Big Data Archiving with Splunk and Hadoop

EMRE BERGE ERGENEKON
and PETTER ERIKSSON

KTH Computer Science
and Communication

Master of Science Thesis
Stockholm, Sweden 2013

Big Data Archiving with Splunk and Hadoop

EMRE BERGE ERGENEKON
and PETTER ERIKSSON

DD221X, Master's Thesis in Computer Science (30 ECTS credits)
Degree Progr. in Computer Science and Engineering 300 credits
Master Programme in Computer Science 120 credits
Royal Institute of Technology year 2013
Supervisor at CSC was Stefan Arnborg
Examiner was Olle Bälter

TRITA-CSC-E 2013:006
ISRN-KTH/CSC/E--13/006--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication

KTH CSC
SE-100 44 Stockholm, Sweden

URL: www.kth.se/csc

Abstract

Splunk is a software that handles large amounts of data every day. With data constantly growing, there is a need to phase out old data to keep the software from running slow. However, some of Splunk's customers have retention policies that require the data to be stored longer than Splunk can offer.

This thesis investigates how to create a solution for archiving large amounts of data. We present the problems with archiving data, the properties of the data we are archiving and the types of file systems suitable for archiving.

By carefully considering data safety, reliability and using the Apache Hadoop project to support multiple distributed file systems, we create a flexible, reliable and scalable archiving solution.

Referat

Arkivering av stordata med Splunk och Hadoop

Splunk är en mjukvara som hanterar stora mängder data varje dag. Eftersom datavolymen ökar med tiden, finns det ett behov att flytta ut gammalt data från programmet så att det inte blir segt. Men vissa av Spunks kunder har datalagringspolicies som kräver att datat lagras längre än vad Splunk kan erbjuda.

Denna rapport undersöker hur man kan lagra stora mängder data. Vi presenterar problemen som finns med att arkivera data, egenskaperna av datat som ska arkiveras och typer av filsystem som passar för arkivering.

Vi skapar en flexibel, tillförlitlig och skalbar lösning för arkivering genom att noga studera datasäkerhet, tillförlitlighet och genom att använda Apache Hadoop för att stödja flera distribuerade filsystem.

Acknowledgments

We want to thank Boris Chen for being a great mentor, his help and his feedback, without which this project would not have been possible. Thanks to Anton Jonsson and André Eriksson, the successor Swedish interns at Splunk, who helped us with performance testing and the implementation of the user interface. We also want to thank the rest of the Splunk-Hadoop team for their support.

Contents

Chapter 1

Introduction

Splunk is a software that is used for performing searches, visualizations and monitoring on text data. The text data can either be structured, such as XML, or in unknown formats generated by an arbitrary software. By having no requirements on the format, Splunk users has the ability to connect data outputs of various software to monitor them from a single interface. The original format of the data is abstracted from the user and can be analyzed using a single search language. Data that was generated independently can be combined by grouping common properties, such as date and user, to create knowledge that was hard to gain when the data was fragmented across multiple sources.

The company Splunk Inc released their first software in 2006. Since then they have acquired over 3.300 customers, including more than half of the Fortune 100 companies [1].

Many of Splunk's customers handle large amounts of data every day. As the data inside the product grows, the search times increase and there is need for more for storage. In order to manage this, customers can implement various retention policies. Policies could be, for instance, to delete data that is older than some time value. Or alternatively, it could be to archive data to another storage system (attached storage, tape backup, etc.). In addition to a policy based on time, data can be archived or deleted based on total size.

Currently, Splunk does not do any of the archive management itself. It rather provides configuration options to control when data should be removed, as well as configuration for a script that is called just before the removal event. What the script does is up to the customer. Via these configurations, the customer can archive the data in whatever fashion they want. Similarly, if they want to restore, they will need to write a script to bring the data back to a pre-determined location for Splunk to search upon.

These scripts are custom implementations. If there were a solution for this provided to the customers, then it would eliminate the need for customers to develop this on their own. Such a solution would provide a simpler way of defining policies, rather than editing configuration parameters. It would also give the customer a

view of the system's state (what has been archived, what has not, and what has been restored), and some user-friendly ways of managing the data.

The goal with this thesis will be to create a complete archiving solution for Splunk. The solution should make Splunk's customers favor using our archiving solution in production, rather than implement their own.

Chapter 2

Problem statement

How can we design and implement an archiving feature for Splunk that satisfies the following attributes:

* Reliability - Prevent data loss even under adverse situations.
* Scalability - Deal with large data sizes, and large distributed deployments.
* Manageability - Provide state of system and metrics through a user interface, allowing users to make decisions.

Chapter 3

Background

In 1898 the Danish inventor Valdemar Poulsen created the first magnetic recording device [2]. His invention recorded his own voice on a length of piano wire. In 1950 magnetic tapes were introduced as data storage medium with a capacity of 8 megabytes [3]. 2011, about 50 years later, a single tape could store 5 terabytes of data [4]. While tape recording enables for high volume storage it requires custom built tape libraries, and these make it hard to access the data for analysis.

In the past couple of years hard drives have evolved fast and can also offer to store large amount of data. By clustering computers, we are able to create storage entities that can contain petabytes of data. A distributed file system software is used on these clusters to abstract the block storage mechanism of a hard drive, from the client that wants to store data. With this abstraction the client can use the file system as if it was a very large hard drive. Another benefit with this abstraction layer is that features like data replication and security can be realized separately, without the client having to change any behavior.

The main benefit of distributed file systems running on computer clusters is that the data is always "live", meaning that the hard drives are always connected to computational unit and this enables the data to be analyzed [5].

Chapter 4

Analysis

To solve the archiving problem, a file system is needed that goes well with the data to be archived. We will first look at the data we are archiving and then analyze different types of distributed file systems, to see how they can solve our problems. Last we will discuss and argue for our archiving solution.

4.1 The data

The data given from Splunk for archiving comes in different formats and sizes. By default the data is one 10 GB gzipped file and around 10 metadata files with different sizes, all contained within a directory. The metadata can be removed to reduce the size without any data loss, and created again when needed. Besides the original format, it can be exported as one large uncompressed CSV file.

4.2 Distributed file systems

A file system suitable for archiving must be reliable enough that no data is ever lost, or the probability of data loss is extremely low. It has to scale with the amount of data stored and should be actively managed and updated, both now and in the future. How it handles big and small files is also important.

Following sections discuss these issues by comparing HDFS, Hadoop Distributed File System, with Lustre. The former is a block storage system while the later is an object storage system [5][6][7].

4.2.1 Data reliability

When the data gets as large as petabytes, the probability of disk failure and corruption gets high [5]. HDFS replicates its data without need for installing any other software or hardware. Lustre however, needs software or hardware support for replication, for example disks on a RAID setup, which will take care of the data duplication internally [8].

HDFS is a block-oriented system, where a file can be split across multiple blocks, and each block in HDFS has a replication factor that defaults to three.

Block oriented systems need to create metadata for translating blocks to files and vice versa. HDFS has one single centralized metadata server to keep track of the blocks. This metadata server is also a single point of failure. However, HDFS comes with backup and failover strategies in case of metadata server failure.

Lustre is an object oriented storage solution. Object oriented storage lets each file be an object, which is stored on one or multiple object storage devices (OSD). Object oriented storage filesystems also separate metadata and files in a metadata server (MDS). However, storage or block allocation does not have to be done by the MDS [9]. In addition, Lustre supports metadata to be spread across multiple MDSs since version 2.x, which removes the single point of failure [10]. Lustre also has failover features for its MDSs.

