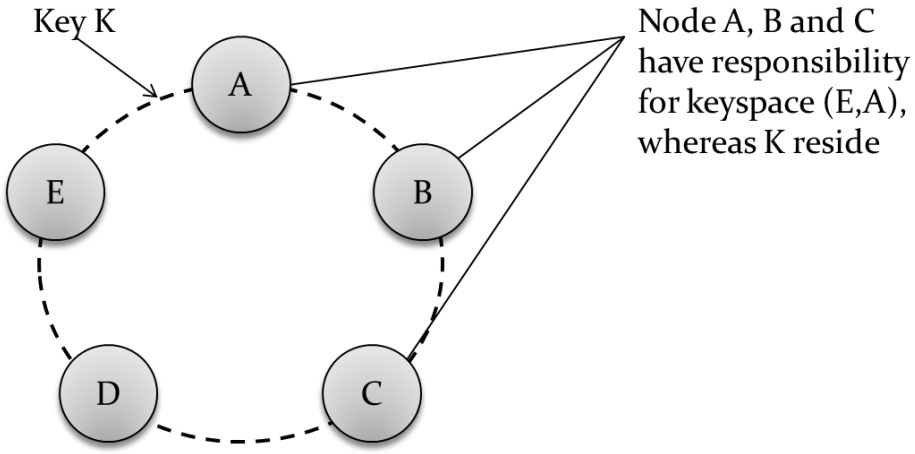


FIGURE 1 – PARTITIONING AND REPLICATION OF KEYS IN DYNAMO RING

Read and Write Quorum

Read and write operations in the Dynamo ring include the first N healthy nodes present in the preference list. A consistency protocol, similar to those used in quorum systems, is used to maintain consistency between nodes handling replicas of data. A quorum system define the minimum number of participants *r* that need to agree for a read operation to be executed and the minimum number of participants *w* to agree for a write operation to be successful, where *r+w > N* [22]. Dynamo allows *r+w <= N* to achieve better latency, as operations will be dictated by the slowest replica.

Dynamo uses vector clocks [23] keeping track of the ordering of read and writes events, and for versioning the data. Upon receiving a new write request, the coordinator will generate a new vector clock for the new version of an object, storing the version locally. The coordinator will then issue the new data and the new vector clock (versioning) to the N highest ranked nodes in the preference list. If the coordinator receives at least w-1 responses, the write-operation is considered successful. Nodes will not respond if they have data with a higher versioning number than the number issued. Same procedure is followed for read operations; the coordinator receiving the get request will request the data from all N highest ranking healthy nodes, and waits for R responses with the same, highest, vector clock versioning before responding the clients with the data. If the coordinated identifies multiple versions of the data, it returns all data to the nodes that are causally unrelated. The data is then reconciled and the newest version is persisted.

Handling Temporary Failures

Dynamo uses techniques such as *sloppy quorum* and *hinted handoff* to handle temporary node failure. Sloppy quorum means that all read and write operations are performed on the first N *healthy* nodes from the preference list, which does not have to be the three nodes following the key range in the dynamo ring.

Given Figure 1 where the replication factor is N=3: nodes A, B and C have the responsibility for key K. If for example node A not is reachable at the time of writing the data represented by key K, a handoff will occur, where node D (successor of the last node, C, in the preference list) will work as a handoff node for A. The data will be temporarily written to D instead of A with a hint in its metadata of its original owner A. When detecting that A has recovered, data written to node D with hints of A will be delivered to A and deleted from D.

Recovering from Permanent Failures

Hinted handoff works best in scenarios where system membership churn is low and node failures are transient, and to recover from replicas being permanently unavailable, Dynamo implements an anti-entropy protocol to keep replicas synchronized. To detect inconsistencies, Merkle trees [24] are used; each node has one Merkle tree for each key range that it hosts. This way, nodes can find out whether keys are consistent by comparing leaf node hash value. To detect whether the whole key space is consistent, they can exchange and compare the root node of the Merkle tree. If inconsistency in the root node occurs, traditional tree traversal and comparison further down in the tree can be done to locate the inconsistent data.

Membership and Failure Detection

In Dynamo, nodes join or leave a ring manually; an administrator uses a command line tool or a browser to add or remove a node. The node that handles the membership-change request persist this change information to a log before it propagates the information to other members of the ring through a gossip-based protocol. By using this type of protocol, the global view of the state of the ring will be eventually consistent. Partitioning and placement information also propagates via the gossip-based protocol. Each node will this way be aware of key spaces and their coordinators, enabling direct forwarding of read/write operations to the correct set of nodes.

Dynamo uses a local notion of failure detection, where individual nodes that is not able to communicate with each other simply consider the unresponsive node as failed, using alternate nodes to serve the request. The node periodically retries the unreachable node to check for its recovery. If no other nodes can communicate with the unresponsive node, no nodes will forward requests to it until it becomes responsive.

2.3 Data Store Classification

2.3.1 Relational databases

Relational databases store well-defined relations (tables) where all tuples (column) names and types must be defined in advanced [25]. Every row within a table contains the same set of columns, and the properties of these columns must be defined in advanced. Properties are typically indexes, uniqueness, nullable values, keys, etc. Each relation can also contain rows that pose as a *foreign key* that refer to other relation's *primary key*, creating a link between two different relations. This way, complex queries can be made across multiple relations, resulting in new relations based on the query.

SQL is a special-purpose programming language designed for managing data in relational database management systems. Traditional RDBMS guarantee that transactions executed sustain ACID properties before they are persisted. These are relative easy to handle in a centralized environment, but quickly becomes hard to obtain when horizontally scaling the entire (or parts of) the database onto several decentralized or distributed servers, resulting in performance loss due to the strict consistency model.

2.3.2 NoSQL Databases

NoSQL (Not only SQL) databases are a more modern approach to solve the loss in performance when guaranteeing ACID properties on large scale distributed database management systems. By introducing a database model without relations (key/value) and a simple interface (such as put/get), scalability problems are simplified by reducing the complexity of the data model. NoSQL database systems are developed to manage large volumes of data that do not necessarily follow a fixed schema, where data is partitioned among multiple servers. To maintain good availability and high performance, NoSQL databases do mostly not guarantee ACID properties for its transactions. On the contrary, it is assured that the state of the database is "Basically Available, Soft state, Eventual consistency" (BASE).

Schema-Less

Tables in NoSQL databases typically do not have a pre-defined schema or a very limited pre-defined schema such as in BigTable [26]. Each record may have a variable number of attributes, while some databases do have the ability to pre-define a set of mandatory attributes. The content in a NoSQL database and their semantics are entirely enforced by the application using the data storage.

2.3.2.1 Key-Value Databases

Key-Value databases provide an efficient key to value storage in a schema-less way. Data is accessed through a simple system interface, often by operations such as put(key, value), get(key) and delete(key). The value that is stored can be either structured objects such as JavaScript Object Notation (JSON) objects in Voldemort [27] or unstructured binary objects as in Dynamo [19]. Unstructured data do not have a pre-defined data model to adhere to. Values are not indexed in either of these cases, and queries can mostly be done using the keys. Some key-value databases, such as Scalaris [28] arrange keys in an ordered fashion to support range queries.

2.3.2.2 Document Databases

A document database consists of a series of self-contained, schema-less, semi-structured documents. Each of these documents are considered as entities containing information only related to themselves, without any relationships to other documents. Documents conform to a dynamic model, where fields can be added or removed from each document freely. This means that where a relational database would need to store an empty value for a given field, document databases can simply remove the whole reference to that field. Document databases can be indexed and queried using primary- and secondary-keys. Examples of document databases are CouchDB [29] [30], Amazon Simple DB [31] and MongoDB [32].

2.3.2.3 Big Table Databases

Big Table databases conforms to a model that was initially developed and implemented by Google as a distributed storage system for managing structured data [26]. These types of databases are also known as *tabular databases*. Databases following this model are more similar to a relational storage model; they contain multiple tables built up from rows that consist of a set of values. These are the fundamental similarities, but Big Table databases differ in several important ways [21]:

1) A Big Table database can be looked at as one *big table* with a three-dimensional key-value data store. This table is indexed by a row index, column index and a timestamp. The timestamp is used for versioning of the data.
2) In Big Table databases, rows can have different set of columns. A table's schema can pre-define a set of required column groups, but each row within the table can differ by specific columns within these groups. All columns do not have to be pre-defined in the schema, but can be added or removed dynamically as whether they are needed or not, such as document-fields in document databases.
3) As relations are not supported, columns in Big Table databases are designed to contain thousands or even millions of column values per row.

2.4 Cassandra

Cassandra is an Apache project that is a distributed storage system for managing very large amounts of structured data [33]. This data can be managed across many commodity servers, while still providing a high level of availability as there is no single point of failure. The Cassandra architecture is based on distribution technologies from Amazon's Dynamo [19] and the data model from Google's BigTable [26]. [21] states that there are five key features which Cassandra provides:

1) **Decentralized:** In Cassandra, all nodes are equal peers. There is no single point of failure.
2) **Fault Tolerant:** Following the Amazon's Dynamo replication scheme, data is replicated among multiple nodes (and even data centers) to provide fault tolerance. Failed nodes can also be replaced without any downtime, providing great availability.
3) **Eventually Consistent:** To enable high availability and partitioning tolerance, the system follows an eventually consistent consistency model. Optimizations such as Hinted Handoff and Read Repair are used to minimize the time frame where the system is inconsistent.
4) **Elasticity:** New servers can transparently be added to the database without any downtime, while performance scales linearly with the amount of new servers that are added.
5) **Rich Data Model:** Cassandra support more complex records than in a simple key/value store.

Many large vendors in the world[1], such as Facebook, Adobe, EBay and finn.no, have deployed Cassandra to host their structured data. It is constantly under development, and progress is made at a daily basis. Cassandra has support for storing binary large objects (BLOBs) in its tables, but is not optimized for it, and can handle files of about 64MB of data without splitting it up into smaller chunks[2].

We consider Cassandra interesting in this thesis due to the fact that it is one of the most applied NoSQL solutions, and have been deployed in many different contexts by many different vendors.

2.5 CouchDB

CouchDB (Cluster Of Untrusted Commodity Hardware Data Base) is a document-oriented database management system [29]. It is released under the open source Apache 2.0 license. The data is stored in a schema-free manner, and documents are stored as a series of documents and offer a MapReduce JavaScript-based view model for aggregating and reporting on the data. These documents can each contain a series of fields and values that is independent of any other document in the database, which is in contrast to a traditional RDBMS where the developer have to predefine a scheme that entries must adhere to. CouchDB has built-in support for attaching files to documents. These attachments are binary

[1] http://www.datastax.com/cassandrausers
[2] http://wiki.apache.org/cassandra/FAQ

objects that are stored in a JSON structure consisting of the name of the attachment, the content (MIME) type and the data itself.

One of the things that CouchDB provides differently from many other traditional data storage systems is that the data is accessible through a RESTful HTTP API. Data can be uploaded, edited or fetched through regular POST, PUT, GET and DELETE HTTP requests.

2.5.1 BigCouch

BigCouch is a product created by Cloudant [30] that also it released under the Apache 2.0 license. It provides a highly scalable, available and fault tolerant structure over a set of CouchDB nodes. These nodes are structured into a ring using consistent hashing with the same partitioning and replication scheme as described in Amazons Dynamo. BigCouch also uses the same read/write quorum protocols as described earlier.

We consider CouchDB with BigCouch interesting due to its focus on high availability, the ability of combining meta-data in forms of documents with video content as attachments, and because it provides a native HTTP API.

2.6 OpenStack

OpenStack [34] is one of the world's most deployed open cloud infrastructure platforms. It is deployed in a number of public and private clouds throughout the world. It started out as collaboration between NASA and Rackspace[3], and has ended up in the open source community licensed under the Apache 2.0 license. Essentially, OpenStack is a cloud operating system that controls pools of resources. These resources can be in the form of computation, networking or storage (Figure 2). These resources can be managed and monitored through a web interface called *OpenStack Dashboard*.

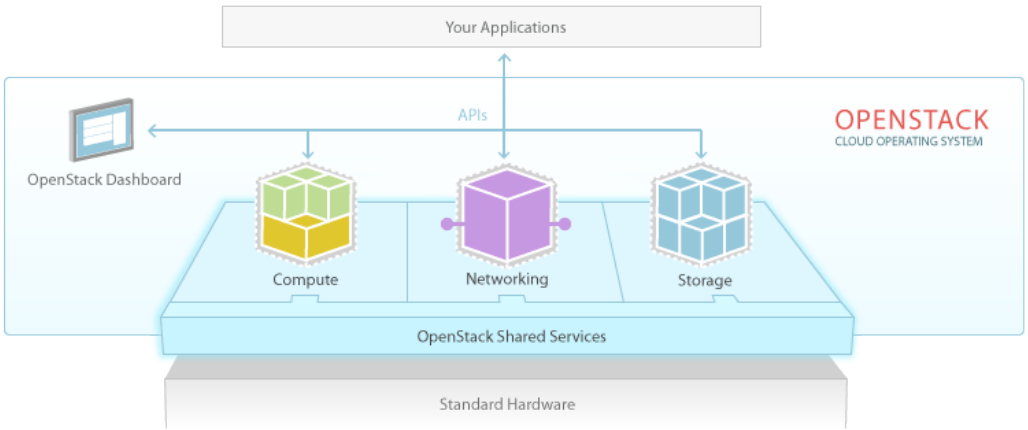


FIGURE 2 – OPENSTACK SERVICES OVERVIEW [34]

Each of the OpenStack services is designed to be loosely coupled and highly configurable through a plug-in architecture, where modules can be added or replaced seamlessly. This way brand new features or functionalities can easily be added to the system, or old modules can be edited to change the behavior of the system.

[3] http://www.rackspace.com/cloud/openstack/

2.6.1 OpenStack Object Storage

Object Storage provides cost effective, scale-out and redundant storage, utilizing clusters of standardized hardware. Its goal is not to provide a file system or real-time data system, but rather provide a long-term storage system for large amount of static data. Data can be retrieved uploaded, maintained and retrieved through an REST API. Authentication and authorization is optional but recommended. [34]

The object storage also provides persistent block level storage service for use with OpenStack compute instances to meet storage needs from virtual instances. This also includes snapshot management for backing up data stored on block storage volumes, including the OpenStack Object storage in the full OpenStack cloud stack integration.

2.6.1.1 Architecture

The OpenStack Object Storage architecture consists of Objects, Containers and Account servers that are organized and replicated in a ring structure. The proxy server is the Objects Storages point of entry from client applications.

Object, Container and Account Servers

These servers each have their responsibility of listing storage in a hierarchical fashion. The account server has the responsibility of listing all containers within an account. Containers contains object-listing without any knowledge where the actual data reside in the storage cluster. Both container and object information are stored in sqlite database files that are replicated throughout the storage base according to the replication scheme defined by the administrator. In the leaf nodes of this hierarchical organization (Account->Container->Objects), we find objects. The Object Server has the responsibility of storing these objects as binary large objects (BLOBs) on local devices with an associated metadata file. The metadata is stored in the file's extended attributes (xattrs), and need to be supported by the underlying operating system.

The Ring

In the OpenStack Object storage architecture, the ring refers to a mapping between the names of entities and their physical location in the cluster. There is a separate ring for account databases, container databases, and individual objects. Despite this, each ring behaves the same way. The location of data in any of these databases is determined through consistent hashing [20] (MD5) of their unique name. To determine the location of "*/account/container/object"* a MD5 hash of this path would be used to locate the node of where the data is to be stored.

Each partition in a ring is replicated *n* number of times (by default 3 times) across zones, and the ring is responsible to handle failure by allocating new devices for handoff in failure scenarios. Each zone can for example represent a set of computers on separate power and network than the other. In large distributed scenarios, they could even be representing datacenters in nations or continents.

Optimally, the rings are evenly distributed across the cluster. When a device is added to the cluster, the rings are rebalanced onto the new device. OpenStack ensures that a minimum number of partitions are moved at a time, and that only a single replica of a partition is moved at a time to prevent any unnecessary bi-effects of adding a new device.

Proxy Server

The proxy server has the responsibility of exposing the public API, handling requests from the public. For each request, it is responsible for looking up location of the requested account, container or object, handling failures, such as requesting for a handoff server if the server the request is sent to is unresponsive. Objects requested or added by a user are streamed directly through the Proxy Server; no data is persisted or spooled in the proxy. OpenStack do not use read or write quorums for consistency. Data written is hashed down to three replicas and persisted to all of the nodes as a part of the critical path before a success is returned. On a read operation, the proxy simply picks a random of the three replicas, asks it for a copy and returns it to the requestor. No versioning is checked.

We consider OpenStack Object Storage an interesting technology to use due to its focus on binary files, its native support for authentication and authorization, its HTTP API and its stack integration.

2.7 Summary

Database models following a strict consistency model, such as relational databases, do not scale very well. As this thesis work with unstructured binary objects in form of video files that are written once, never updated and downloaded multiple times, we have chosen to investigate and compare three open source distributed database solutions in Table 1. All databases have support for binary objects, and do not adhere to a strict consistency model.

TABLE 1 - STORAGE SOLUTION SUMMARY

	Cassandra	**BigCouch**	**OpenStack Object Storage**
OpenSource	Yes	Yes	Yes
License	Apache 2.0	Apache 2.0	Apache 2.0
Written in	Java	Erlang	Python
Protocol	Custom, binary (Thrift)	HTTP/REST	HTTP/REST
BLOB Support	Yes (Up to 64MB)	Yes	Yes
Lookup	Key/Value	Key/Value	Key/Value
Data Model	Big Table	Document	Binary Objects
MapReduce Views	With Apache MapReduce	Embedded	No
Partitioning	Consistent Hashing	Consistent Hashing	Consistent Hashing
Consistency	Sloppy Quorum	Sloppy Quorum	Eventual Consistency
Availability	High	High	High
Partitioning Tolerant	Yes	Yes	Yes
Stack Integration	Stand Alone	Stand Alone	OpenStack

3. Requirement Specifications & Architecture

To successfully design, deploy and evaluate the video storage system, we must first outline a high level system model. In this chapter we will present the technical requirements that this storage solution needs to adhere to, and the context of which this system model fits into our research. An architecture based on our desired the storage properties will be outlined.

3.1 System Model

The video storage system shall serve as a scalable and fault tolerant backend that deliver video content to clients on request. Figure 3 illustrates a high level system model for an end-to-end scenario where files are uploaded to the video data storage through an encoder that standardizes the video format stored in the video storage system. The encoded video files are further delivered in an efficient way to clients on demand.

FIGURE 3 – END-TO-END SYSTEM MODEL

3.2 Muithu

Muithu [6] is a lightweight video and cellular phone based sport notational analysis, where the main focus is decoupling the analysis process from a large team of analysts. This is achieved by adding the head coach into the main loop of the analysis process, equipping him with a non-invasive cellular phone during practices or matches. Multiple video cameras are installed to record the exercise or competition, and each time the coach observes a situation of particular interest that is desirable for post analysis; he annotates a situation on the cellular phone, marking an offset into each of the active recordings. This enables Muithu to automatically cradle only desired video-snippets from the full recordings of a practice or match.

Athletes and coaches can register for an account to watch videos in the Muithu web application built in HTML5. A coach can browse all videos uploaded into the web application, associating them with a set of predefined athletes or event types. He has the ability of adding comments to athletes, engaging in an interactive educational discussion around the video, providing the involved athletes access to the particular video on their home page.

Muithu is agnostic to type of sport, but is currently in use by the Norwegian elite soccer club Tromsø IL. It is used both for match and practice analysis, and multiple athletes are registered and active in the social network.

Every aspect of Muithu is under constantly development with close cooperation and feedback from users. There have been questionnaires regarding opening a public social network where supporters can register and watch videos, but before this can become a reality, the backend

video delivery system need to be re-designed to fit the availability, performance and scalability needs.

We want to relate this thesis with Muithu to provide a context for the use of this video data storage. Figure 4 illustrates a typical deployment of Muithu, where its architecture is divided into three main groups with a total of six layers:

The field group contains two layers of components used to record, annotate and cradle data from an exercise or match. These components are typically a set of cameras and a cellular device used to capture and annotate (1) and a computer to collect these data (2). After data is collected, it is persisted into video data storage (3) in **the cloud** group. This data storage delivers videos to applications (4) or information systems (5) running in the cloud. These cloud services deliver content and services to end-user applications (6), such as web, desktop and mobile applications in **the web** group.

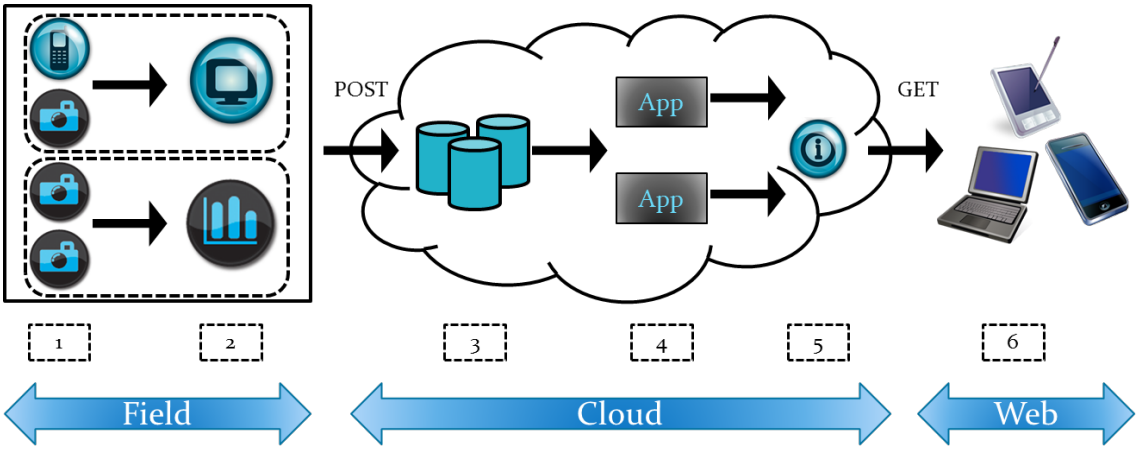


FIGURE 4 – MUITHU SYSTEM ARCHITECTURE

This video storage system will function as layer 3 in this architecture. As this system will provide storage as a service to the Muithu web front-end we need to provide a standardized way of collecting and delivering data. We propose a REST API for both uploading data to – and downloading data from – the video storage. Hyper Text Transfer Protocol (HTTP) is ubiquitous and widely adopted, and provides interoperability with other cloud storage solutions. We do not want to use proprietary solutions that lock us in with a single vendor or a single program language that not comply with standard ways of exchanging data. This video storage solution is completely decoupled from the Muithu system architecture, and could then easily be put into other contexts where storage is needed as a service using standard HTTP libraries to communicate with it.

The Muithu web application is currently built in HTML5 and is optimized for Internet Explorer, Chrome and Safari.

3.3 Storage Properties

There are multiple functional and non-functional properties this video storage system needs to provide. Some functional changes need to be done to the Muithu web application to add support for a decoupled storage solution, as well as there are multiple properties we need to consider when outlining an system architecture for the video storage.

3.3.1 Functional

3.3.1.1 Upload

In the context of Muithu, videos will be uploaded through a HTTP POST API on the web front-end server. The front-end will extract and store meta-data in a SQL database. Thumbnails shall be extracted and persisted, and the video will be encoded in a satisfying video format before it will be persisted in the video storage.

3.3.1.2 Encode

Videos will be encoded in a satisfying and feasible format. Camera encoded video data produces a large footprint and is not optimized for web applications. The encoding function should encode the video in a format that is compatible with playback on both commodity and state of the art computer and mobile hardware.

3.3.1.3 Store

The videos that are persisted in the video storage system should be replicated onto different physical storage devices to avoid loss of data if hardware fails.

3.3.1.4 Download

Clients will request video files from the video storage system for playback in a HTML5 web interface. The downloading process need to be optimized for the encoded video and should be delivered at an acceptable rate such that the client does not experience jitter or buffering.

3.3.2 Non-Functional

3.3.2.1 Scalability

The deployed storage solution will need to have the ability of scaling horizontally in a heterogeneous environment by adding new nodes to the solution. To provide elasticity to our solution, we will consider how easily data can be offloaded into a public cloud environment to offload our private cloud if extreme peaks in number of requests occur.

3.3.2.2 Fault Tolerance

The video storage system shall be fault tolerant. Failure is the norm, and we need to provide a storage service that is available despite node failure. When failed nodes re-join the storage cluster they will be recovered and brought up to date. Content need to be distributed among multiple storage nodes that is interconnected through a network.

3.3.2.3 Consistency

Video files are files that are written once, never updated and read many times. The video storage does not need to adhere to a very strict consistency model. A strict consistency model would lead to less scalability, as extra overhead would occur when reading or writing data.

3.3.2.4 Performance

Videos delivered should not jitter and a smooth playback shall occur. To ensure the best user experience; latency before video starts to play should be at a minimum, and optimally before the whole video is downloaded by the client.

3.3.2.5 Security & Privacy

Authentication and authorization shall be provided when accessing videos. The video storage might contain sensitive videos, and users need to be confident that these videos are not delivered to other clients than those that are authorized to view that specific video. We also do not want to leave servers containing video files exposed to direct attacks on a public IP address, and need some kind of technique to stream video files to clients without the client being in direct contact with the server.

3.4 Storage Architecture

In the previous chapter we outlined a sketch of an end-to-end scenario of how videos will be uploaded and downloaded from our system. This sketch builds the foundation for our architectural design (Figure 5). The next step is to decide a suited distributed storage solution to apply to our problem, before deciding how to encode the videos, and how to access them from our web interface.

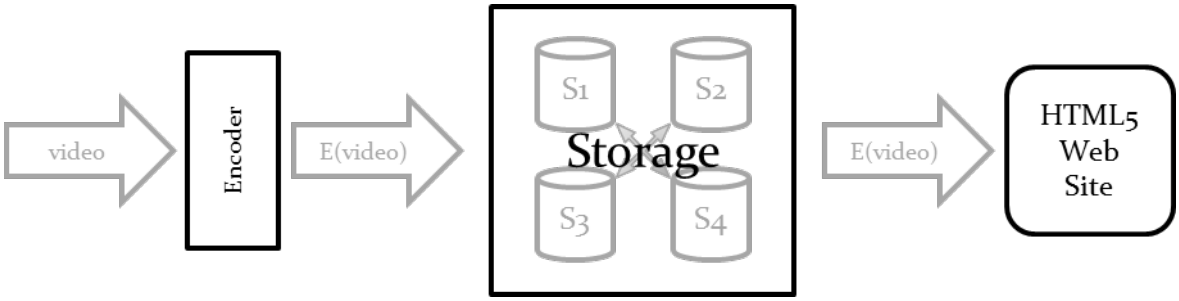


FIGURE 5 – ARCHITECTURE OUTLINE

3.4.1 Video Uploading

Metadata resides in a relational database on the web-front end server in the Muithu web application. Uploading of video files has been done through a HTTP POST API [6], and this mechanism is desirable to keep ensuring that the web application metadata storage is consistent with data persisted in the back end video storage. We outline a desired architecture for uploading the video in Figure 6, where client uploads video through the HTTP POST API in the web front-end (1). The Web front-end temporary stores the file locally and issues it to the encoder (2), which encodes it into the desirable formats and uploads them to the video storage proxy. When the web front-end receives a positive response that videos are persisted, it uses the temporary stored video to extract video-metadata, storing it together with user inputted meta-data into the local SQL server. The web front-end will then be ready to display the video metadata to clients and handle requests on the video file.

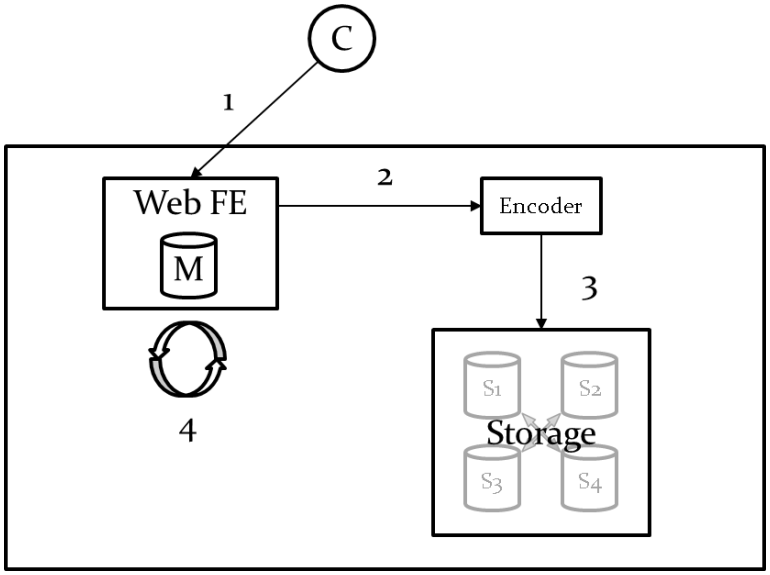


FIGURE 6 – VIDEO UPLOADING

3.4.2 Video Downloading

References to videos will be retrieved by the client when a web page containing the video loads. The client will then be referred to a new location where the video can downloaded from. Such a naïve approach is illustrated in Figure 7.

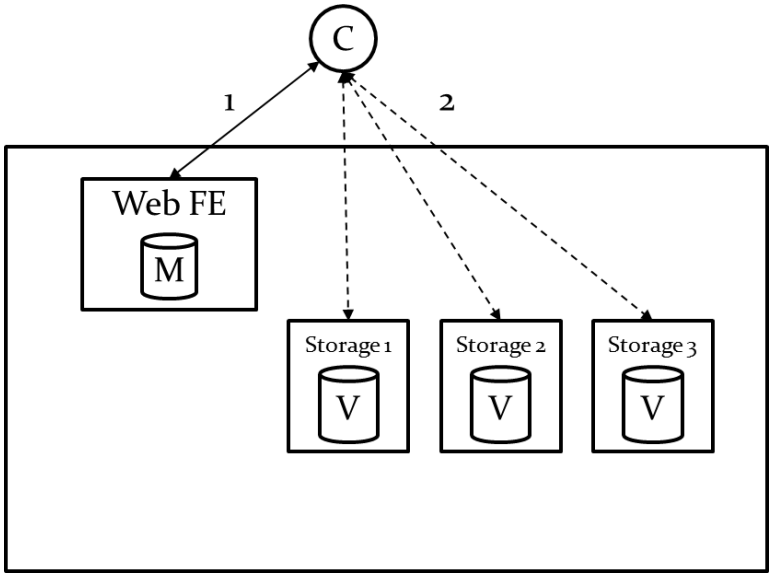


FIGURE 7 – NAÏVE VIDEO REFERRAL

By decoupling the video storage from the web front-end, we can scale out horizontally with a couple of dedicated storage nodes that host the videos. This changes the system properties. We can expect higher throughput, as every video streamed to requesting clients do not origin from the same physical server. By simple replication schemes we can also increase the reliability and availability of our system by replicating the video data onto more than one

node. This way, we can handle nodes failing by utilizing a simple hand-off mechanism. Even though this architecture provides some of our desired properties, there are some problems with this design:

1) A single point of failure quickly occurs in the web front end. The entire data storage, as well as the web application itself, will be unavailable if the node encounters a failure.
2) To be able to implement a fail-over mechanism, the web front-end needs to be aware of the state of all storage nodes. We do not want to further couple our front- and back-end by sharing such states.
3) In this initial design, clients have direct access to the storage nodes. This can be exploited in an attack. We want to locate our storage nodes behind a firewall or on a private network, using proxy nodes to get access to data on the inside. Proxy nodes will become an extra level of indirection that will keep track of node states within the private network.
4) The storage nodes cannot explicitly know whether a client is authorized to watch the video requested. We need some kind of authorization that confirms that a client has the correct privileges to access the requested video.

We can solve some of the mentioned problems above by adding a proxy node to our architecture, locating the storage nodes behind on a private network. The back-end storage will still be available if the web front-end fails, and vice versa. Proxy node would be aware of the underlying structure and state of the underlying storage nodes, being the only point of communication.

3.4.2.1 Proxy

In Figure 8, we propose a more sophisticated architecture for storing our videos. The front-end still works as the initial entry point for the client; the client browses a web page and gets a URL-reference to a video (1). This reference will now be to a *proxy node* that is located on a public network (2), but has access within the private network. If the client is authorized to access the data requested, the proxy node will request the video from one of the storage nodes containing a copy and stream it to the client. The proxy should not persist any data to disk, but can maintain an in-memory cache of the most popular files, to avoid unnecessary requests to storage nodes.

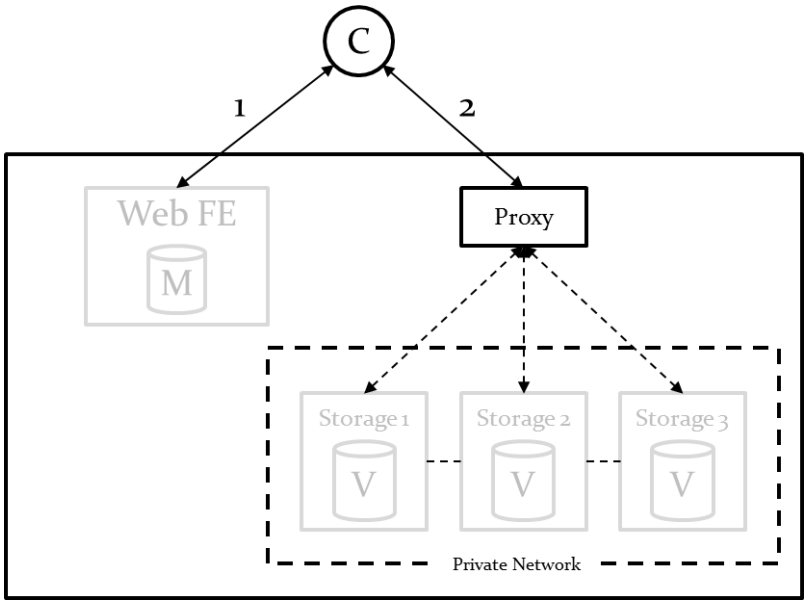


FIGURE 8 – PROXY NODE

By adding the proxy component and locating storage nodes in a private network, we now increase the security and fault tolerance of our system:

Security is increased because we have an entry point auditing video access that is decoupled from the storage nodes. Storage nodes are also located within an isolated private network, ensuring that direct attacks not an option for potential attackers.