4.2.2 Scaling

This section examines how the file systems scale with respect to different data sizes. To make the comparison easier to understand the comparison is made when Lustre is configured with only a single metadata server. In this case Lustre's architecture resembles HDFS's architecture. Clients interacting with the systems, in both cases, get file locations from the metadata server. All the I/O operations are handled directly between the clients and the dedicated data store servers.

The metadata grows with number of files on both file systems. Lustre stores the metadata on local disk, while HDFS keeps all the metadata in memory. Local storage has the benefit of storing more data than a memory, but the memory is much faster. The metadata for a file, directory and block in HDFS is about 150bytes [5]. So with 50 million files, each taking one block, you would need at least 15 GB of memory. While it is possible to store millions of files on HDFS, it is not possible to store billions of files with the space limitations of current memory hardware.

When storing large files however, HDFS is more likely to run out of disk space before running out of memory. With an average file size of 1 GB, HDFS can store 200 petabytes with 64 GB memory.

Luster's ability to spread the meta data to multiple metadata servers makes it scale well with growing data. It can store up to 4 billion files in one file system [11]. HDFS's capacity of millions of files should however be enough for most use cases.

Both filesystems handle reading and writing large files similarly. They can both split files into multiple parts, so that the individual parts can be transferred and read from multiple data store servers in parallel. And since both reading and writing is done in parallel, adding more data store servers is cheap.

4.2.3 Conclusion

Lustre scales better when storing hundreds of millions of small files, and both would be equally good choices for larger files. HDFS has data replication and Lustre has

not. But Lustre can achieve the same level of reliability by using a third party software or hardware solution. Neither file systems has obvious benefits over the other, when taking our data in to consideration.

HDFS is a part of a platform with additional features. These features are discussed in section 4.4 and have to be considered as well.

Another alternative would be to use a storage service provider, which has the benefits of not having to maintain the file system. Section 4.3 discusses this.

4.3 Storage as a service

Storing data can also be done in the cloud, where data reliability and scaling is handled by the service provider. Amazon S3 is such a service. It is designed to be highly scalable, reliable, fast and inexpensive. It uses the same infrastructure as Amazon's internal web sites and, according to them, it can handle both small and large files. Payment is made for monthly storage capacity and data outbound. Moving data in S3 or Amazon's other services is free within the same region [12].

If there is not enough resources for maintaining an own cluster and if one can trust that the service never lose any data, then storage as a service can certainly be a good alternative for an archiving file system.

4.4 Hadoop as a platform

HDFS, Hadoop Distributed File System, is part of the Hadoop project, which is open source. The Hadoop project contains more than just a distributed file system. It has a MapReduce framework that, if desired, could enable the archive to be analyzed effectively, due to map reduces' distributed nature. This framework is one of the reason why Hadoop is popular and widely used by companies such as Adobe, Yahoo! and Facebook [13].

The file system communication for all Hadoop modules goes through a file system interface. Integrating other file systems other than HDFS is as easy as creating a new implementation with this interface. Amazon S3, for example, has an implementation with the FileSystem interface. Hadoop also has an implementation for using the local file system. Lustre can be mounted as a local file system and can therefore be used with Hadoop.

Distributed file systems that are gaining popularity also make sure that they are integrated with Hadoop. File systems such as XtreemFS and GlusterFS can be configured to be used with Hadoop [14][15].

4.5 Choice and motivation

We want our archiving solution to be flexible, since data and resources differ from customer to customer. This is why we think having an implementation that supports Hadoop's FileSystem is a good choice. The underlying archiving file system is just

a question of configuration. Different clusters can have different file systems, and still use our archiving solution. We will also be able to take advantage of Hadoop's MapReduce framework. It can be useful when data is exported to CSV. Another reason to go with Hadoop is that many companies already use it [13].

Chapter 5

Hadoop and HDFS

HDFS is a distributed file system which fits well with our problem. This chapter will in depth look at how HDFS can help us solve the problems described in chapter 2.

5.1 Design

This section is about the core design concepts of HDFS. How they work together, what the design is good for and what it is bad at.

5.1.1 Blocks

Like normal disks, HDFS contains blocks. These blocks are much larger than normal disk blocks, and a smaller file than a single block does not occupy a full block's worth of storage [5]. The concept of blocks are important for replicating data, providing fault tolerance and availability. Each block is replicated to a small number of physically separate machines. If a block becomes unavailable for any reason, it can be reproduced from any of the replications on other machines. Blocks containing files with high popularity can be replicated more than this factor, to help with load balancing the cluster's I/O operations.

Large files that do not fit in a single block are not guaranteed to be stored on the same machine. In fact, it is possible that one file is stored in every block, on all machines in the whole HDFS cluster.

5.1.2 Name nodes and data nodes

Name nodes and data nodes are the components that manage the file system's communication and data transfers [5]. An HDFS cluster normally consists of one name node and several data nodes. A physical machine can both be a name node and a data node. It is very common to have a dedicated machine as the name node, which we will discuss later.

The name node manages the file system namespace, maintains the file system tree and the metadata for all the files and directory in the tree. The name node also persists an image of the file system namespace, which is an edit log of the name node and knowledge of which files are in which blocks. This image is not updated on every change to the file system, but it is updated regularly, so that a name node crash can be recovered with as little damage to the file system as possible. The image can also be updated with a command to the file system.

Data nodes are the workers of HDFS. They store and retrieve blocks when they are told to by the name node, and they report back to the name node periodically with lists of blocks that they are currently storing. This periodical report is called a heartbeat.

If the name node dies and cannot be recovered, there is no way to reconstruct the files from the blocks on the data nodes. For this reason, Hadoop provides two mechanisms to prevent this. The first is to backup the files that make up the persistent state of the file system metadata. The other is to have a secondary name node. However, since the secondary name node is not updated in realtime, data loss is almost guaranteed in the event of a total failure of the primary name node.

5.1.3 Summary

Here is a list that summarizes HDFS's design.

* HDFS is designed to store very large files. Sizes from hundreds of MB to GBs or TBs, which is possible with the concept of splitting files with blocks.

* Streaming data access with a write-once, read-many-times pattern. Makes it possible for multiple jobs to access the data in parallel.

* Designed for cheap hardware. Data is large and streamed. Streaming data is still very fast on hard drives and does not need expensive solid state drives.

* Not designed for low-latency data access. HDFS is instead optimized for delivering high throughput of data.

* Not designed for multiple writers or arbitrary file modifications. Files in HDFS may be written to by a single writer, and writes are always made at the end of the file

* Since the name node holds the file system metadata in memory, the limit to the number of files in a file system depends on the amount of memory on the name node. It is possible to store tens of millions of files, but billions is not feasible with current hardware[1].

[1]Storing 1 billion files in HDFS requires about 320 GB of memory.

5.2 Availability and usability

HDFS has a UNIX-like command-line interface. Commands like ls, put, mv and rm are familiar to anyone who is used to a UNIX or POSIX command-line. HDFS has interfaces for Java and C as well. There is also a service named Thrift that makes it possible for many more languages to communicate with HDFS [16]. We will not cover thrift in this thesis since we used Java for our implementation, but it increases the availability of HDFS. The Java interface is a rich library with many methods for reading, writing and getting status of files in the file system.

5.3 Data corruption

Every I/O operation on a disk or on a network, carries a small chance of introducing errors when reading or writing. When the data gets as large as with systems using Hadoop, the chance of data corruption occurring is high [5]. HDFS is designed to be fault-tolerant and handles this high probability of data corruption.

HDFS transparently checksums all data that is written to it and verifies the checksums when reading. A separate checksum is created for a configurable value of bytes, named "io.bytes.per.checksum". In addition, data nodes log all checksum verifications for keeping statistics of the system and detecting bad disks. The checksum creation and verification will have a small impact on performance, and it can be disabled.

These checksums are created and verified on active data, when reading and writing. Inactive data needs another protection against "bit rot". Hadoop has a background process called DataBlockScanner that is run periodically and verifies data on the data nodes, looking for bit rot [5]. When data is corrupt, the whole block that has the corrupted data is removed. The block is then replicated from another machine that has a non-corrupted copy of the block.

5.4 Data Reliability

Data reliability is achieved as long as there is at least one undamaged copy of each block. Replicating a block is a trade-off between read/write bandwidth and redundancy. To maximize redundancy, you would put the blocks in different datacenters. But that would also minimize bandwidth. To maximize bandwidth, you would put the data in the same rack[2] but on different nodes. But all the data would be lost if only a single rack gets damaged.

Hadoop has the following strategy since version 0.17.0.0, with the default value of the redundancy factor set to three:

1. Place the first replica on the same node as the client. The node is chosen at random if client is outside of cluster.

[2] A rack is referring to a network switch mounted on an equipment rack.

2. Pick a node in a different rack from the first one (off-rack), chosen at random.
3. Pick a different node on the same rack as the second one. Any later replicas will be chosen at random.

5.5 Transfer safety

Now that we know that data is safe within blocks and that blocks are redundant in a reliable fashion, let us see how the data is transferred safely to the HDFS cluster.

When a user of HDFS wants to put data on the file system it calls a create function on the client. The client will ask the name node for file permissions, and if all goes well, the name node will return data nodes suitable for storing the data. These data nodes form a pipeline, and the client will only send data to the first data node. When a data node gets data, it stores and forwards the data to the next node in the pipeline. The forwarding ends when the last node is reached. When the client has finished writing data it calls a close function. This action will flush all the remaining packets through the data node pipeline and wait for an acknowledge before signaling to the name node that the file is complete. See figure 5.1.

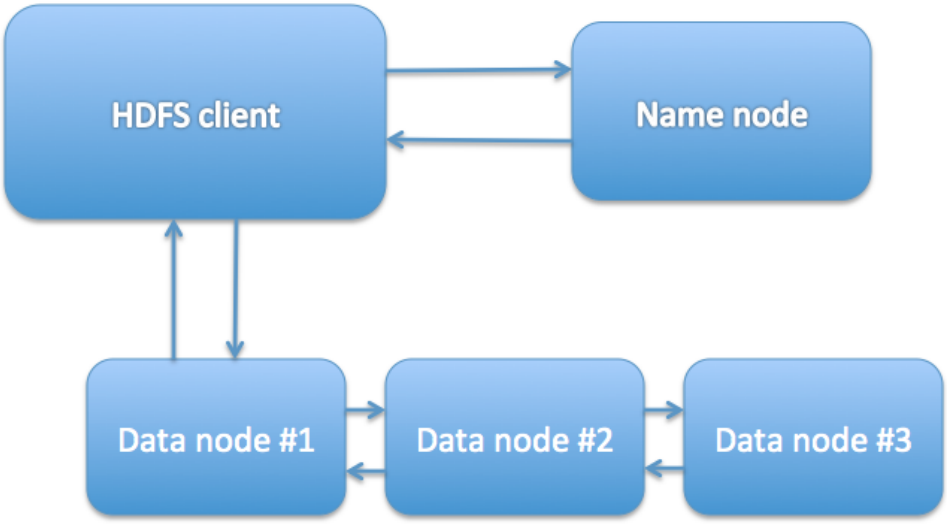

Figure 5.1. Communication between the HDFS client, name node and data nodes.

In case one of the data nodes in the pipeline fails during the transfer, the failed blocks will be marked for removal. The data will continue to stream to the other functioning data nodes, and HDFS will eventually see that the transferred blocks are under-replicated and find a suitable data node to replicate the blocks to.

The process is similar when reading data from HDFS. The only thing to note in the difference between reading and writing, is that if a read operation from a block

on a data node fails, that data node will be blacklisted so it is not read from again. If the block was corrupt, it will be marked as corrupt and replicated from another healthy block later.

Chapter 6

Implementation

This chapter discusses design decisions we made to write our code. First it explains on a high level how the problem was approached, and later it focuses on the details of how specific problems mentioned in chapter 2 were solved.

6.1 High level design

There were two major factors that we had to take into account when designing and planning how this project would be accomplished. (1) We had limited time and (2) we were working as part of a bigger project.

To save time, we decided to use third party Java libraries and open source projects wherever possible. It was also important to define the modules of the project independently from each other. By having well defined modules and using well known libraries, other developers in our team would have an easier time continuing our work.

In addition to the requirements of an archiving solution, being a part of the bigger project added the requirements of using common solutions for the following problems.

* Exposing functionality through UI[1].
* Configuration.
* Persisting the configuration.
* Logging.

REST[17] API was chosen to be the interface for exposing the functionality. A web server named Jetty was used to handle the web requests made to query the exposed functionality. Jetty is a lightweight web server that is designed to be a part of other applications [18].

[1] UI is an abbreviation of user interface

Several problems were addressed by using Jetty. We do not need to implement a web server, and the web server solves concurrency problems, thread pool allocation and thread pool management. The UI could be designed by a separate team as long as the REST endpoints were well documented.

For configuration Java Management Extensions are used [19]. This provides a project wise standard for handling configuration. The resources are stored in entities called MBeans. MBeans can be configured from a UI called JConsole. JConsole is a tool that comes with the Java JDK that connects to a running JVM[2] to inspect and modify the MBeans visible to that JVM. The JConsole makes it possible for us to test and configure our program while it is running.

The open source project Log4j from Apache, was used for logging states of the application. The logging is a necessity for tracing errors that occurs in different production environments. Log4j is a highly configurable logging library, which makes it possible for us to configure the logging late in the project, without having to change the source code [20].

The open source commons libraries from Apache was also used. These libraries improves upon the standard library of Java.

Figure 6.1 shows how every thing is connected together.

6.2 Splunk requirements

Splunk's architecture has constraints that need to be taken in to consideration when designing our solution. This section discusses and solves these constraints.

6.2.1 Uniqueness

A Splunk instance that stores and indexes data is called an indexer. Splunk can be setup with a single indexer or in a cluster configuration with multiple indexers. Each indexer can contain multiple indices. Each index contains multiple buckets of data. A bucket has an id that is unique within the index it belongs to. The id is not sufficient to identify a single bucket amongst multiple indexers. There may exist other Splunk indexers that have the same index name and may therefore generate buckets with the same index and id pair. This is why configuration parameters were introduced to distinguish between instances of Splunk.

The introduced configuration parameters are:

* Cluster Name: A given name to the Splunk cluster, unique in the archiver instance.

* Server Name: A given name to the Splunk indexer, unique in the Splunk cluster.