Fault Tolerance is also increased, as the proxy can keep track of the state of the nodes, assigning hand-off nodes to fail over when a node goes down. The system as a whole is not as error prone: the video storage system is now completely decoupled from the application using it. Video delivery can be carried on if web front end fails and web content can still be shown (with exception of video files) if the proxy node fails.

By adding the extra component to our architecture, we will also potentially increase the **performance** of the system, as the web server can focus on delivering web content while the proxy node is dedicated to deliver video content. This also allows hardware to be specialized for its purpose.

Having multiple layers in the architecture also enables the administrator to prioritize what to spend **money** on when new hardware is needed. Money can be spent on storage (hard drives, solid state disks, etc.) when the storage layer is to be extended, while CPU, memory and network components have a higher priority when deploying a new proxy node.

We quickly observe that adding a single proxy node forms a bottle neck to our architecture, and if number of requests increases, serving binary data to all requestors can become a challenging job to carry out. Especially for video streaming where the minimum amount of bits per second streamed must exceed the bitrate of the video, for every connection. We do also still have a single point of failure for the back-end video storage in the single proxy.

3.4.2.2 Load Balancer

This problem can be solved by using multiple proxies, but we still only want to have a single address to use to access our database. By adding yet another level of indirection to our system either by [35]:

1) Setting up a DNS server that resolves a single address to multiple IP-addresses using a simple round robin technique to load balance.
2) Using a server that load balances by redirecting incoming HTTP requests to a set of IP-addresses.

For scalability and fault tolerance purposes, we propose the final architecture illustrated in Figure 9. In this architecture, we still have a single point of failure in the load balancer, but it is a simple component, and can quickly be replaced by a new node with the same list of IP-addresses.

Clients will (as before) get a video reference from the web front-end (1). The end-point of this reference will always be the load balancer (2), which will redirect the request to a proxy node (3). The proxy node will authorize the client and fetch the video file from one of the storage nodes within the private network, and respond with the data to the client (4).

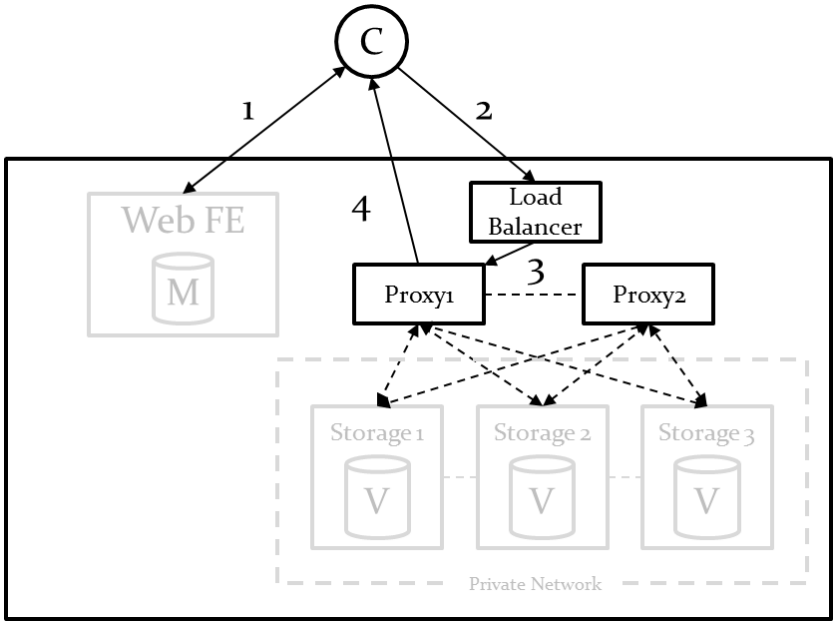


FIGURE 9 – FINAL PROPOSED ARCHITECTURE

3.5 Summary

We have outlined a complete architecture of the desired video storage solution that we wish to deploy. Communication with the storage solution will be done through one or more proxy nodes, and data operations are desired to be done through a REST API. The architecture scales horizontally in both the proxy and storage layer, and extra hardware can be used where it is needed in either of these two layers. Security and privacy is maintained by keeping storage nodes within a private network and inaccessible for potentially attackers.

4. Design & Integration

In this chapter we will find a proper storage solution that adheres to our requirements and desired architecture. We will design how our Muithu web application should work with this solution, deploy the chosen storage solution on some of the nodes in our cluster and finally integrate it with Muithu.

4.1 Storage Solutions

We have considered three popular storage solutions that might adhere to our architecture: Cassandra, CouchDB with BigCouch and OpenStack Object Storage (summary given in Table 1, page 15). Cassandra and BigCouch are both storage solutions that support more complex data models than a simple object storage, and by utilizing one of these makes it possible to also relocate video metadata from the sport analytics web application onto the same storage solutions as the video binaries itself, which limits later maintenance to a single storage system. On the contrary, it will most likely be more effective to use a storage solution specialized for binary objects that seldom are updated, such as OpenStack Object Storage. Proxies and authentication/authorization techniques are also natively integrated in the OpenStack Object Storage solution.

Cassandra does not natively have a HTTP API and is limited to a maximum size of 64MB for each binary. BigCouch do provide all the properties that we want, including large binary support and HTTP API, and we could potentially use it together with some third party proxy such as HAProxy[4] with some kind of implemented security layer on top of that using access control lists or digital signatures to validate user access.

In both Cassandra and CouchDB, we consider that the complex read/write quorums are unnecessary overhead that we do not want or need in our video storage. All solutions follow an eventual consistency model, but since OpenStack does not validate versioning in any way, it is desirable in the context of video files. Video files are typically files that are written once, never modified and read many times.

All of the proposed solutions are freely available under the Apache 2.0 license. This makes all systems highly configurable, as changes can be done with the source code: extra functionalities can be added, existing functionalities can be rewritten and components can easily be removed and replaced by other components using the same API. BigCouch is written in Erlang, a programming language that is optimized for building massively scalable soft real-time systems with requirements on high availability[5]. Erlang is not as widely adopted as Java (Cassandra) or Python (OpenStack), and we conjecture that it will be easier to work with one of the latter two programming languages in the future if some functionalities or components were to be replaced or modified.

Looking back at the Muithu system architecture in Figure 4 on page 18; storage, applications and services are illustrated to run in the cloud environment. By using OpenStack Object

[4] http://haproxy.1wt.eu/
[5] http://www.erlang-embedded.com/?page_id=127

Storage as our video storage solution, we will provide easy integration with a complete OpenStack deployment on our cluster at a later time.

We conclude that the best candidate for our architecture and desired properties are the OpenStack Object Storage. This chapter will continue explaining how we have deployed and configured this storage solution, and how we have integrated it into the Muithu architecture.

4.2 Video Uploading

Videos uploaded through the web front-end web server need to be processed and persisted in the back-end video storage solution. These videos will in general be recordings of matches or practices, and will be video files that initially are encoded by a recording device. The uploading pipeline to be implemented on the front-end server is illustrated in Figure 10.

Videos are uploaded through a HTTP POST API on the web server (1). Using a REST API enables us to easily implement an uploading portal on top of this in the web interface for events that require manual uploading. This part of the web application can also only be restricted to privileged users to avoid unauthorized uploading of videos. When the web server retrieves the uploaded video, it will immediately persist a copy locally (2). This copy will be encoded into a desired video format (3) before it is uploaded to the video back-end storage (4). After a successful response from the back-end, thumbnail (5) and metadata (6) will be extracted. Finally information, relations and references will be stored in a SQL database (7) before displaying and enabling the video to be requested through the web front-end (8). The encoder has previously been presented as a separate component in the architecture, but we choose to integrate it with the web front-end as we do not have any spare node to dedicate for such a component.

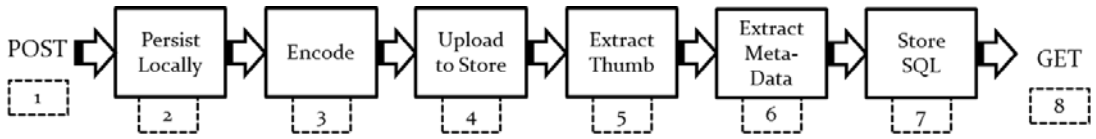


FIGURE 10 - UPLOADING PIPELINE

4.2.1 Encoding Scheme

In early stages of the development of Muithu, videos were encoded on the client side before they were uploaded to the web server. We wish to streamline this and ensure that all videos are encoded in the same format independently of the client that uploads it. The encoding scheme previously used client side was 1080p HD resolution (1920x1080) with a bit rate of 2048kbps. Using this scheme has resulted in multiple issues, such as very poor performance on old hardware, while first generation iPhones and iPads have not been able to playback videos at all.

We have chosen to reduce our video resolution to 720p (1280x720) to support smoother playback on hardware that is not state of the art. YouTube recommends a video bit-rate of 5000kbps for videos with 720p resolution [36]. We have had users of Muithu to evaluate the

new video encoding (keeping the same 2048kbps bit-rate). Feedback has been positive, and current active users are satisfied with the quality of both the image and the playback. We continue to use the same bitrate to maintain the scalability property described in Muithu by keeping the footprint low [6].

For an HTML5 web application, the video encoding decides which browsers that are supported and how smooth the video playbacks on different devices. The Muithu web application is designed to run best in Internet Explorer, Safari and Chrome, which has common support for the MPEG-4 video format. We choose to use this format to maintain support for the largest browsers.

Mpeg-4 Format

A rough outline of how the MPEG-4 video format looks like is illustrated figure 11. A MPEG-4 file consists of three major parts (or atoms) [37].

1) The first part is always the *ftyp* atom, which contains information about the file type and basic versioning of atom structures.
2) Secondly, the *moov* atom defines the timescale, duration, display characteristics of the movie, as well as subatoms containing information for each track in the movie. The optimal location of the moov atom depends on the selected delivery method, and it can be located either before or after the actual media data (beginning or end of the file).
3) The third part is the *mdat* atom. This atom contains the actual media data.

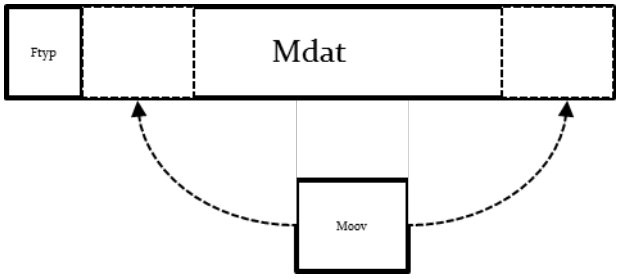


FIGURE 11 – HIGH LEVEL VIEW OF MPEG-4 VIDEO FORMAT

The location of the moov atom is especially important for progressive delivery where the moov atom data must be received before playback can begin.

4.3 Storage Cluster

Our cluster consists of 16 Dell blade servers, each having 2 Intel Xeon E 5335 with 4 cores running at 2.00GHz, and 8GB of main memory. All of these computers are connected to a private network interconnected with a public accessible front-end server. This thesis cannot use all blades in the cluster, as some are reserved for other research projects. We will try to use a minimum amount of servers to deploy the video storage solution. Storage nodes can be added at a later time if there is need for more storage.

The cluster front end computer has an Intel Xeon CPU E5335 2.00GHz with four cores, 16GB physical memory and Gigabit network card. This computer runs Ubuntu Server 12.10.

4.3.1 Metal as a Service (MaaS)

Metal as a Service (MaaS) is yet another service model in the world of cloud computing that is created and coined by Canonical [38] to help facilitate and automate deployment physical servers in *hyper scale* computing environments. The MaaS model finds it place below IaaS (Infrastructure as a Service) in the service hierarchy, and is designed for horizontally scaled environments such as big data workloads and private cloud solutions. The MaaS hardware provisioning technology is built into Ubuntu Server, and provides necessary dynamic allocation of physical resources, reserving physical nodes on demand. After a node is allocated, the computer will install itself automatically from an image of choice on the next boot attempt using Preboot Execution Environment (PXE). This image can either be a standard image of Ubuntu Server, or be a highly customized image. This hardware provisioning can in example be combined with Ubuntu's cloud deployment tool, Juju [39]. MaaS supports deployment of OpenStack, Hadoop, CloudStack, Load Balanced Web and Cloud Foundry, which are infrastructures are often deployed on farms or clusters of physical servers.

In a private network environment, the MaaS server will own the DHCP of that network, acting as a DHCP server for any devices connecting to that network. Nodes added to the network need to be configured with Wake On LAN and PXE boot, with PXE boot as the default boot device. When booting a computer connected to the private network through PXE, the node will register itself with the MaaS server as a new node before shutting down. This node is now declared and can be accepted manually by the MaaS administrator, and become ready to be allocated.

An alternative to use MaaS is to manually install and configure every single node in the cluster.

4.3.2 Network Topology

Figure 12 outlines the topology of our network configuration. We have the cluster front-end node running as the MaaS server, interconnected to both a public and private switch. One of the blade servers are reserved to work as the proxy/authentication/cache server and is connected to both a public (with a static public hostname) and private switch, while the three storage nodes only are connected to the private network, and is not accessible from the outside world. We choose to use a single proxy node in this current deployment, as we only have single Gigabit out-link. Adding more proxy nodes would maybe add support for handling more connections per second, but throughput out would be limited to the Gigabit link.

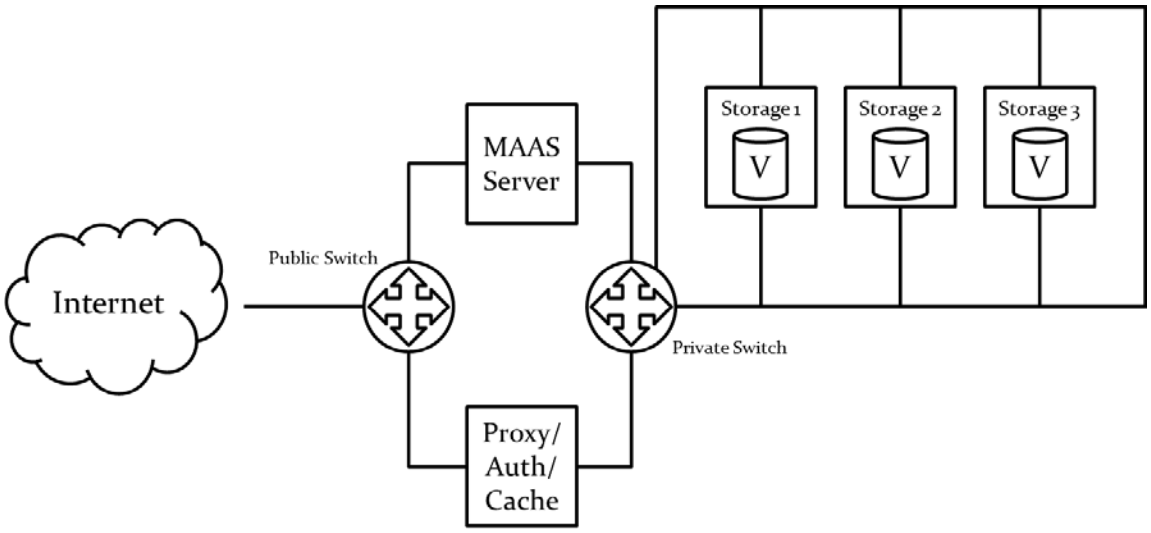


FIGURE 12 – OPENSTACK NETWORK TOPOLOGY

We explored the option of deploying the OpenStack Object Storage through using Juju (Ubuntu's cloud deployment tool), but encountered some problems with the automatic installation, configuration and linking of the proxy and storage nodes. Automatic linking of nodes in the storage ring failed due to inconsistence naming references in the configuration. Alternatives to solve this problem were either to manually reconfigure all nodes or to discard the Juju-deployed version and start over from scratch. As we at this moment did not have the competence to debug configurations in the installed version, we chose to discard usage of Juju to both have more control over the deployment configurations and to learn more about OpenStack in the process.

4.3.3 OpenStack Object Storage

4.3.3.1 Proxy Node

Our cluster is interconnected through a single Gigabit switch, so we have only deployed a single proxy node for our solution. From our proxy node, we have *built a ring* with a replication factor of 3 and added our three storage nodes to it. This creates a larger end-footprint than storing a single copy, but failure is the norm and we do not want to lose data. Replicating data three times means that we can handle two nodes failing simultaneously as illustrated in Figure 13. In this example, we have three storage nodes with a replication factor of 3, where two nodes fail at the same time. Since we have set the system up with a replication factor of 3, storage node 1 have not only persisted its own content (video 1), but also a copy of video 2 and 3. Requests on these three videos would is among the three storage nodes, but since node 2 and 3 currently have failed, node 1 can serve the video content alone until the other nodes are replaced or brought back up.