[2] JVM is an abbreviation Java Virtual Machine

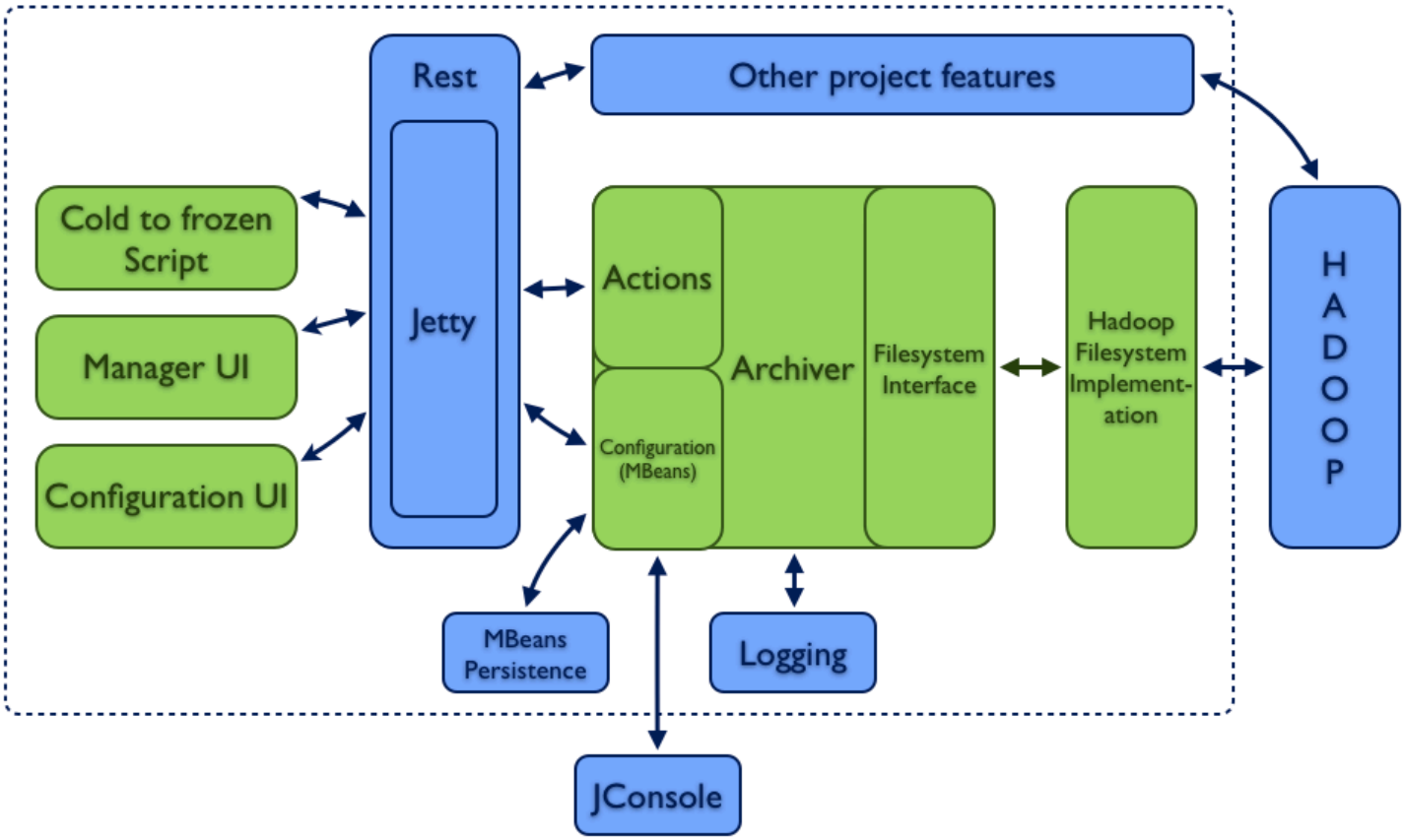

Figure 6.1. High level design of archiver with regard to the surrounding project. Archiver code are the areas colored with green.

6.2.2 Phase out script

When a bucket is phased out, Splunk calls a specified script with the phased out bucket as its argument. This is where our projects kicks in to handle the archiving of the provided bucket. After our script is terminated, Splunk will remove the bucket from local storage, regardless of what the script did[3].

6.3 Data archiving

The archiving process is the most important step. During this process, the data will be removed from one file system and stored in another one. Data loss is unacceptable

[3]Splunk wants to delete the bucket because it wants to free up space for new buckets. Splunk will not be able to index more data if the hard drive is full.

and it must not be error prone. Figure 6.2 presents a sequence diagram of how the archiving is done.

To begin with the bucket has to be moved to a safe storage place and this has to happen as fast as possible. The reason for this is that after the archiving code is run, Splunk will remove the bucket that was passed to the script. The removal of the bucket with occur for all outcomes of the script, including errors. We need to keep the bucket away from being removed, in case the transfer to the archive fails.

A transactional transfer strategy makes sure that there is no data loss. The requirement is that the data is either still on local disk, un-transferred, or it is on the archiving system. The local data is only deleted when the data is fully transferred to Hadoop. If a bucket is present on Hadoop that means that the transfer was successful. To make sure there is not any partially transferred buckets, we use a 2 step transfer:

1. Move the bucket to /tmp/real/path
2. If the moving was successful. Do a rename operation from the tmp location to the original one. /tmp/real/path -> /real/path

The move operation only changes a pointer on the name node and it will be atomic. Either the data was moved and the transfer was successful, or the move was not successful and the transfer should be redone.

Now that we have taken care of the data loss, we need to take care of re-transferring the data that could not be archived. This is done by letting a new archiving process also clean up the failed transfers. If a bucket is in safe storage it is either being transferred right now, or it failed to be archived by a prior process. After an archiving process has archived its own bucket, the safe storage directory is checked for additional buckets to transfer. To distinguish between a failed bucket and an ongoing transfer, a file lock is used. A new archiving process will archive every bucket in the safe location that is not locked and will acquire a lock on the bucket before starting the transfer.

Hadoop has an implementation of a DFS[4], but the design should rely on Hadoop being maintained. Thats why we created our own file system interface, that abstracts the file system functions from the archiver. This way it is possible to write other implementations that talks to other file systems. It was also very important to us that this interface is very thin, which it is.

6.3.1 Transactional nature of the archive file system

A transaction either changes the state of a system as requested or, in case of an error or a request for abortion, leaves the system in the same state as it began with. A transaction is, commonly, built out of multiple operations that have the following properties [21][22].

The ACID properties:

[4]DFS is an abbreviation of Distributed File System

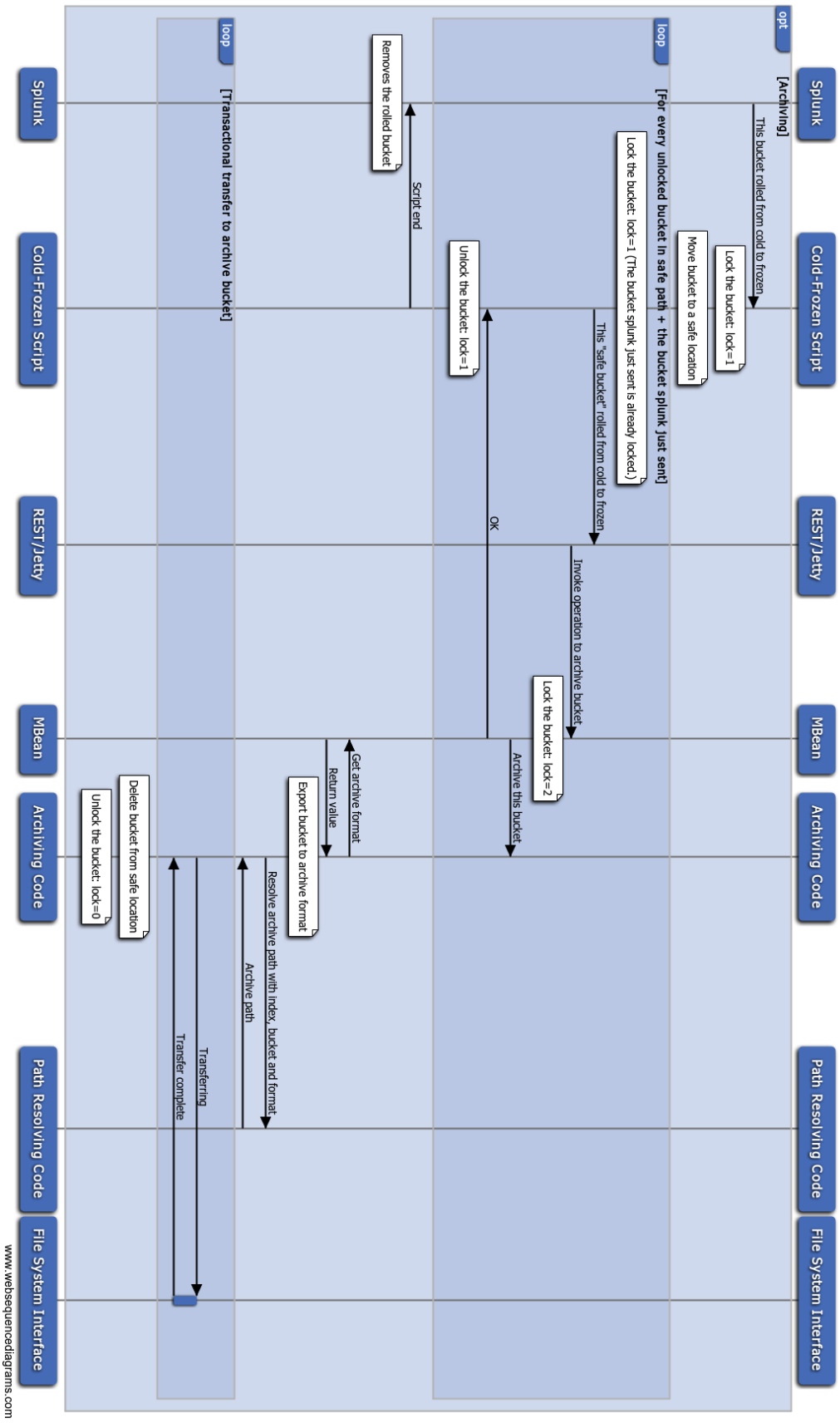

Figure 6.2. Archiving process describing how we lock data for concurrency and how we recover from failed archiving.

* Atomic: Either all the operations in the transaction is applied to the system or non of them affects the system.

* Consistent: If the system is in an allowed state before the transaction began, it will be in an allowed state after the transaction finishes.

* Isolated: A transaction that is in progress is not visible to the rest of the system.

* Durable: The changes applied to a system by a finished transaction is persistent and will survive a system crash.

Further, the phases of a transaction can be defined by following steps [23]:

1. Begin a transaction.
2. Execute all the operations in the transaction.
3. Commit the changes to the transaction.

In our system these steps are:

1. Lock the file in safe storage, and only allow a single transfer process to access the file.
2. Transfer the file from local safe storage to the archiving file system.
3. Commit the changes by doing the atomic move (rename) operation.

And the state of the system can be defined as:
The archiving files system contains a user specified root directory in which a directory structure containing the archived Splunk data persists. All of the data representing a Splunk bucket is either fully present in the archive directory structure or not present at all.

The transfer operation of a single Splunk bucket needs to be a transaction in our system. Here is how our bucket transfer operation fulfills all the ACID properties of a transaction:

Atomic

The transfer of the bucket is performed by a series of operations in a sequence. If a single operation in the sequence fails, the next one will not happen. The last operation in the sequence is the move operation which is atomic by it self. Further the system is not affected until the move operation is done.

Consistent

The temporary folder is not a part of the system. The important part is the directory structure in the archive root path. This structure is only changed once a move operation is made. Further a move operation adds a whole Splunk bucket

to the system, and thus the system is not put in an inconsistent state by the file transfer operations.

Isolated

While a transfer is being made to the temporary location, the archiving system is left unchanged, and therefore the bucket that is being transferred can not be used. For example, it cannot be recovered at the same time as it is being transferred. Thus a transfer in progress is invisible to the rest of the system.

Durable

Once the bucket is in the archive directory structure it is stored in the archiving file system. A file system is by definition a place where data is persisted between system life cycles. Once the Splunk bucket is in correct location (when the transaction has ended) it will survive system crashes.

The bucket transfer operation does indeed have all the ACID properties and is therefore a transaction.

Modern transaction example

While the need of transactions can be traced to accounting and bookkeeping practices [21], much earlier than computers, it is still needed in modern systems such as iPhones. An example can be found in objective-c library NSData class [24].

```
- (BOOL)writeToFile:(NSString *)path atomically:(BOOL)flag
```

The documentation explains the flag argument as follows

> If YES, the data is written to a backup file, and then-assuming no errors occur-the backup file is renamed to the name specified by path; otherwise, the data is written directly to path.

6.4 Scalability

The same archiving file system should be able to store big data sizes from a single data source as well as from multiple data sources. This is solved by a our path resolver.

The path resolver is used to convert a bucket's local path to where it can be stored in the archive. It is a mapper from the configuration parameters (mentioned in section 6.2) to a path on the archiver file system. It also makes sure that bucket locations does not collide with each other.

When search is made for buckets belonging to a specific index, path resolver will return a list of paths. While this list will contain a single path in the current implementation, the rest of the system is implemented to be able to use a list of multiple paths. This makes it possible for the path resolver to act as a load balancer, without the need of changing the rest of the system. For example:

1. In case a directory contains too many buckets, the large directory can be split in to many smaller directories.

2. A file system could get too large, and the path resolver could then direct the new buckets to a new file system.

Figure 6.3 show how the path resolver works.

6.5 Sustainable development

It is not just about making the product work for now, we also have to think about the future of the product. That is why we have many tests and high code coverage in the project, to make it easier for new contributors to add functionality.

The tests make sure that the system behaves in the way we intended it to. After tests pass we are able to refactor the code to make it more readable and understandable. At every step of the refactoring the tests are run, to make sure that the system still behaves as we want it to. The refactoring enable us to produce readable code by:

* Giving methods more expressive names.

* Extracting code snippets from a large method, that has a single purpose, in to their own method.

* Extracting methods that together solve a larger problem to their own classes thus logically grouping code that belong together.

With this approach we produce code that not only works but also reads well.

Tests together with readable code will ensure the future of the product. Future developers will benefit from the clean code and the tests will ensure that they do not break anything with the changes they make [25].

Figure 6.4 shows a screenshot of running our fast unit tests under 1 second. Figure 6.5 shows the code coverage of our archiving product. In addition to the unit tests, there are also acceptance, functionality and manual tests, to make sure the project work as a whole.