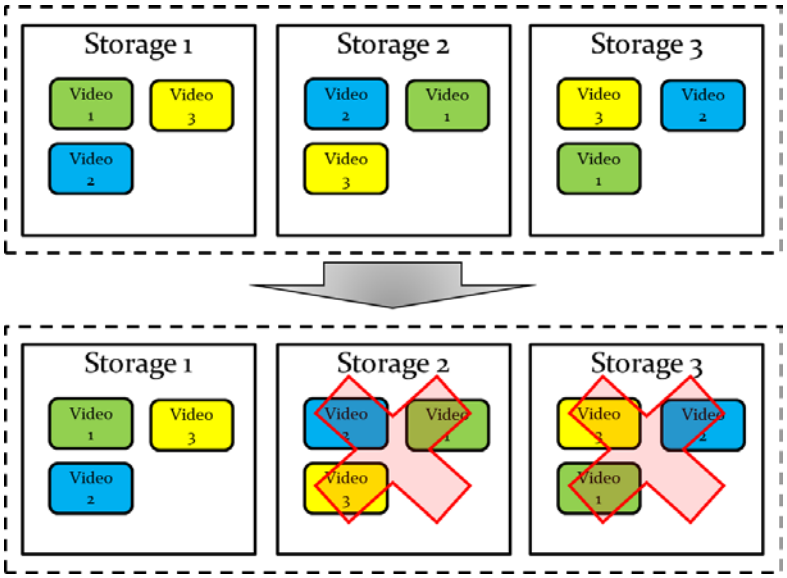


FIGURE 13 - SIMULTANEOUS FAILING OF NODES

Multiple replicas lead to more overhead during consistency checks. As the OpenStack Object Storage architecture only works with random picking of the node to deliver content and does not have any read quorums to validate the versioning on data read, this overhead will not be notable from two to three copies. We prioritize having data stored in three locations to ensure that we do not lose any of the video content if hardware fails.

Each of the three storage nodes is associated with different zones (1, 2 and 3). Replication in Object Storage is configured to be done across zones, and a deployment with replication factor of 3 with a single zone would not work. Zones can be redefined at a later time and the ring can be re-balanced to re-distribute the data across the storage cluster according to new configurations. Managing the ring, such as adding or removing members and re-balancing or deleting the ring, is done with the swift-ring-builder tool that installs with the proxy[6].

The object storage has been configured to progressively deliver files at chunks of 500KB size, which equals about two seconds of video data with the given encoding bit-rate: 1 second of video-data encoded with 2048kbps bit-rate equals about 262144 bytes. By delivering videos at a progressive rate using an encoding scheme where the moov atom is located at the start of the video file, we achieve the possibility of initiating playback in our web interface already after the first chunk of data is delivered [37].

REST API

Communication with the proxy is done through a REST API. Videos are uploaded using HTTP PUT, retrieved using GET and deleted using DELETE. Upon uploading videos, headers in the HTTP PUT request can be set. In the context of video files, the mime type is typically set in the header, such as “video/mp4”. This header is stored with the file in the object storage, and issued with the response when a GET request for that object is made at a later time.

[6] http://docs.openstack.org/developer/swift/admin_guide.html

Authentication

There are two major alternatives to handle authentication and authorization on data operations in OpenStack. We can utilize the OpenStack Keystone Identity Service[7] to handle user accounts, passwords and authorization. In a full OpenStack cloud deployment (Storage and Compute) keystone is a popular service to set up on a dedicated node to handle authentication and authorization many users and groups. Another option is to use the built-in OpenStack *tempauth* for account and access control. Accounts, passwords and roles are defined in the proxy-server configuration file. This solution is simpler, is handled by the proxy node itself, and does not scale to too many users. At the moment, we do not have a large amount of users in our system: we need an administrator account and an account for the web application. We chose to use tempauth, as migrating users to keystone at a later time should not be an issue.

Access control will be carried out by the proxy node. A typical authentication and authorized GET request for an object is illustrated in Figure 14: the client first does an authentication towards the proxy, providing user credentials in the HTTP header with the "X-Storage-User" and "X-Storage-Pass" tags (1). The storage server responds with a successful 200 (OK) or 401 (Unauthorized) HTTP response (2). If the response is 200 OK, the HTTP header contains a token under the "X-Storage-Token" tag (3). This token can later be added in the header of HTTP requests when executing operations on objects within the authorized account (4). A token for a user expires when a new token is created for that user.

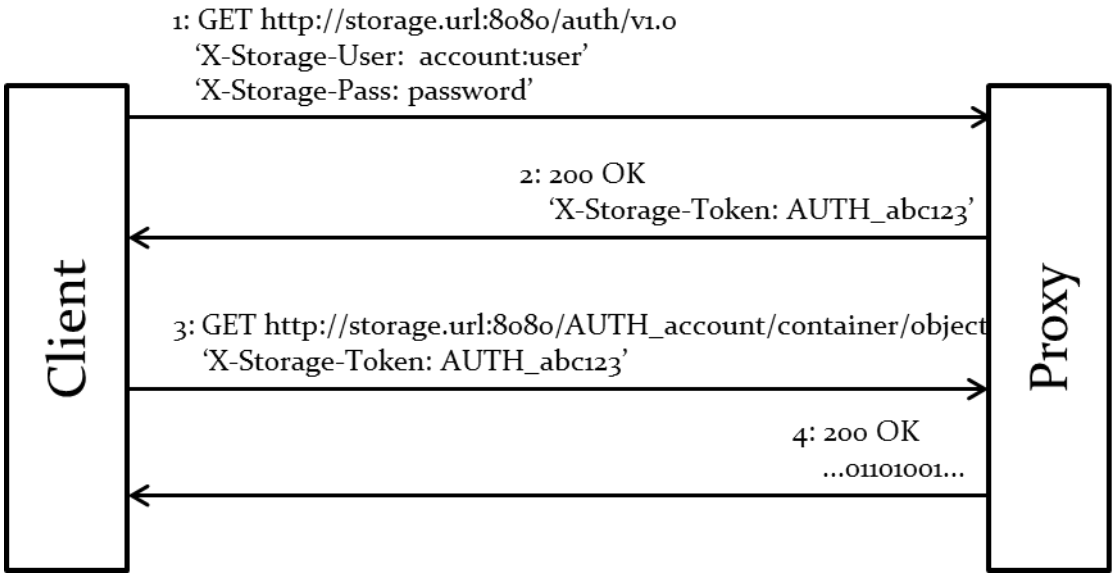


FIGURE 14 – TEMPAUTH ACCESS CONTROL

A problem with this way of accessing objects is that the client needs to have the ability to modify the HTTP header of the object request. This is not the case with browser-accesses such

[7] http://docs.openstack.org/developer/keystone/

as the Muithu web application. OpenStack have provided a solution to this problem by introducing *Temporary URL*. By using a *secret* stored in the account information on the storage cluster, a unique hash signature can be used to authorize object access. This secret can be set by doing an HTTP PUT on the storage account URL (i.e. http://storage.url:8080/AUTH_account) with the X-Account-Meta-Temp-URL-Key header set to the secret, as well as a valid authorization token. The hash signature is a HMAC-SHA1 (RFC 2014) of the valid HTTP method the temporary URL is allowed for ("GET" or "PUT"), the path the object ("/v1/AUTH_account/container/object") and a UNIX timestamp (time since epoch) of when the URL shall expire, where each part is separated by newlines ("\n") in a string. The signature needs to be generated as a hex digest, and ends up looking similar to this: "c8a02bd2efde6950bcdcda7248823d2790f93346".

The valid temporary URL will consist of the URL to the object with the temporary URL signature and its expiry timestamp as a query string, such as:

```
http://storage.url:8080/AUTH_account/container/object?
temp_url_sig=c8a02bd2efde6950bcdcda7248823d2790f93346&
temp_url_expires=1354810106
```

Caching

OpenStack Object Storage has native support for using the free and open source, high performance, distributed memory caching system *Memcached*[8], which is used by high trafficking vendors such as for example Wikipedia, Flickr, Twitter and YouTube. OpenStack Object Storage does not cache any of the actual objects, other than default operating system level caching on the storage nodes. Memcached support is added to cache certain types of lookups, such as auth tokens and container/account existence. We have a limited amount of computers that can be dedicated for caching, but as our proxy node in practice only needs to utilize a minimal amount of its 16GB amount of memory, we have chosen to co-host our proxy with a single node instance of Memcached.

As OpenStack is under the Apache2.0 license and is written in Python, proxy-caching of object data can be added in the future by implementing it manually. This could increase the overhead of processing client requests, and thorough design and evaluation would need to be done.

OpenStack also support client-side caching of objects. When a client requests an object for the first time, it uses an ETag signature in the HTTP header to version the response to the client. If the client has cached the object, the next request for the same object will be contain the ETag signature that was retrieved on the previous request. The proxy will issue a HTTP "304: Not modified" response to the client if there is no newer version of the object in the storage, and the client will use its cached version of the object. If the URL differs in any way, such as in the case of two different temporary URLs for the same object, client-caching will not work.

[8] http://memcached.org/

4.3.3.2 Storage Nodes

Three of the blades have been used as storage nodes. These nodes are equipped with two SCSI disks in RAID 0 of size 126GB. SCSI drives of course provide a minimum performance-gain to disk-accesses, but economic-wise they are not optimal for our set up. Also, the RAID 0 setup is unnecessary, as we already have redundancy of data stored on the storage nodes by distributed data replication. Even though this OpenStack Object Storage is a part of this thesis, we do not want to change BIOS configurations on the blades. As we do not have multiple physical disks accessible, each node have formatted (with XFS file system) and mounted a partition of 80GB to use for this project.

Similar to the proxy nodes, each storage node is configured to deliver data in 500KB chunks. This is to avoid that the proxy have to buffer up or split up data before it sends it to the client. By having the same chunk size in both layers, the proxy can simply forward the data to the requesting client without modifying it.

4.4 Video Downloading

Videos are served through a HTML5 web interface. A temporary URL needs to be generated for the client, such that it can be viewed in the video player in the web application. A temporary URL can either be generated every time a web page containing a video loads with a short expiration time. Alternatively, it can be generated batch-wise (once a day, for example at midnight) and stored in a front-end database to be used every time that particular video is accessed. Both alternatives are valid, but the first will generate more overhead on the server as the URL will need to be generated each time a video is accessed. A client browsing the same video multiple times will also end up issuing requests for the same video with unique URLs, disabling the ability of caching the video client side. This problem is solved by having a longer lasting temporary URL. On the contrary, there might be security and privacy issues by using long lasting temporary URLs, as they provide a signed access to the video. If such an URL were to be posted on a forum or in any other way go astray, unauthorized persons can get access to the video content.

To maintain our strict privacy policies, we choose to generate a temporary URL at every request that is valid for two minutes.

4.5 Integration with web application

After configuring the OpenStack Object Storage and making it accessible by a static hostname, it was ready to go into production with our Muithu web application, serving video files to staff and athletes of the Tromsø IL soccer club. We created an account and user for the Muithu application and set the X-Account-Meta-Temp-URL for this account to be a secret key to enable temporary URLs.

4.5.1 Uploading

The Muithu web application already provides an API for authorized uploading of videos and meta-data [6]. In this version, videos are persisted locally in the same format as they are uploaded in. To further integrate the web application according to the uploading pipeline designed in Figure 10 at page 26, we must implement server-side code for encoding and uploading the video to the video storage system. Thumbnail and metadata are later extracted and inserted to the database. The web application is implemented using the Razor syntax[9] that combines HTML for viewing the data and C# to implement server-side logic [40].

4.5.1.1 Encoding

The encoder in this thesis has been implemented as a C# wrapper utilizing the FFmpeg[10] and qt-faststart[11] tools to encode the videos in a desired format. C# is used for easier integration with the web application. These tools does not have a C# compatible API, so our wrapper run these third party tools as executable with arguments dependent on what is to be executed.

All videos are encoded in a 1280x720 resolution with a bit-rate of 2048kbps. To have fully compatibility with older mobile devices, such as older Apple or Android devices, the FFmpeg *"-profile:v baseline"* profile need to be used when encoding. This setting disables some advanced mpeg4 features, but improves the compatibility. None of the features that are disabled seems to have any impact on downloading or playing the video in the Muithu web application.

FFmpeg does locate the moov atom in the FFmpeg video format at the end of the video file and no encoding options have been found to relocate it to the front of the video file as a part of the encoding process. Qt-faststart has been used to re-locate the moov atom. This tool is created to optimize videos for web site delivery, and the functionality of this tool is to move the moov atom from the end of the file and relocate it to the beginning.

4.5.1.2 Storing

A C# library has been implemented to authenticate, retrieve authorization token and upload video files to the OpenStack Object Storage solution. Video files can either be uploaded from a file or directly from a C# byte array in memory. When video files are uploaded, the "Content-Type" header in the HTTP PUT request is set to "video/mp4" and the "X-Auth-Token" is set to the retrieved token.

[9] http://www.microsoft.com/web/category/razor
[10] http://ffmpeg.org/
[11] http://multimedia.cx/eggs/improving-qt-faststart

4.5.2 Downloading

We have added the functionality of generating temporary URLs to the C# library, based given the path to the OpenStack object. We currently generate a unique temporary URL lasting two minutes at every video request. The implementation of the generation has been implemented according to the description in the design.

4.5.3 Deletion

Functionality of issuing HTTP delete requests on videos have also been implemented and added to the C# library. A copy of the deleted video is stored locally on the video-server as backup, and has to later be garbage-collected manually.

4.4. Summary

We have evaluated that OpenStack Object Storage is a nice fit for our property requirements and our desired architecture. This chapter has designed, deployed and integrated OpenStack Object Storage with the Muithu web interface. We have considered and implemented the necessary libraries and wrappers to support end-to-end scenario from uploading a video through the web interface to secure ways of accessing it in a feasible and streamlined format.

5. Evaluation

In this chapter we conduct a case study evaluating how Muithu have been used for the past year. Experiments are also executed to evaluate end-to-end process from uploading a video file to requesting it in the video browser. Discussions and evaluations according to our desired system properties have also been carried out.

5.1 Notational Usage Case Study

To evaluate and further develop the Muithu system, we have been in a constant feedback-loop with coaches from Tromsø IL. We have studied how and when they use the notational tool, and have adjusted our tools accordingly. The main test subject have been assistant head coach for the last season, Agnar Christensen (current head coach), and the notational device have been a cellular running Windows 7.5 OS. Figure 15 shows Agnar using Muithu during a season match.

FIGURE 15 - AGNAR CHRISTENSEN USING MUITHU DURING A MATCH

This case study observes how Muithu have been used, how many events that are annotated and how many videos that have been extracted. We also investigate if the camera setup has been of a satisfying quality.

5.1.1 Notations during Competitions

Using our system during practices is only one of the areas where our system is applicable. During practices, the application user interface contained multiple players and multiple types of events dependent on the exercise carried out. Each annotation would be related to an event and a player in a tile based interface as illustrated in. We tried to apply the same notational workflow to friendly matches leading up to the season kick-off.

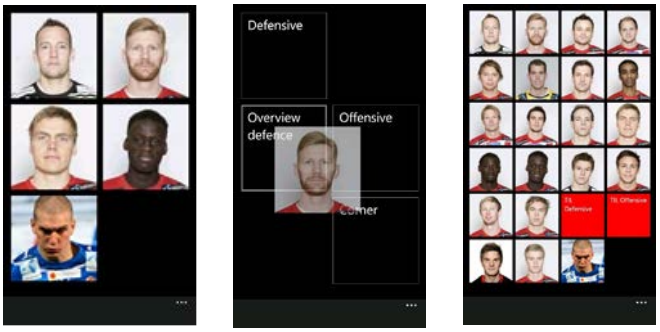


FIGURE 16 – CELLULAR MUITHU INTERFACE [6]

During these friendly matches, we did some valuable observations that lead to fundamental changes in the workflow for the coaches during matches. We observed that setting up a number of predefined players and events prior to a match was hard, as matches are highly unpredictable. Filling the cellular device interface with many players and events also made it hard to use with high precision in a stressing environment such as a competition; the user interface consisted of many tiles that had to be dragged around the screen to annotate the correct player with the correct event type. This workflow is illustrated in Figure 16. This led to many players being tagged with wrong event types, and also wrong players being annotated. This quickly led to changing the device to present two types of events (offensive and defensive) and no players during matches. This way, the coach can do a coarse grained filtering during the match without using a lot of time looking at the device to determine which player and which event type to annotate. The coach can then later use a desktop or web application to do a finer grained filter and organization of the video snippets.

Since the season kick-off, Tromsø IL has been using the Muithu notational system and web interface at close to every home match, both in cup and league matches. During the past season, we have observed how the system has been used, listened to feedback from the club and adapted our development accordingly.