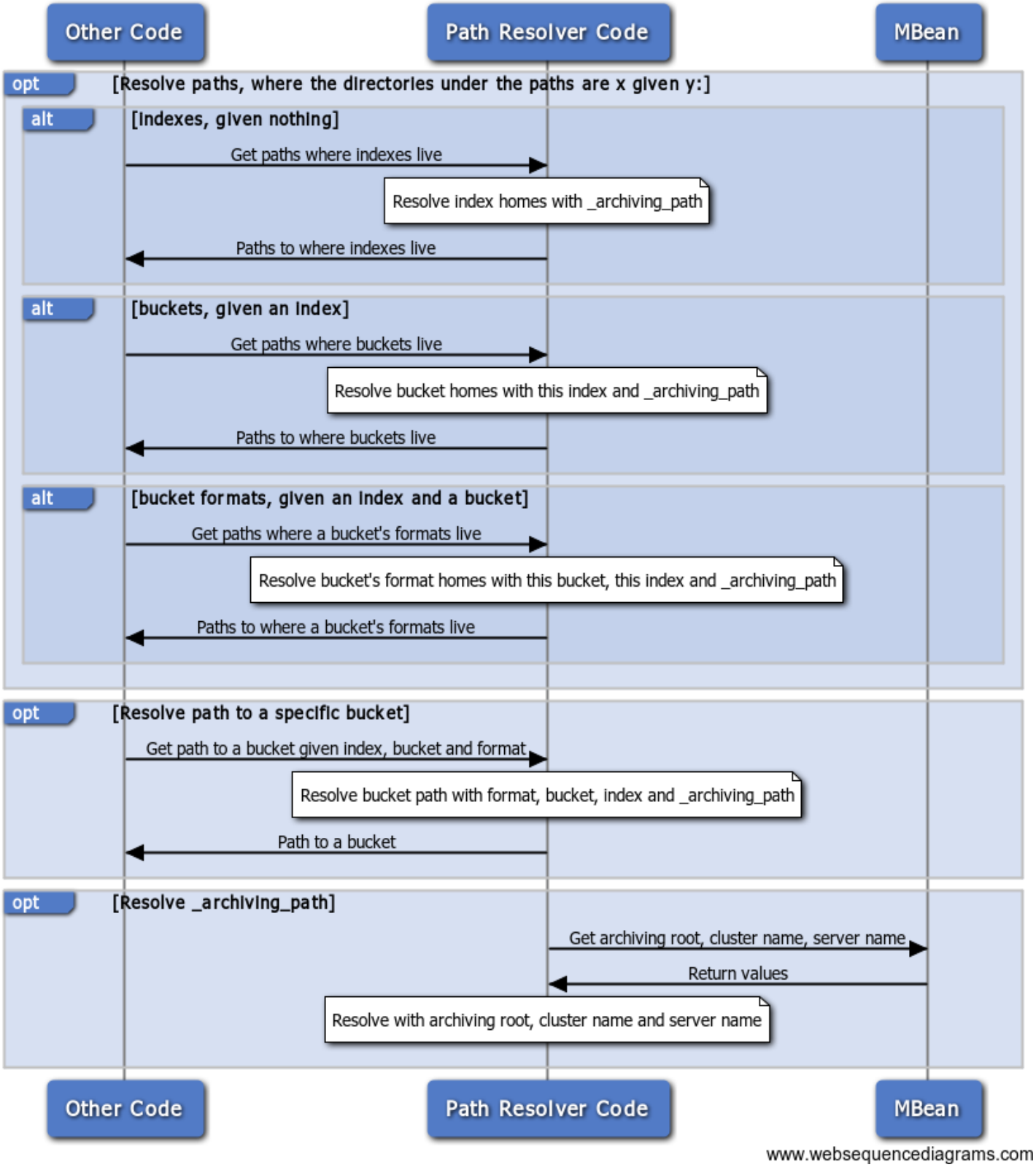

Figure 6.3. Sequence diagram over how the path resolver works and what it does.

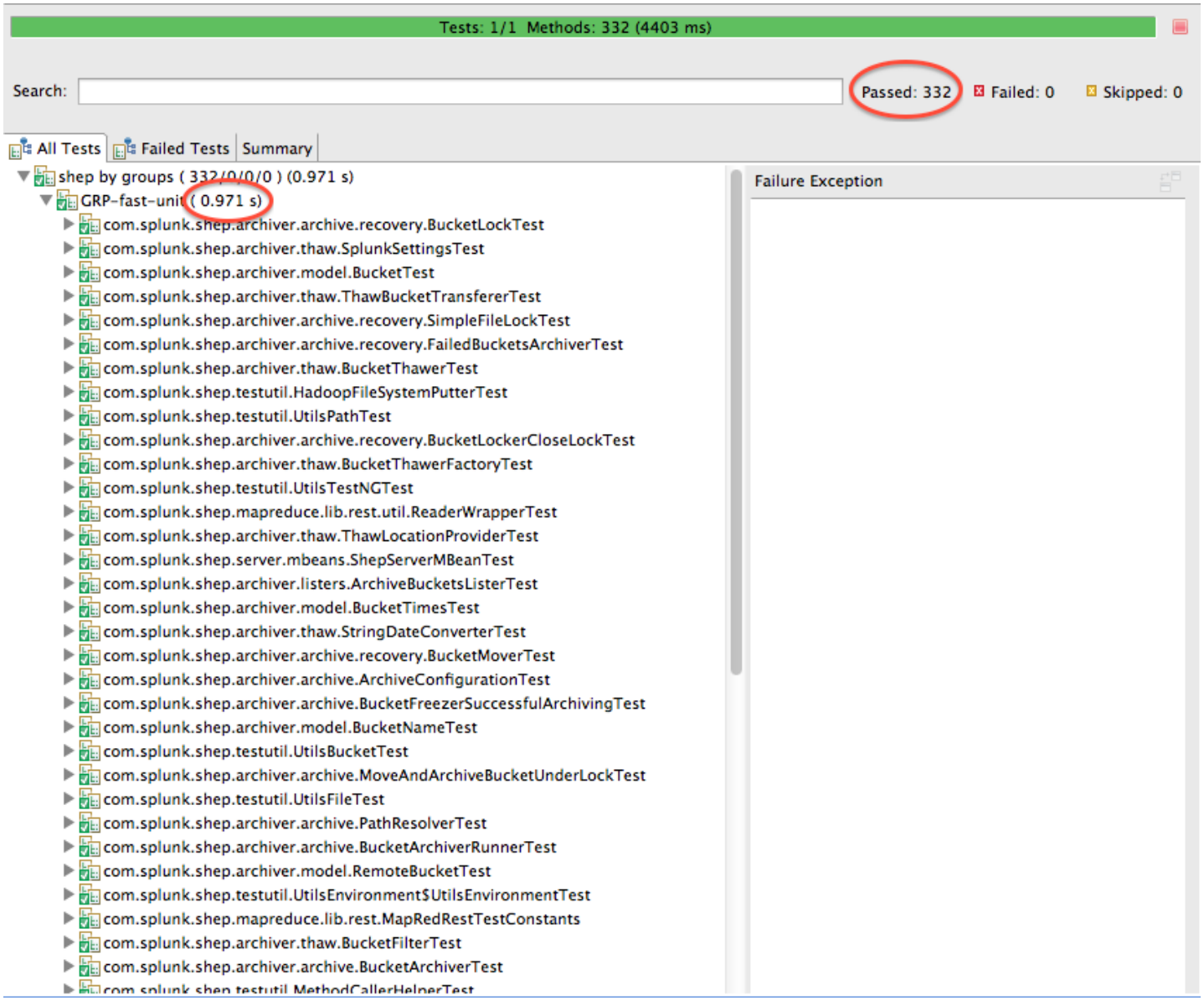

Figure 6.4. Running our fast unit tests. 332 tests in under 1 second.

Run fast tests (May 17, 2012 3:26:17 PM)

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
com.splunk.shep	0.0 %	0	191	191
com.splunk.shep.archiver	98.0 %	294	6	300
com.splunk.shep.archiver.archive	72.1 %	1341	519	1860
com.splunk.shep.archiver.archive.recovery	67.0 %	393	194	587
com.splunk.shep.archiver.fileSystem	66.8 %	267	133	400
com.splunk.shep.archiver.listers	43.6 %	160	207	367
com.splunk.shep.archiver.model	88.4 %	420	55	475
com.splunk.shep.archiver.thaw	68.9 %	617	279	896
com.splunk.shep.archiver.util	26.7 %	43	118	161
com.splunk.shep.cli	0.0 %	0	685	685
com.splunk.shep.customsearch	0.0 %	0	165	165
com.splunk.shep.exporter	0.0 %	0	735	735
com.splunk.shep.exporter.io	0.0 %	0	527	527
com.splunk.shep.exporter.model	0.0 %	0	56	56
com.splunk.shep.mapred.lib.rest	0.0 %	0	1038	1038
com.splunk.shep.mapreduce.lib.rest	0.0 %	0	1541	1541
com.splunk.shep.mapreduce.lib.rest.util	59.5 %	144	98	242
com.splunk.shep.metrics	0.0 %	0	22	22
com.splunk.shep.s2s	0.0 %	0	2130	2130
com.splunk.shep.s2s.forwarder	0.0 %	0	2427	2427
com.splunk.shep.s2s.forwarder.jetty	0.0 %	0	267	267
com.splunk.shep.s2s.timer	0.0 %	0	367	367
com.splunk.shep.server	0.7 %	4	573	577
com.splunk.shep.server.mbeans	42.9 %	493	655	1148
com.splunk.shep.server.mbeans.rest	0.0 %	0	1119	1119
com.splunk.shep.server.mbeans.util	63.5 %	87	50	137
com.splunk.shep.server.model	43.4 %	163	213	376
edu.jhu.nlp.language	0.0 %	0	3	3
edu.jhu.nlp.util	0.0 %	0	41	41
edu.jhu.nlp.wikipedia	0.0 %	0	1151	1151

Figure 6.5. Code coverage of our fast tests. Our part of the project is in the red rectangle.

Chapter 7

Partitioning the work

We worked together as much as possible because we wanted to discuss and solve problems together as they occurred, rather than solving problems alone and then reviewing each others work later. Here is a list of what we managed to do together:

* Researched the background to the problem.
* Defined the problem.
* Read the same base literature.
* Designed the system up front.
* Reviewed and edited the report.

7.1 Working separately

We divided some work between us to be able to work separately, in the event of when one of us was away from the office. Our work differ in reading, programming and how we wrote the first contents of the report.

7.1.1 Reading

Our differences in what we read are merely about implementation details. Petter read more about HDFS's internals, while Emre researched more about what tools that would help us during the development and the transactional file transfer mechanism.

7.1.2 Programming

This is where we divided the work the most. It can be argued that pair programming would have given us a lot of benefits, like design quality, less code defects and better knowledge sharing within the team [26]. But since we solved most of the problems in the design up front and we both practiced test driven development to reduce

code defects, we felt confident that dividing the programming task would work out just fine.

All the unexpected problems that occurred during the implementation were addressed together.

Petter implemented the preparation of the data before it was archived, including cases where a transfer failed and it had to be re-archived. Emre implemented the atomic transferring to an archiving file system and integration with third party libraries.

7.1.3 Writing the report

We wrote the outline of the report together, and then we divided the work as equally as possible across the outline. After we had filled in the blanks of the outline, we reviewed and edited the whole report together. Some parts were rewritten, some were left untouched and some part were removed.

Chapter 8

Result

While our project ended with a working implementation, we did not have the time to implement the user interface and it was instead assigned to other developers. The design solution with REST API made it possible to create a user interface by only knowing the REST endpoint. Making the code modular was one of our aims and seeing that others could take over our code partly validates this. Splunk is planning of open sourcing the project.

8.1 Performance

Figure 8.1 presents the benchmarking results of the archiver. The y-axes shows the time in seconds and the x-axes the bucket size in MB. Each bucket size is tested 4 times. The straight line represents the linear regression calculated using the least-squares method, and the volatile curve around the linear regression shows the average of the repetitions. The correlation coefficient is 0.720036 with the standard error of 0.000641.

As seen in figure 8.1, the times scale linearly with the bucket size. But there are some runs that generate irregular times. While we have not researched the reason extensively, our theory is that it is Java's garbage collection that is creating these irregular times. It may start in the middle of a transfer and delaying the execution of the transfer.

8.2 Problem statement revisited

We think that we succeeded with designing and implementing an archiving feature for Splunk that satisfies reliability, scalability and manageability.

We achieve reliability by transferring the data transactionally, locking buckets when transferring them to prevent multiple processes from accessing the same data simultaneously, moving the data to a location where it will live untouched and by retransferring the data that has not been successfully transferred.

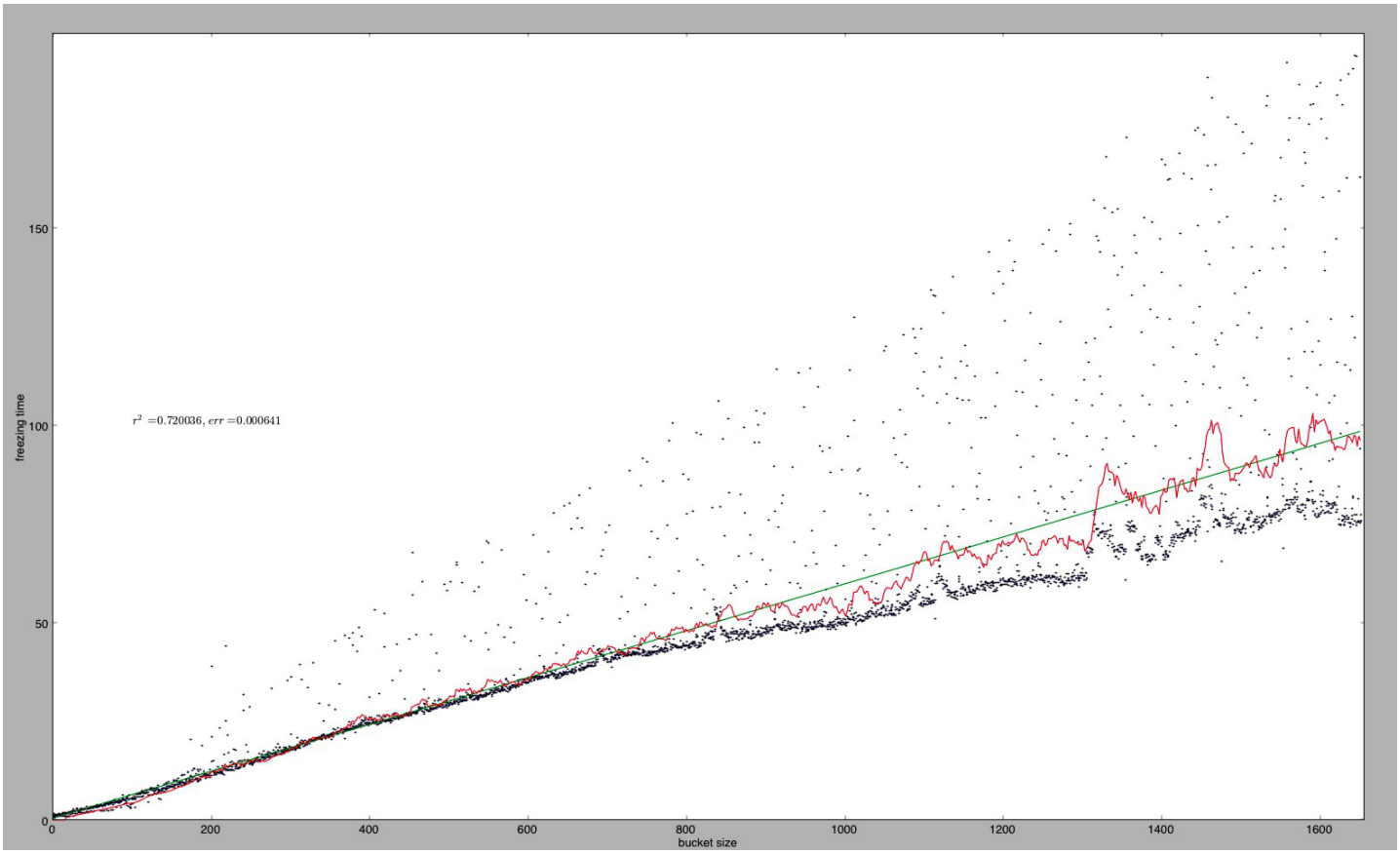

Figure 8.1. Benchmark of archiving buckets.