We have recorded each match using at least four cameras with the two latest matches being recorded with as many as six. Our camera setup is illustrated in Figure 17, where camera 1, 2, 3 and 4 have been used in all matches while also camera 5 and 6 have been added in the two last matches. Camera 1 and 4 covers the area in front of goal and the keeper, while 2, 3, 5 and

6 are angled towards their closest goal to cover the part of the pitch closest to their locality from above.

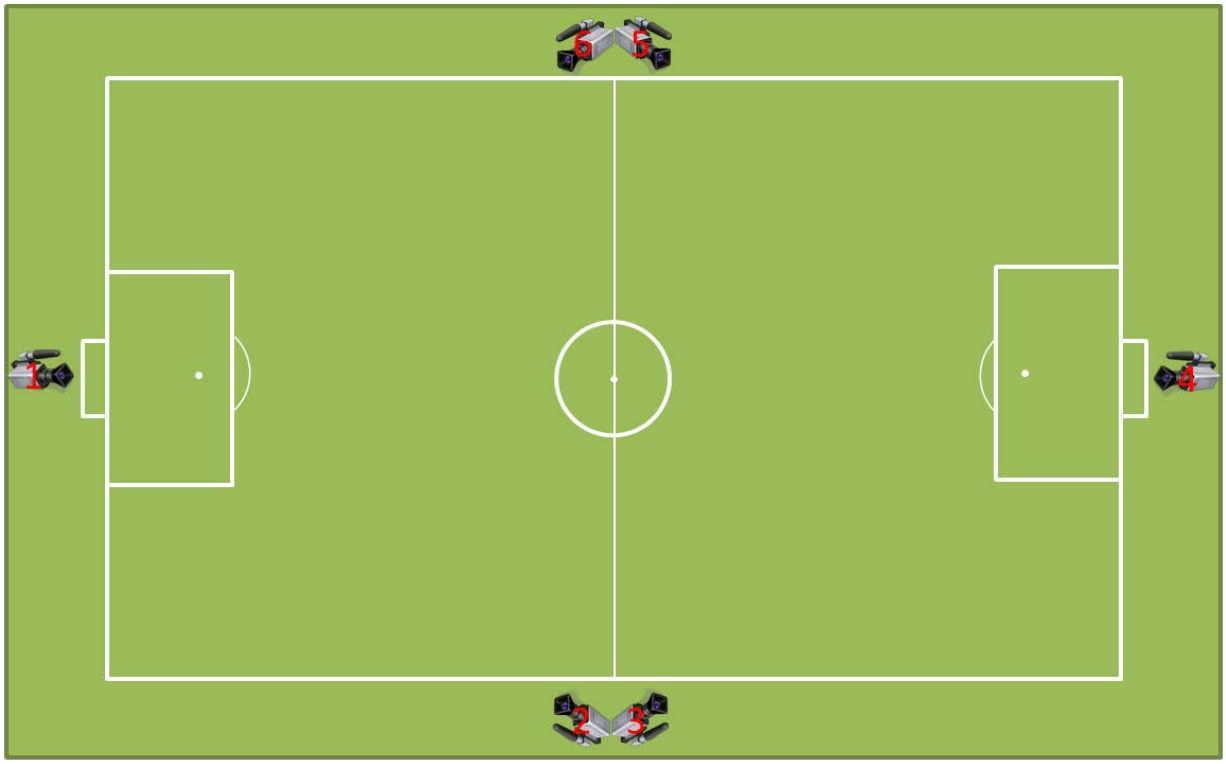


FIGURE 17 - CAMERA POSITIONS DURING CASE STUDY

In Figure 18 we illustrate number of events annotated by Agnar Christensen during home matches in the 2012 season. We observe that Agnar at average annotate close to 15 situations each match. We observe a relation between the performance of the team and the number of events annotated:

1) In the match with most annotations made against Hønefoss, 28 events were annotated. Tromsø IL struggled with their performance, and the game ended in a disappointing 0-0.
2) While in the match with the least amount of annotations (5) made against Viking, Tromsø dominated the whole match and won comfortably with 5 against 1.

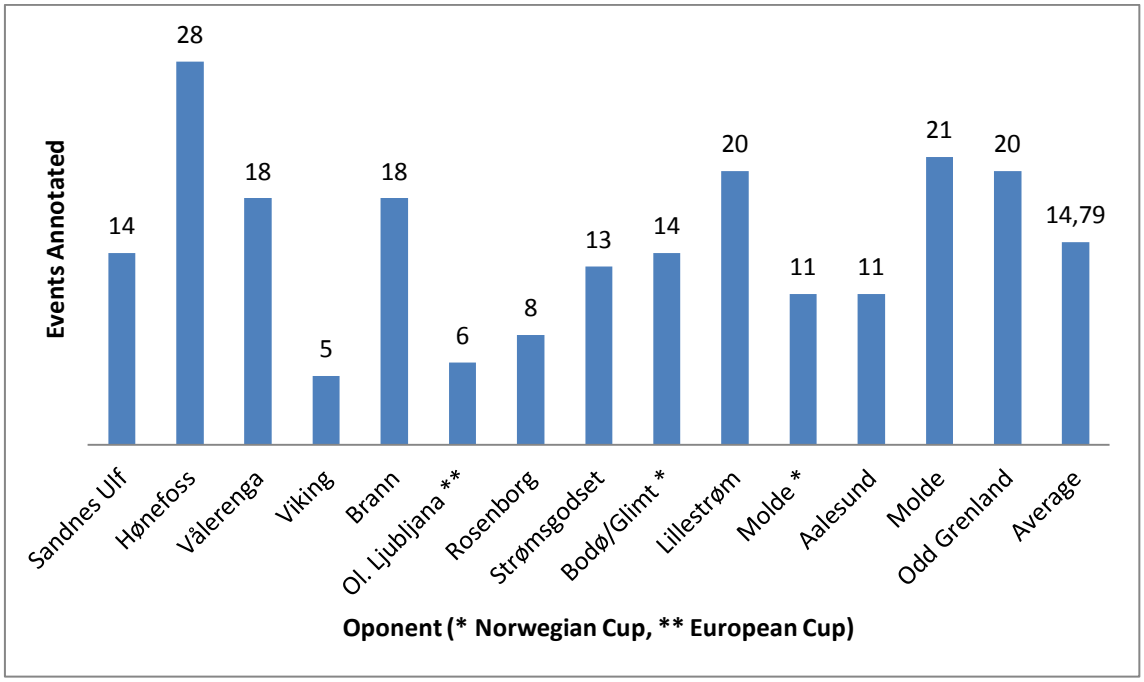


FIGURE 18 – EVENTS CAPUTED DURING MATCHES

By breaking these events into offensive and defensive events, we observe that Tromsø IL have had most of their focus on the offensive part of their game for the past season. Our team only lost one match on their home ground for the past season, indicating a solid defensive work. This is reflected in the focus on the offensive game, as shown in Figure 19. About two thirds of annotations made during the 2012 season were of an offensive type. This can be used in further developing the cellular application with for example making the offensive button closest to the finger (at the bottom). Other choices based on these indications can also be made in the future.

From Figure 20 we can observe the amount of events captured, and how many that actually was kept after removing those that in retrospect weren't interesting after a closer look, or that was annotated too early or too late for the desired event to be captured. We see that our expert had at average almost 50% precision, and almost ~90% at most (10/11 vs. Aalesund).

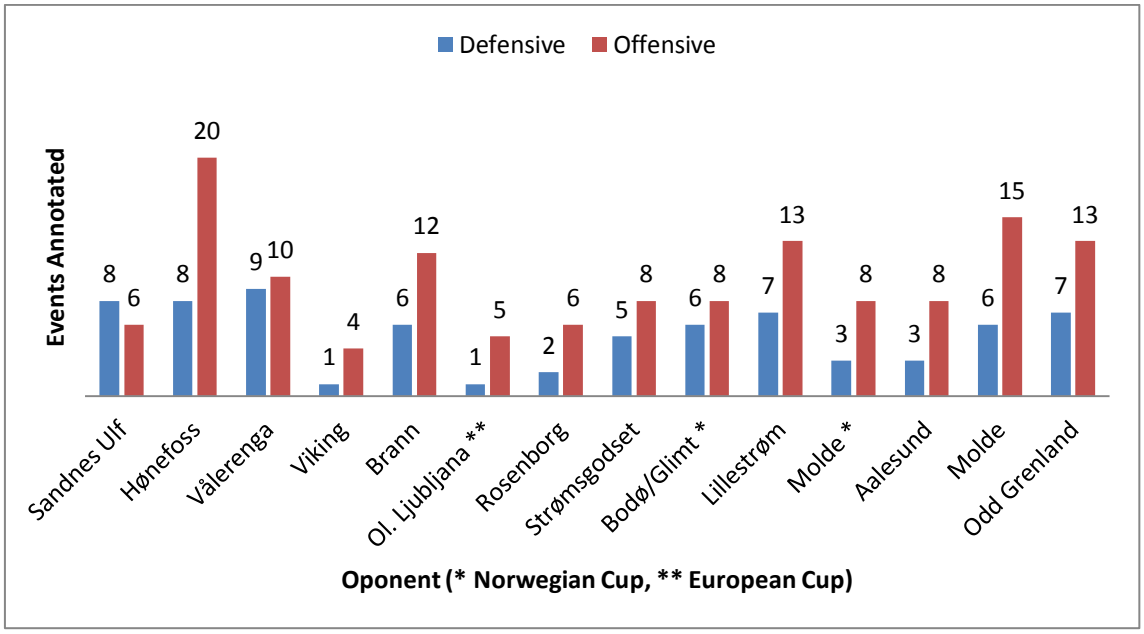


FIGURE 19 – DEFENSIVE VS. OFFENSIVE EVENTS

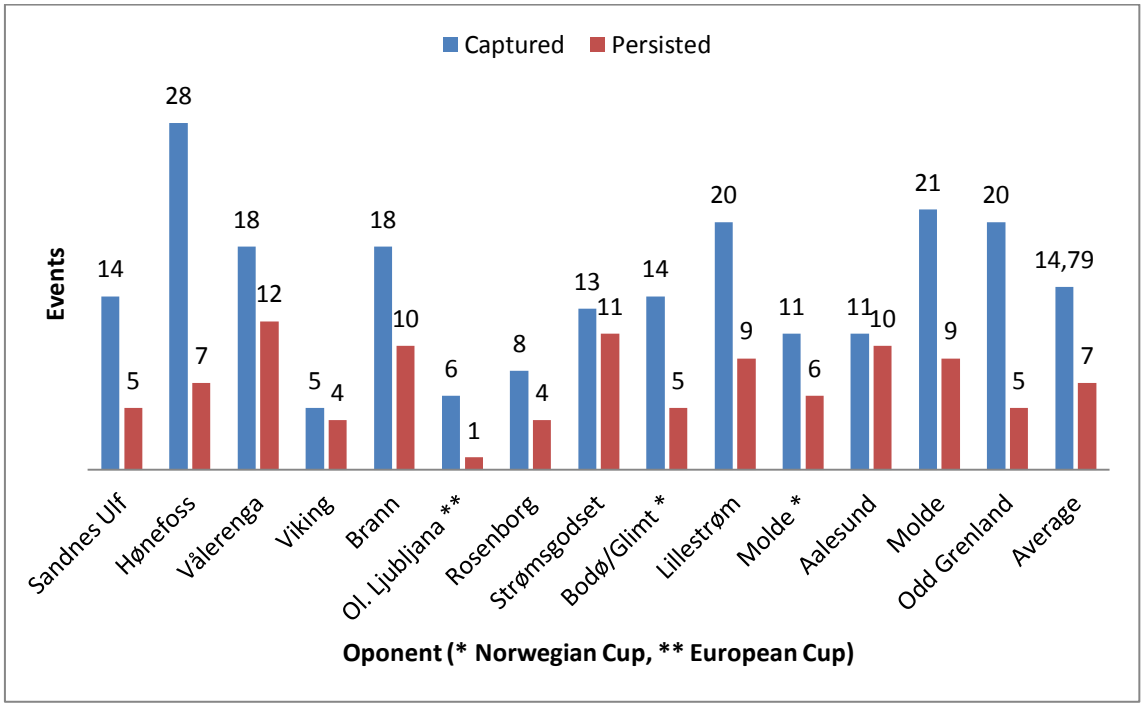


FIGURE 20 - EVENTS PERSISTED

The last two matches have been recorded from six angles, while the rest have been recorded from four. Figure 21 illustrates how many events that still contain 1, 2, 3 or 4 camera angles after unwanted videos have been filtered away. A notion to make from this is that our expert has chosen to keep events while persisting more than two camera angle during the last season; four with 3 camera angles and one with 4 cameras. Some of them have even produced multiple comments on the same situation to different athletes based on the angle the situation is seen from. Many of the processed events also contain recordings from 2 angles, but most of the time, one camera captures the situation of interest.

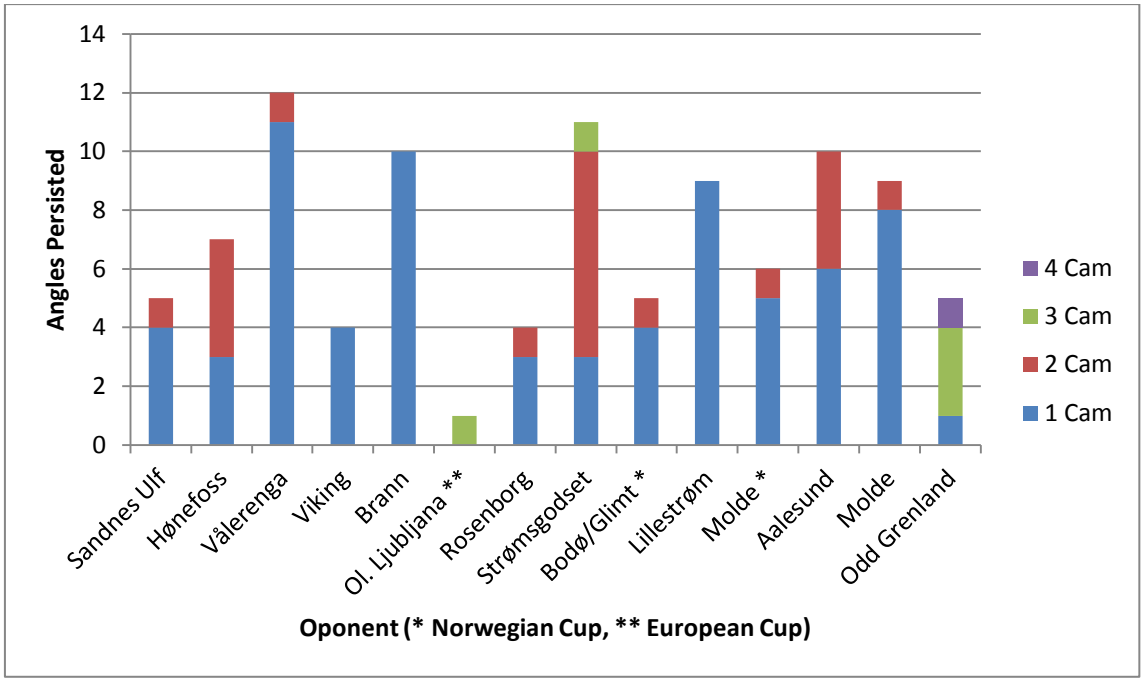


FIGURE 21 - ANGLES PERSISTED PER MATCH

Figure 22 shows the amount of video snippets that have been captured, and the amount of videos that actually have been persisted after Agnar have deleted those that were not interesting. At average this result in just over 9 videos persisted after each match and during a season these results in a season footprint of around 500MB of video content.

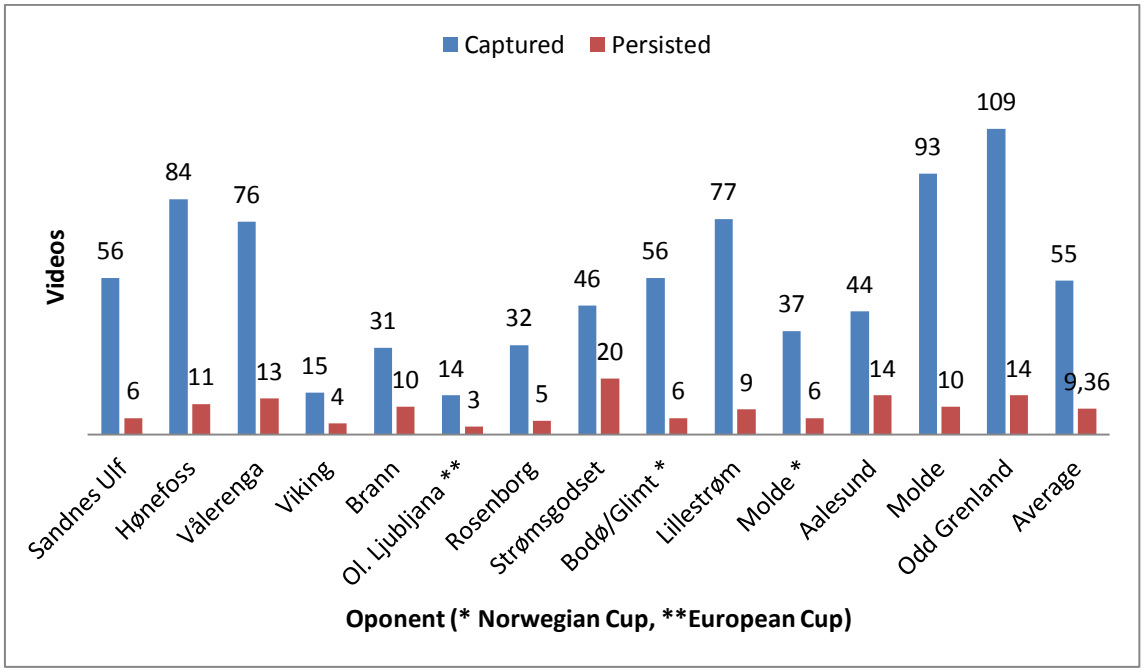


FIGURE 22 - VIDEOS CAPTURED VS. VIDEOS PERSISTED

This case study has provided us with valuable information of how the system is used, what footprint we can expect and how the camera setup has worked out.

5.2 Experiments

We have conducted a series of experiments regarding uploading and downloading of the video files to/from the deployed video storage system.

5.2.1 Experiments Setup

To avoid presenting the specifications multiple times during the description of the different experiments, components and files used are described below. References are made during the description of the specific experiments setup.

Web Server

The web server have an Intel Xeon W3550 3.06GHz with 4 cores, 6GB of physical memory and 2TB NTFS storage device. It is equipped with a Gigabit network card that is connected to a Gigabit switch for outgoing data transfers. As a platform for hosting our web site, we use Windows Server 2008 with IIS 7.0. Videos have a dedicated folder in the web site project, and are delivered by IIS when accesses directly to the video URL is made. The Muithu web application runs on this web server.

Storage

The OpenStack proxy and storage nodes run on Dell blade servers, each having 2 Intel Xeon E 5335 with 4 cores running at 2.00GHz and 8GB of main memory. All servers are equipped with Gigabit network cards that are connected to the same gigabit switch. The proxy server is connected to both a private switch and a public switch.

Client Computers

1) The first client side is a desktop computer with an Intel Core2 Quad CPU Q6600 2.40GHz with 4GB physical memory. This computer runs Windows 7 operating system is equipped with a Gigabit network card.
2) The second client computer that have been used is an Intel Xeon CPU E5335 2.00GHz with four cores, 16GB physical memory and Gigabit network card. This computer runs Ubuntu Server 12.10.

Custom Video Server

A custom HTTP video server was implemented in C# to serve video files on request was implemented running on the Web Server described above. The first experiment is executed where server simply read the file requested from disk and progressively uploaded it to the client in chunks of 300’000 bytes. HTTP range request (RFC 2616) support has also been implemented.

Video Files

We have used videos of three different sizes in our experiments.

1) The first video file is of size 3.87MB and length 15 seconds, which is about the size of an average video file that is served in the Muithu system. This video file is encoded to the encoding scheme integrated with the Muithu web application.
2) The other file is of about size 308MB and length 20 minutes 17 seconds. The latter video file is larger than the expected size of what we are to deliver, but as we do not have any limit on the size of video files uploaded, we need to assume that such a scenario is plausible. This video file is also encoded to the encoding scheme integrated with the Muithu web application.
3) The last file is a 15 second camera-encoded video file of size 8.87MB that is directly extracted from a larger video of an exercise that is captured by GoPro HD Hero2 camera.

All communications and file transfers are done through Gigabit links with a round-trip-time that is less than 1ms.

5.2.2 Test Plan

With the experiments, we want to conduct a series of measurements to evaluate and quantify the properties of the system. We want to make some comparisons between the previous video delivery system (web server) and the new video storage solution.

There will be focus on the end-to-end performance from a video file is uploaded, the encoding process, and to the video request from a client is issued. The throughput performance of the web and storage server will be measured. We also want to learn more about the impact of the moov atom locality, and will design some experiments to observe how the moov atom locality impacts the latency performance.

5.2.3 Upload

5.2.3.1 Uploading Pipeline

Setup

In this experiment, we want to measure the amount of time it takes for a video to be made available for download through the Muithu web interface. A video file will be posted in the uploading portal in the web interface; each step of the pipeline described in Figure 10 on page 26 will be measured using C# server side code. We use video file 3 (from experiment setup) as input to the system. The experiment has been repeated ten times, and the average measurements are presented.

Uploading

We observe from our measurements in Figure 23 that it takes about 7 seconds to store a 15 second recorded video. The major part of this time (almost 95%) is used encoding the video. Uploading the file to the video backend storage takes about 141.9ms, and is the time it takes to ensure that we have the video persisted in three replicas on the storage cluster. From previous measurements made in [6], we have reduced the thumbnail process time from 718.7ms to 137ms. This is probably due to the new encoding scheme where the moov atom is located at the beginning of the video file instead at the end, as the timescale and duration is

available at the beginning of the video file. We have also updated the FFmpeg executable to a newer version, which also can have impact on the thumbnail processing time.

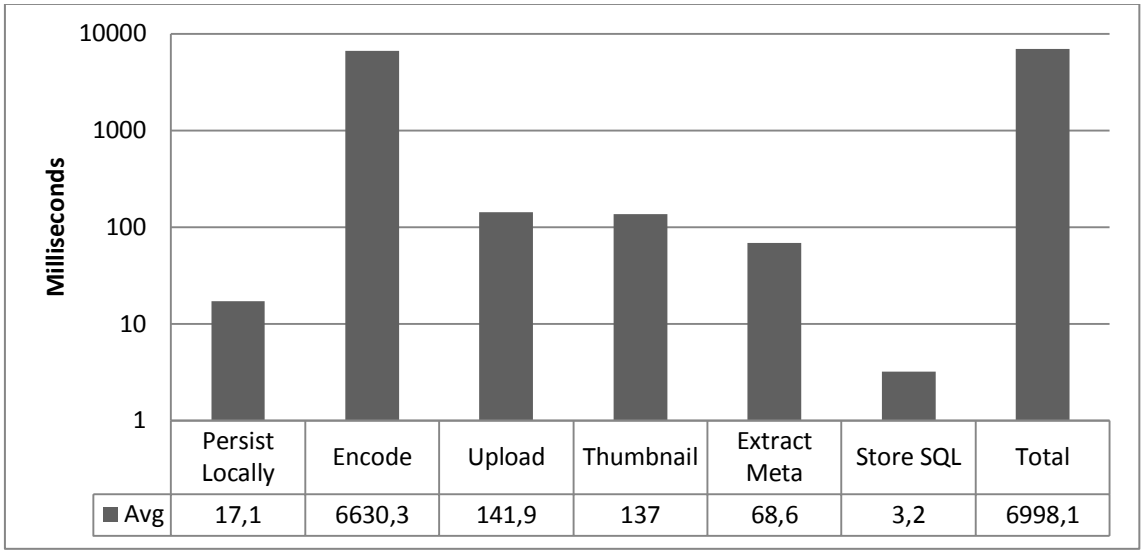


FIGURE 23 - STORAGE PIPELINE MEASUREMENT (LOGARITMIC SCALE)

5.2.3.2 Encoding

Setup

In these two experiments, we want to measure and evaluate some different compression schemes available when using FFmpeg: ultrafast (low), no specified (default) and veryslow (high). We will inspect how the different schemes impact the time used encoding a video, and how much we can reduce the footprint by increasing the compression. Each encoding scheme will be run 20 times to get a representative amount of samples for timing measurement. Error bars will be represented in a 95% confidence interval. These measurements are made on the client computer 1, so the timing measurements will differ from the previous experiment. The experiment is executed using a PowerShell script that runs FFmpeg with the different configurations, using the "Measure-Command" PowerShell command to measure the execute time of the executable. The camera encoded video file 3 is used as input to the encoder in these experiments.

We also want to inspect the CPU utilization of FFmpeg to evaluate the impact on other software running on the same computer. This will be done by using the Windows Performance Monitor for logging the CPU utilization.

Compression

Figure 24 illustrates our observations made from experimenting with the three different compression schemes. Video file 3 is the baseline in our experiment, and the encoding ratios are listed relative to this. We observe that encoding the video with the fastest encoding scheme results in a 45 percent footprint relative to the original recording, and takes 6.25seconds to encode. By using the default compression scheme, we can utterly reduce the footprint by 1 percent. This takes about twice as long as the lowest encoding. With the highest

compression scheme, footprint is reduced by less than 1 more percent, but takes more than ten times longer time than the low encoding.

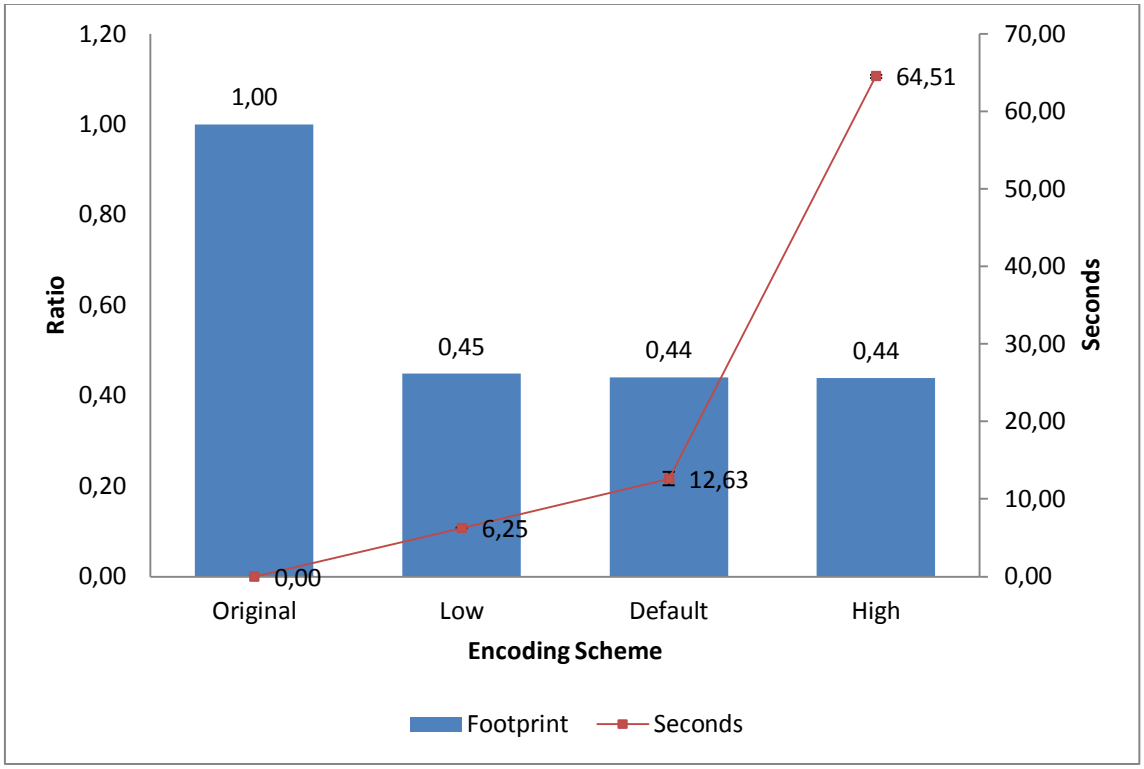


FIGURE 24 - RELATIVE COMPRESSION RATE AND TIME SPENT ENCODING

We have chosen to use the default compression scheme for our encoding. Even though one percent does not seem much, in our experiment, it results in a footprint decrease of 75.4KB. Considering the amount of videos we potentially end up archiving in the future, we conclude that we can spare the extra seconds of CPU time for encoding each video snippet. From the evaluation of the uploading pipeline in Figure 23, we also observe that the web server uses about half the time of the measurements used in this experiments. This is due to its two extra CPU cores.

CPU Utilization

Figure 24 plots the CPU utilization of FFmpeg when encoding a 15 second video sequence with our chosen encoding scheme. We observe that it utilizes close to 100% the multicore CPU throughout the whole encoding process. FFmpeg have configurations allowing us to restrict the amount of threads it is allowed to use, ensuring that it does not grab all the resources. This can be preferable when the encoder is cohosted with other services, such as our case with the web application.

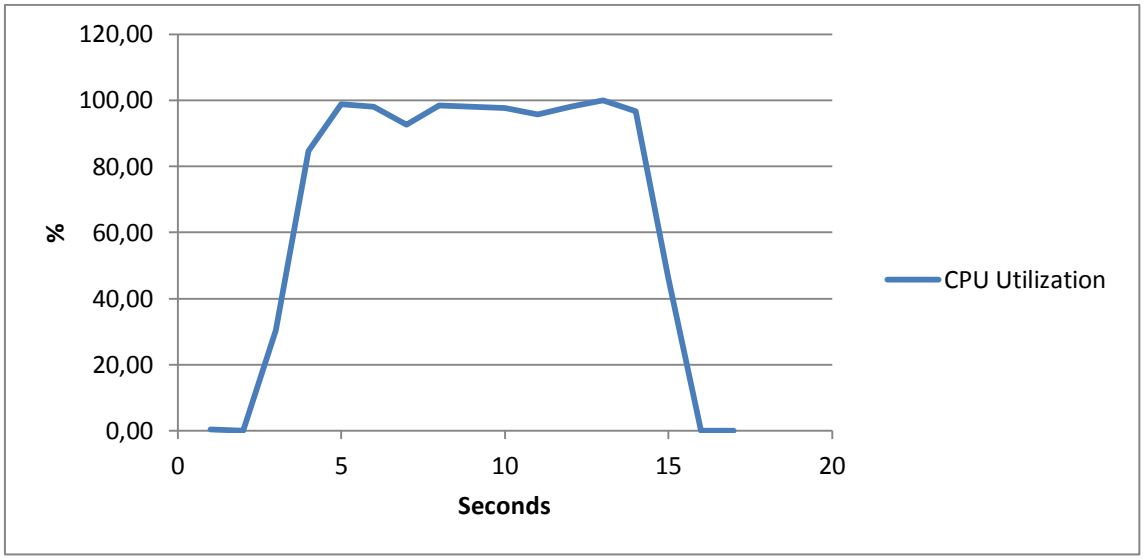


FIGURE 25 - FFMPEG CPU UTILIZATION

With a close to 100% CPU utilization, the encoder is a good example of a service that could be run as a cloud service in a virtualized cloud environment with maximum payoff for the money spent on it [4]. The encoding service could also be scaled out horizontally to support multiple concurrent video encodings.

This also explains why the web server with four cores spends half the time encoding the video compared to the client computer with two cores.

5.2.4 Download

5.2.4.1 Latency

Setup

Latency of the video loading is important for the user experience. The load to play latency will in all cases be measured client-side using JavaScript code where the start timer is triggered on the event when the video player starts requesting the video, and the stop timer is triggered on the event where the video player actually start playing the video. The HTML5 video player is set to automatically start playing the video.

The client computer 1 is used in these experiments using Google Chrome 21.0.1180.89m web browser for measuring the latencies.

For this experiment, we will repeat the experiment ten times to calculate a representative mean. Error bars plotted is within a 95% confidence interval.

Latency Measurements

We can observe from our experiment results in Figure 26 that our OpenStack deployment delivers the first chunk of data to the video player within the first 35.6 milliseconds, while our old solution delivers the first chunk within 38.5 milliseconds.

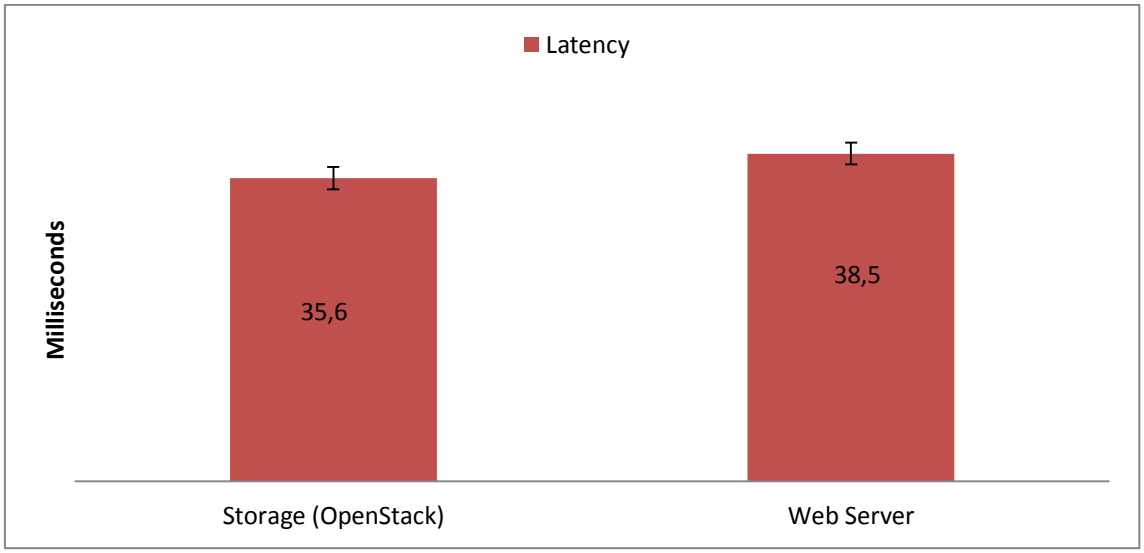


FIGURE 26 - LOAD-TO-PLAY LATENCY IN DEPLOYED SOLUTIONS

We observe that OpenStack has lower latency, despite the communication overhead between our proxy and storage nodes. A logical explanation to this would be that the OpenStack storage solution is optimized to work with unstructured binary files, and is configured to work with a smaller delivery chunk size than the web server. This explanation has not been confirmed.

The load-to-play latency is in both cases acceptable, but will increase as the total load on the system increase.

5.2.4.2 Moov Atom Locality

Setup

In this experiment, we want to observe the impact of the moov atom locality both regarding latency, but also in the context of client footprint. We will execute four measurements; two with the small video file (video file 1) and two with the large video file (video file 2), each with the moov atom located at the beginning and the end of the video file.

We also want to inspect how the HTTP request pattern is when requesting videos. The request inspection will be done using Fiddler[12] web debugging tool, which is a HTTP debugging proxy that every HTTP requests passes through.

Video files are delivered from the custom video server to have more insight and control on the request process.

The client computer 1 is used in these experiments using Google Chrome 21.0.1180.89m web browser for measuring the latencies.

[12] http://www.fiddler2.com/

For this experiment, we will repeat the experiment ten times to calculate a representative mean. Error bars plotted is within a 95% confidence interval.

Latency Impact

We observe from Figure 27 that the load latency impact of the moov atom locality delivered from a server without caching. When the moov atom is located before the mdat, the load-to-play latency is at average 37.7ms for the small video files and 79.3ms for the large. When the moov atom is located at the end of the video file, we observe that the load to play latency is 59.5ms for the smaller file, while it is 786.1ms for the larger. By relocationg the moov atom, the load to play latency improves by about 37% for our small video file and by 90% for our large file.

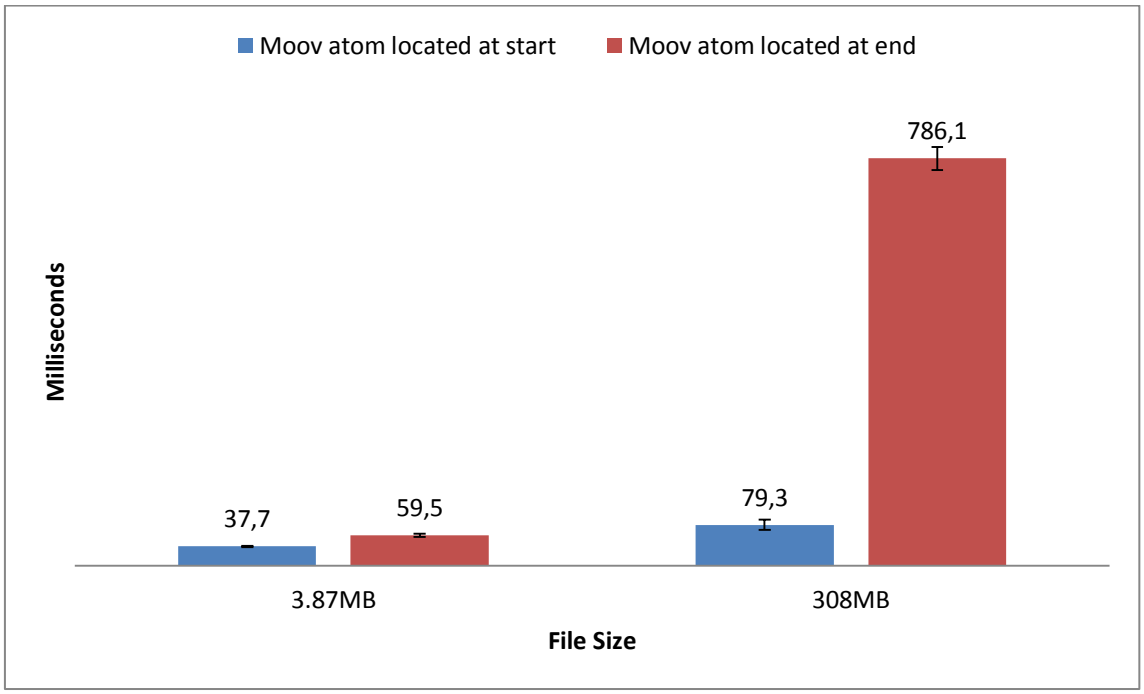


FIGURE 27 – LOAD-TO-PLAY LATENCY (NO CACHING)

HTTP Request Pattern

We early on observed that the HTML5 video player sent multiple HTTP GET requests to our custom video server when playing a video encoded using FFMPEG (mpeg-4 format). Inspecting the HTTP headers with Fiddler, the request pattern was the following for a video file of 3.78MB size:

1) Request made for whole video file
2) Range request made for ~last 8k bytes
3) Range request made for whole file except a small portion in the beginning

By combining this information with information from logging on the server, it seemed that the first request was cancelled after just sending 300k-600k bytes of data. This indicates that the HTML5 player expects the moov atom to be located at the beginning of the file, but cancels the request when noticing that it is not. It then range requests the moov atom from the end of

the video file, before requesting the actual video data to enable playback. We have inspected and measured the latency from requesting a video and to it plays off in the HTML5 video player. Encoding video with FFmpeg, by default, result in a file where the moov atom is located at the end of the file.

What we observe from our experiments is that locating the moov atom at the end of the video file is far from optimal in the context of serving video files to a HTML5 web site using progressive downloading to deliver data. By having the moov atom located prior to the mdat reduces the number of requests, implicitly reducing the number of disk accesses to be made on the server. These observations indicate that for small video files, the moov atom location have less impact on the performance than when working with large video files. By relocating the moov atom, we also reduce the total amount of HTTP requests made from the client, as well as the amount of data sent through the wire. The request count is reduced from 3 to 1, while the data sent onto the link is reduced to up to 50% if the video player doesn't manage to cancel the request before the whole file is sent from the server. As FFmpeg encoding alone locate the moov atom at the end of the file by default, we learned from this experiment that another step to relocate the moov atom is needed in the encoding process to avoid unnecessary requests and data transfer.

5.2.4.3 Throughput

Setup

We want to measure the throughput of our deployed OpenStack Object Storage solution to inspect how close to theoretical maximum throughput it can perform. Comparison with the old video storage solution will be made. Analysis of maximum throughput, how many concurrent video files that can be served, and alternatives for scaling will be discussed if sufficient throughput cannot be provided.

When investigating the throughput performances, we will slowly increase the number of requestors with one per 10 second, allowing us to get 10 samples of each added node, calculating the error margin within a 95% confidence interval. The throughput experiment will show up to 25 concurrent video requests, while bandwidth per requestor analysis will show up to 500 concurrent requests.

Requests are generated by client computer 2 using a python script to spawn threads that simulate concurrent users. Each thread accesses the video file 1 in a repetitive manner. We have validated that close to all of the concurrent threads TCP connections are established during the build-up.

On our web server, we will use Windows Performance Monitor to measure and log the bandwidth usage.

For measurements on our storage solution, we will use a simple Linux terminal shell script which poll for network information and pipes the results out to a log file. This shell script will be run using the Linux *watch*[13] command to repeatedly run it every second.

Throughput

Figure 28 illustrates our throughput measurements with a standard error within a 95% confidence interval on both the web server and the storage solution. The red dotted line illustrated the theoretical maximum throughput on Gigabit Ethernet. By slowly increasing the number of concurrent requestors, we can observe how the two different solutions provision their capacity, and Windows Server steadily serves data at a higher rate than OpenStack, but they peak at close to the same throughput: 95.7% of theoretical max throughput for the web server and 94.9% for OpenStack Object Storage.

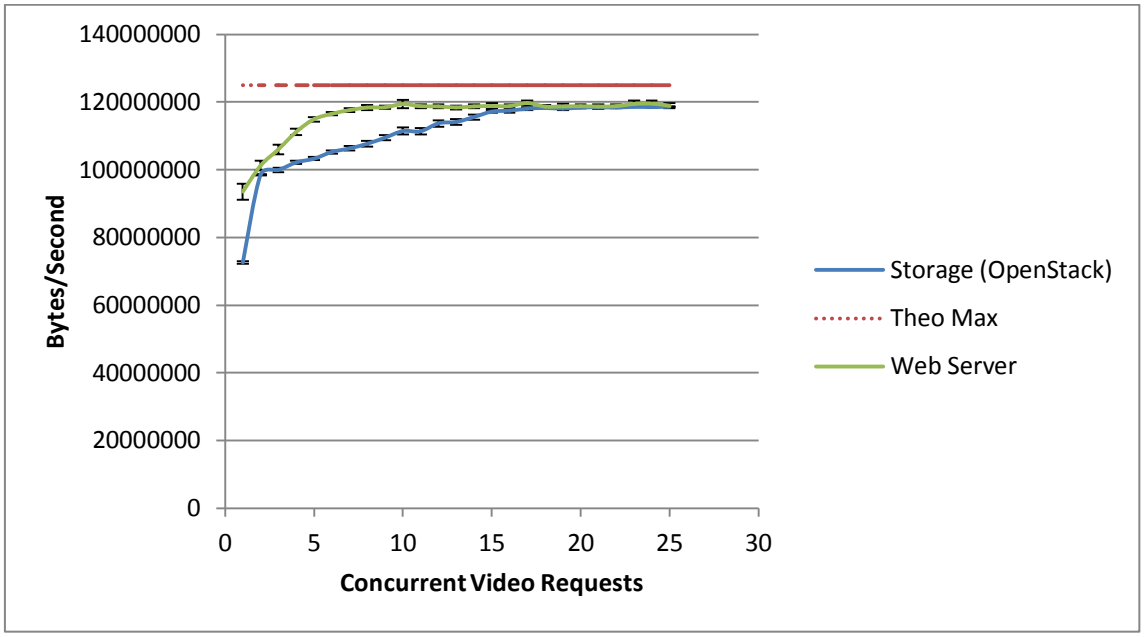


FIGURE 28 - NETWORK THROUGHPUT

Videos are delivered in a progressive manner, and with our current encoding scheme, this enables the video to start playback already after buffering a small amount of the video data. A concrete example is illustrated in Figure 29 where each requestor has the same network priority: If smooth playback is equivalent with that the whole video need to be downloaded within the playback time (15 seconds), about 450 concurrent requests are supported for a video file of size 3867918 bytes size. Our calculation of the mean value with 450 concurrent requests had a standard deviance of 136340 bytes/second, indicating that assuming to support this amount of concurrent requests would be naïve and a gamble. To be safe, we recommend that no more than 300-350 concurrent requests of this video size should be handled by a single proxy node with the current setup. We have experimented with browsing the videos in the web interface when stressing the video storage solution: Notable latency and jitter were noticed when closing in on around 400 concurrent requests in the stress-test.

[13] http://linux.about.com/library/cmd/blcmdl1_watch.htm

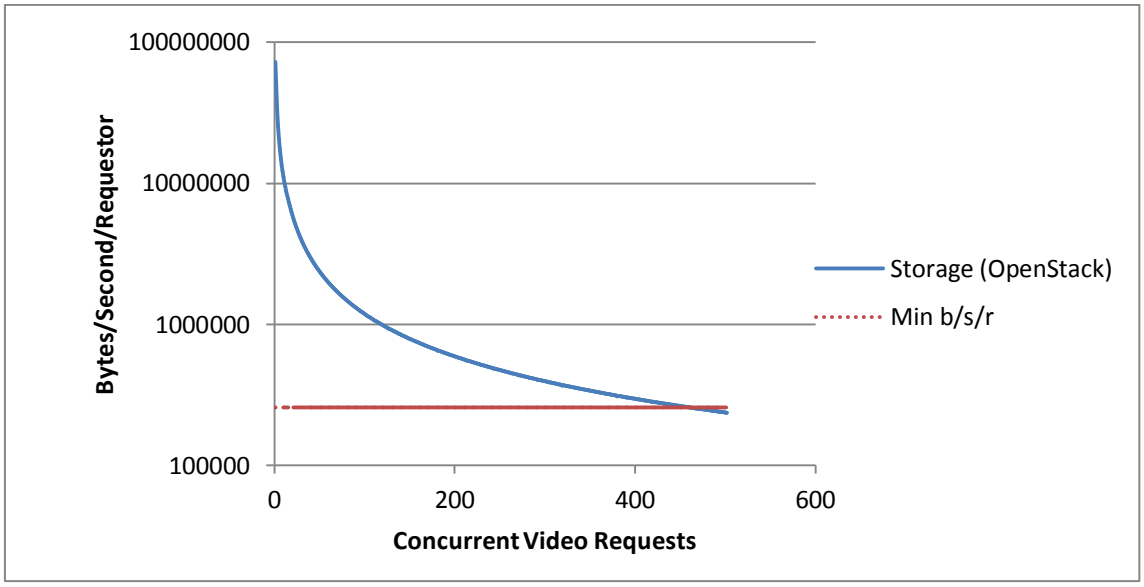


FIGURE 29 – MINIMUM BANDWIDTH PER REQUESTOR WITH PROGRESSIVE DOWNLOADING (LOGARITMIC SCALE)

If videos were encoded with a scheme not supporting progressive downloading, the whole video would need to download before the playback can start at all. This would change the number of concurrent users supported. Studies show the average shopper expects a web site to load within two seconds [41]. If this is number also is applicable to video surfers, a video not supporting progressive downloading would need to download within two seconds. This example is illustrated in Figure 30, where the average requestor throughput per second would need to match half the size of the video. Using this encoding scheme would lead to support for around 60 concurrent users, but with a playback latency of up to two seconds.

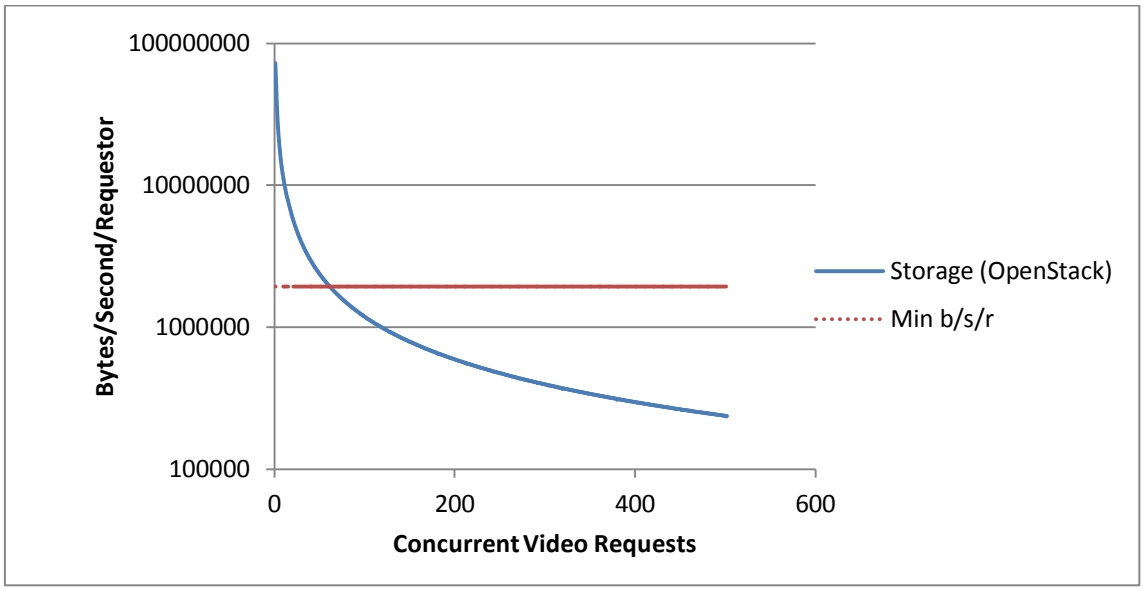


FIGURE 30 - MINIMUM BANDWIDTH PER REQUESTOR WITHOUT PROGRESSIVE DOWNLOADING (LOGARITMIC SCALE)

There are multiple options available when the system register that the theoretical maximum number of requests are about to be reached.

Horizontal scaling of private storage cloud

An extra proxy node can be added to support more concurrent requests. With this approach, the underlying network infrastructure and storage nodes would also need to support the addition of potential concurrent requestors for the added proxy.

Public cloud vendors

We could utilize a public cloud storage service for storing and delivering all of our content. With this approach, we would need to upload all of our video-content to a public cloud provider. The amount of data uploaded could be in the range of Giga- or Terabytes, while the amount of interesting videos at each peak might be only a subset and in the range of a couple of hundred megabytes (20-100 interesting videos in our context). With this approach, we would end up paying a public cloud provider a large amount of money to archive all of our videos. This could also conflict with privacy policies set for the video data.

Table 2 contains an extended version (with this year's pricing) of the changes in price of Amazon Web Services Simple Storage Service (AWS S3) from [4]. We observe that the pricing have decreased drastically the last four years, and is less than 63% of the original price introduced in 2006. It can be economically wise for elasticity purposes to use a public cloud vendor to handle sudden extreme peaks in traffic to avoid potential loss of customers, but archiving content in the cloud can be expensive compared to buying new hardware. The price of storing 1TB of content in AWS S3 costs 95$ each month and 1140$ for storing it for a year, while several new 1TB storage drives can be bought for the same amount of money. A more economic-efficient way of storing and serving data would be to have less content stored in the cloud that is accessed a lot. This could i.e. be the top accessed content that a private cloud solution cannot handle serving due to its limit in throughput. Privacy policies could also restrict a storage vendor to migrate data into a public cloud solution.

TABLE 2 - CHANGES IN PRICE OF AWS S3 STORAGE AND NETWORKING OVER TIME

Storage	**Cost of Data Stored per GB-Month**						
Date	< 1 TB	1-50TB	50-100 TB	100-500 TB	500-1000 TB	1000-5000 TB	5000-10000 TB
3/13/06	$0.15	$0.15	$0.15	$0.15	$0.15	$0.15	$0.15
10/9/08	$0.15	$0.15	$0.14	$0.13	$0.12	$0.12	$0.12
12/1/12	$0.095	$0.080	$0.070	$0.070	$0.065	$0.060	$0.055
% Original Price	63%	53%	47%	47%	43%	40%	37%
Networking	**Cost per GB of Wide-Area Networking Traffic**						
Date	In	Out: < 1GB	Out: 1GB - 10 TB	Out: 10-50 TB	Out: 50-150 TB	Out: 150–500 TB	Out: > 500 TB
3/13/06	$0.20	$0.20	$0.20	$0.20	$0.20	$0.20	$0.20
10/31/07	$0.10	$0.18	$0.18	$0.16	$0.13	$0.13	$0.13
5/1/08	$0.10	$0.17	$0.17	$0.13	$0.11	$0.10	$0.10
12/1/12	$0.00	$0.00	$0.12	$0.09	$0.07	$0.05	*Special*[14]
% Original Price	∞	∞	60%	45%	35%	25%	Unknown

Federated solutions

To solve these problems, we could in example use a system as Balava [11] to seamlessly ship the most accessed public data into a public cloud solution to offload the private cloud infrastructure for the period of time needed, then revoke the data when its popularity decreases. Uploading data to the AWS S3 storage is for free. The load balancer described in our architecture could redirect users to the public cloud instead of the private cloud when video accesses for these specific files are made. This would offload our private cloud solution in an economic fashion, while maintaining the privacy policies. Interoperating with public cloud vendors should be relative simple due to our focus on using REST API for accessing video content.

5.3 Fault Tolerance

The deployed OpenStack Object Storage solutions provide greater fault tolerance than the previous single server solution. Our backend solution now handles individual failure of multiple single storage nodes, by both keeping data available and easy to recover via replication and fail-over mechanisms.

As all nodes are currently configured and connected to the same switch and located on the same rack, making switch- or rack-failure a weak spot for the current deployment. To solve this problem, nodes would need to be dispersed across multiple racks and switches, utilizing the availability zones in OpenStack Object Storage to ensure that data is localized on nodes not residing on the same rack or switch.

The current deployment is configured to replicate data onto three nodes, to ensure recovery and availability of up to two failing nodes. Adding nodes to the cluster will lead to automatic

[14] Amazon negotiate bandwidths higher than 500TB/month

re-assigning of primary and fail-over nodes, which essentially mean that data is replicating back and forth over the private network.

5.4 Security and Privacy

Security and privacy is ensured through the front-end web site. Temporary URLs are generated and signed with a secret only known to the application and the proxy servers. This temporary URL is given to user that the web front-end authorizes to access the requested video. Proxy servers are simply validating the temporary URL given, authorizing the user, giving him or her access to the video. This authorization mechanism is the same mechanism used in Rackspace Cloud Files[15].

By providing sufficient access control to data, we ensure data is kept private to those that are allowed to download it. Temporary URLs are also only authorized for a limited amount of time, which shuts down access to URLs that accidently have been revealed to the public.

[15] http://www.rackspace.com/blog/rackspace-cloud-files-how-to-create-temporary-urls/

6. Conclusions

This chapter will sum up our achievements, the future work and a conclusion of this thesis. We have designed, implemented and evaluated a distributed storage solution serving video content to a sports analysis web site.

6.1 Achievements

The thesis states its problem definition, interpretation, scope and limitations in chapter 1.1 and chapter 1.2. The problem definition states the following:

> *"This thesis shall design, deploy and evaluate a video storage management system for efficiently storing, managing, securing and disseminate video content on a heterogeneous intra-cloud platform. Focus will be to provide horizontal scalability within a private cluster and fault tolerance in a system that efficiently serves video files to a sport analytics web interface in a secure manner. Video format and encoding must be evaluated to ensure that playback quality is satisfying."*

This thesis has focused on deploying a video storage back-end serving video content to a sport analysis web site. We have evaluated a few popular open source distributed storage solutions based on requirements and a desired architecture presented in chapter 3, and ended up with deploying OpenStack Object Storage. This choice was based on its consistency model, its focus on unstructured binary content, its integrated authentication and authorization, and because it is a part of a full cloud stack integration for potential future integration. We describe the deployment and configuration process, as well as integration with the web front-end in chapter 4. We have also discussed and designed appropriate encoding schemes to ensure that all video files served to the web front end is encoded with the same scheme. Users are authorized to access videos for only a short period of time for privacy purposes.

To evaluate the deployed video storage, we have also integrated the video storage management system with the existing Muithu web application. Integration regarding access, uploading and deletion has been added to prove that it can be used in the desired context of serving video files in an efficient manner to a sport analytics web interface. Our case study has been done using both the old and the new video storage system to upload, access and delete videos.

The experiments executed analyze different parts of the uploading and downloading videos to/from the video storage system to quantify the performance of our storage and its integration with the web application.

The video storage and its integration with Muithu are in production and are actively used internally by athletes and coaches of the Norwegian soccer club Tromsø IL.

6.2 Future Work

We have now deployed our private storage cloud, and persisted videos in that cloud in a fault tolerant, available and scalable manner. Our solution is currently only using a single proxy, which is feasible due to the fact that we only have a single Gigabit link out of our cluster. In the

future, it would be natural to complete the deployment according to the architecture outlined in Figure 9 (page 24) with multiple proxies and a load balancer to redirect requests. Hardware should also have been specialized for its purpose, such that storage nodes are equipped with more storage capacity than the current deployment.

As the entire OpenStack Object Storage is open source, possibilities of implementing some kind of memory caching on the proxy nodes could been investigated. Alternative security mechanism using i.e. Code Capabilities [9] for authorizing client requests on the proxy nodes could also be further investigated.

At the moment, the deployed solution provides the ability to horizontally scale out by adding more nodes to the storage cluster. We have considered and discussed cloud federation using Balava [11] to provide elasticity. This concept should be included and evaluated further in future versions to provide elasticity with focus on privacy and economics. By utilizing the cloud to offload network traffic on certain data can be proven to become an economic and efficient way of utilizing the best of both private and public cloud computing, while still maintaining a strict privacy on content that demands it.

Optimally, a compute cloud running for example OpenStack could be deployed on the remaining nodes in the cluster. This way, we could have a virtual environment to run services and components, such as the encoder, decoupled from the web front-end. In such an environment, we could also host a fault tolerant storage solution to host the structured metadata, which currently reside on the web front end coupled with the Muithu web application. This can either be relocated in a highly consistent SQL store (minimal changes to metadata relations) or in a scalable NoSQL store such as Cassandra (lots of changes to metadata structure).

The web application should also run in a virtualized environment to make it more faults tolerant and scalable. Currently, the single web server deployment is a single point of failure that will bring down the whole Muithu web interface. As the whole web application is developed using the Microsoft Stack (.NET), it would be preferably to have a private virtualized environment having support for the Microsoft Hypervisor (Hyper-V). Another solution would be to deploy the web application in a public cloud, such as Azure.

6.3 Concluding Remark

We have successfully deployed a distributed storage solution with integrated fault tolerance and horizontal scaling. We have also configured and integrated it as the back-end video storage serving video content of a sport analytic web interface. Considerations regarding functional and non-functional properties have been taken during this thesis, and we have deployed a system according to a desired set of properties and architecture. Video encoding and system performance have both been evaluated by expert end users as well as quantified in relevant experiments.

The system is actively being used by members of the Tromsø IL soccer club, which also have been involved in iterative feedback process.

Bibliography

[1] Cisco, "Cisco Visual Networking Index: Forecast and Methodology, 2011-2016," 30 May 2012. [Online]. Available: http://www.cisco.com/en/US/solutions/collateral/ns341/ns525/ns537/ns705/ns827/white_paper_c11-481360.pdf.

[2] Pew Research Center, "The State of Online Video," 3 June 2010. [Online]. Available: http://pewinternet.org/~/media//Files/Reports/2010/PIP-The-State-of-Online-Video.pdf.

[3] Pew Research Center, "PewInternet," 13 September 2012. [Online]. Available: http://pewinternet.org/~/media//Files/Reports/2012/PIP_OnlineLifeinPictures_PDF.pdf .

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," 2009.

[5] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner and P. R. Young, "Computing as a dicipline," *Communications of the ACM,* vol. 32, no. 1, pp. 9-23, 1989.

[6] D. Johansen, M. Stenhaug, R. B. A. Hansen, A. Christensen and P.-M. Høgmo, "Muithu: Smaller footprint, potentially larger imprint," in *2012 Seventh International Conference on Digital Information Management (ICDIM)*, Macau, China, 2012.

[7] P. Halvorsen, S. Særgov, A. Mortensen, D. K. C. Kristensen, A. Eichhorn, M. Stenhaug, S. Dahl, H. K. Stensland, V. R. Gaddam, C. Griwodz and D. Johansen, "Bagadus: An Integrated System for Arena Sports Analytics – A Soccer Case Study.," in *Proceedings of the 4th ACM International Conference on Multimedia Systems (MMSys). To appear.*, Oslo, Norway, 2013.

[8] Å. Kvalnes, D. Johansen, R. v. Renesse, S. V. Valvåg and F. B. Schneider, "Design Principles for Isolation Kernels," 2011.

[9] R. v. Renesse, H. Johansen, N. Naigaonkar and D. Johansen, "Secure Abstraction with Code Capabilities," in *Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Belfast, Northern Ireland, 2013.

[10] D. Johansen and J. Hurley, "Overlay Cloud Networking Through Meta-Code," in *35th IEEE Annual Computer Software and Applications Conference Workshops*, 2011.

[11] A. Nordal, Å. Kvalnes, J. Hurley and D. Johansen, "Balava: Federating Private and Public

Clouds," in *IEEE 2011 World Congress on Services*, Washington, 2011.

[12] S. V. Valvåg and D. Johansen, "Cogset: A Unified Engine for Reliable Storage and Parallel Processing," in *NPC '09. Sixth IFIP International Conference on Network and Parallel Computing*, 2009.

[13] S. V. Valvåg and D. Johansen, "Oivos: Simple and Efficient Distributed Data Processing," in *HPCC '08. 10th IEEE International Conference on High Performance Computing and Communications*, 2008.

[14] D. Johansen, P. Halvorsen, H. Johansen, R. Haakon, C. Gurrin, B. Olstad, C. Grwidz, Å. Kvalnes, J. Hurley and T. Kuoka, "Search-based composition, streaming and playback of video archive content," Springer, 2011.

[15] M. T. Özsu and P. Valduriez, Principles of Distributed Database Systems 3rd Edition, Springer Science+Business Media, 2011.

[16] J. Pokorny, "NoSQL databases: a step to database scalability in web environment," in *Proceedings of the 13th International Conference on Information Integration and Web-based Applications and Services*, Bali, Indonesia, 2011.

[17] E. Brewer, *Towards Robust Distributed Systems. (Invited Talk),* Portland, Oregon: Principles of Distributed Computing, 2000.

[18] Microsoft, "Federations: Building Scalable, Elastic, and Multi-tenant Database Solutions with Windows Azure SQL Database (en-US)," Microsoft, 16 February 2011. [Online]. Available: http://social.technet.microsoft.com/wiki/contents/articles/2281.federations-building-scalable-elastic-and-multi-tenant-database-solutions-with-windows-azure-sql-database-en-us.aspx.

[19] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," in *SOSP '07 Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, Stevenson, WA, 2007.

[20] D. Karger, E. Lehman, T. Leighton, M. Levine, D. Lewin and R. Panigrahy, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *ACM Symposium on Theory of Computing*, New York, NY, 1997.

[21] "Survey of Distributed Databases," [Online]. Available: http://dbpedias.com/wiki/NoSQL:Survey_of_Distributed_Databases.

[22] D. K. Gifford, "Weighted voting for replicated data," in *SOSP '79 Proceedings of the seventh ACM symposium on Operating systems principles*, Pacific Grove, CA, 1979.

[23] 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.

[24] R. C. Merkle, "A Digital Signature Based on a Conventional Encryption Function," in *CRYPTO '87 A Conference on the Theory and Applications of Cryptographic Techniques on Advances in Cryptology*, Santa Barbara, CA, 1988.

[25] G. L. Sanders and S. K. Shin, "Denormalization effects on performance of RDBMS," in *Proceedings of the 34th Annual Hawaii International Conference on System Sciences*, 2001.

[26] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes and R. E. Gruber, "Bigtable: A Distributed Storage System for Structured Data," in *OSDI '06: 7th USENIX Symposium on Operating Systems Design and Implementation*, Seattle, WA, 2006.

[27] R. Sumbaly, J. Kreps, L. Gao, A. Feinberg, C. Soman and S. Shah, "Serving Large-scale Batch Computed Data with Project Voldemort," in *Proceedings of the 10th USENIX Conference on File and Storage Technologies*, 2012.

[28] T. Schütt, F. Schinkte and A. Reinefeld, "Scalaris: Reliable Transactional P2P Key/Value Store," in *ACM SIGPLAN Erlang Workshop*, 2008.

[29] J. Lennon, Beginning CouchDB, New York: Springer-Verlag, 2009.

[30] Cloudant, "BigCouch," Cloudant, [Online]. Available: http://bigcouch.cloudant.com/.

[31] Amazon.com, "Amazon SimpleDB," [Online]. Available: http://aws.amazon.com/simpledb.

[32] 10gen, "MongoDB," 10gen, [Online]. Available: http://www.mongodb.org/.

[33] A. Lakshman and P. Malik, "Cassandra: a decentralized structured storage system," *ACM SIGOPS Operating Systems Review,* vol. 44, no. 2, pp. 35-40, 2010.

[34] "OpenStack," [Online]. Available: http://www.openstack.org/.

[35] J. Elson and J. Howell, "Handling flash crowds from your garage," in *ATC'08 USENIX 2008 Annual Technical Conference on Annual Technical Conference*, Boston, MA, 2008.

[36] YouTube, "Encoding Video For YouTube: Advanced Specifications," YouTube, [Online]. Available: https://support.google.com/youtube/bin/answer.py?hl=en&answer=1722171&topic=1728573&ctx=topic.

[37] M. Levkov, "Understanding the MPEG-4 movie atom," Adobe, 18 October 2010. [Online]. Available: http://www.adobe.com/cn/devnet/video/articles/mp4_movie_atom.html.

[38] Canonical, "Ubuntu Wiki: MAAS," Ubuntu, [Online]. Available: https://wiki.ubuntu.com/ServerTeam/MAAS.

[39] Canonical, "About Juju," Ubuntu, [Online]. Available: https://juju.ubuntu.com/docs/about.html.

[40] R. B. A. Hansen, "A Prototype Information Access System for Soccer Analytics," UiT, 2012.

[41] Gomez, "Why Web Performance Matters: Is Your Site Driving Customers Away?," 2 February 2010. [Online]. Available: http://whitepapers.bx.businessweek.com/whitepaper8022.

[42] I. Khan, "Distributed Caching On The Path To Scalability," Microsoft, [Online]. Available: http://msdn.microsoft.com/en-us/magazine/dd942840.aspx.

[43] B. Calder, J. Wang, A. Ogus, N. Nilakantan, A. Skjolsvold, S. McKelvie, Y. Xu, S. Srivastav, J. Wu, H. Simitci, J. Haridas, C. Uddaraju, H. Khatri, A. Edwards, V. Badekar, S. Mainali, R. Abbasi, A. Agarwal, M. F. u. Haq, M. I. u. Haq, D. Bhardwaj, S. Dayanand, A. Adusumilli, M. McNett, S. Sankaran, K. Manivannan and L. Rigas, "Windows Azure Storage: a highly available cloud storage service with strong consistency," in *SOSP '11 Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, Cascais, 2011.

[44] Amazon, "Amazon Simple Storage Service (Amazon S3)," Amazon, [Online]. Available: http://aws.amazon.com/s3/.