We achieve scalability by using distributed file systems, having an archiving file system interface that can be implemented by new file systems in the future and we have one single class that is responsible for where to archive the data.

We provide manageability by having a UI where users can monitor and control our archiving solution. We had to get help to implement the UI, but all the REST endpoints needed to create a UI was provided by us.

Chapter 9

Conclusion

This chapter will first discusses Splunk's intentions with our archiving solution, and then we discuss our thoughts about what can be improved.

9.1 How Splunk will use our solution

We conducted a mail interview with Boris Chen, the product owner of the archiver, to get a better understanding for the future plans of the product. His answers are indented.

What are the future development plans of the product?

> Possible new features include:
>
> * S3 integration for backing up data to a cloud service
> * localized in-place thaw operations - if the hdfs node is local to the splunk node, then do a softlink type operation rather than a copy
> * bucket rebalancing operations - customer want to see buckets of uniform sizes, which splunk does not gauratee, but can be done on frozen buckets

How will Splunk use the archiver to market it self to customers?

> This enables Splunk to utilize HDFS for storage and archive. But it is also a generalized solution for archive management. This is something not currently in the feature set of Splunk and has been requested often by customers large and small.

Which customers will Splunk target this product towards?

> Those needing an archive management solution that is open source.

What are the improvements of this product when compared to solutions implemented by customers?

> Customer solutions are simple script based solutions implemented by their respective IT departments. They are custom made, and closed source. This will be general and open source. The hope is that users also make additional enhancements to it.

Is there anything else you want to add?

> It is also a learning exercise into the limitations of Splunk's archiving and restore hooks, index management, HDFS integration issues, and other data transfer and scaling issues. As the product is used in real-life, we will no doubt continue to discover new limitations, and find ways of improving all these aspects further.

9.2 Future work on our solution

There are several improvements that can be made to the system.

One is using several Hadoop instances for load balancing purposes. This improvement can simply be implemented by letting the path resolver, mentioned in section 6.4, return paths that contain specification of which instance to use. The file system interface can then implement the logic for connecting to specified instance.

Another is to support other file systems. This is accomplished by implementing the file system interface for an other system. Amazon S3 is one of the possibilities.

While the archiving process is very solid, there are still some execution cycles that may cause errors. These errors are:

1. If the start script crashes before moving the bucket to the safe location, the bucket will be deleted by Splunk and the data will be lost!

2. If our archiving server crashes after the bucket was successfully transferred to Hadoop but before it was deleted from local file system, the bucket will be re-transferred the next time our start script is invoked by Hadoop. But since the bucket is already there, this will generate a FileOverwriteException.

The possibility that the first problem arises is low. The move operation is atomic and it is the first thing the archiver does. It could happen if a user sends a kill signal to the application. A power failure would also crash Splunk, causing it to run our script again once the system is up and running. However, this can be improved upon by letting Splunk do the moving operation to a safe path and then invoking our script.

This first problem could also arise if our script does not start. Our script should always start if it is configured correctly. We have integration tests for testing this case, so if someone does not use a version of the archiver where the test is failing, it should not happen. However, a human error can of course happen when configuring the script.

The second problem does not cause any data loss. Since we use transactional transfers to the archive, we know that if a bucket is in the archive, it is a complete bucket. So when we discover that the file is already in the archive, we will throw the FileOverwriteException. It can then be handled by deleting the local file.

Bibliography

[1] Splunk inc., "Splunk industries." http://www.splunk.com/industries, May 2012.

[2] E. Daniel, C. Mee, and M. Clark, *Magnetic Recording: The First 100 Years*. IEEE Press, 1999.

[3] R. Dee, "Magnetic tape for data storage: An enduring technology," *Proceedings of the IEEE*, vol. 96, pp. 1775 –1785, nov. 2008.

[4] Oracle, "Oracle introduces storagetek t10000c tape drive." http://www.oracle.com/us/corporate/press/302409(2012-05-14), January 2011.

[5] T. White, *Hadoop: The Definitive Guide*. O'Reilly Series, O'Reilly, 2009.

[6] Sun Microsystems, "Lustre File System - High-Performance Storage Architecture and Scalable Cluster File System." http://www.raidinc.com/assets/documents/lustrefilesystem_wp.pdf, December 2007.

[7] P. J. Braam, "The Lustre Storage Architecture." ftp://ftp.uni-duisburg.de/linux/filesys/Lustre/lustre.pdf, August 2004.

[8] N. Rutman, "Map/reduce on lustre - hadoop performance in hpc environments." http://www.xyratex.com/pdfs/whitepapers/Xyratex_white_paper_MapReduce_1-4.pdf, June 2011.

[9] M. Factor, K. Meth, D. Naor, O. Rodeh, and J. Satran, "Object Storage: The Future Building Block for Storage Systems." https://www.research.ibm.com/haifa/projects/storage/objectstore/papers/PositionOSD.pdf, June 2005.

[10] J. Layton, "Survey of file systems for clusters (and storage options)." http://gec.di.uminho.pt/DISCIP/MInf/cpd0708/PAC/survey%20filesystem.pdf, August 2005.

[11] Oracle, "Setting up a lustre file system." http://wiki.lustre.org/manual/LustreManual20_HTML/SettingUpLustreSystem.html#50438256_pgfId-1290698, January 2011.

[12] Amazon, "Amazon simple storage service (amazon s3)." http://aws.amazon.com/s3/, June 2012. [Online; accessed 5-June-2012].

[13] Hadoop wiki contributors, "Poweredby - hadoop wiki." http://wiki.apache.org/hadoop/PoweredBy, 2012. [Online; accessed 5-June-2012].

[14] Björn, "A hadoop file system driver for xtreemfs." http://xtreemfs.blogspot.se/2009/12/hadoop-file-system-driver-for-xtreemfs.html, December 2009.

[15] Venky Shankar and Vijay Bellur, "Glusterfs hadoop plugin." https://raw.github.com/gluster/hadoop-glusterfs/master/glusterfs-hadoop/README, August 2011.

[16] Apache Software Foundation, "Apache Thrift." http://thrift.apache.org, June 2012.

[17] R. T. Fielding, "Representational state transfer (rest)." http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000.

[18] The Eclipse Foundation, "About jetty." http://www.eclipse.org/jetty/about.php, June 2012.

[19] J. Hanson, *Pro Jmx: Java Management Extensions*. Apress Series, Apress, 2004.

[20] Apache, "Apache log4j 1.2." http://logging.apache.org/log4j/1.2/index.html, March 2010.

[21] J. Gray, "The transaction concept: Virtues and limitations," Tech. Rep. Tandem TR 81.3, Tandem Computers Incorporated, June 1988.

[22] Microsoft, "What is a transaction?." http://msdn.microsoft.com/en-us/library/aa366402(VS.85).aspx, April 2012.

[23] J. Amsterdam, "Atomic file transactions." http://onjava.com/pub/a/onjava/2001/11/07/atomic.html, July 2001.

[24] Apple, "Nsdata class reference." https://developer.apple.com/library/mac/#documentation/Cocoa/Reference/Foundation/Classes/NSData_Class/Reference/Reference.html, July 2012.

[25] R. Martin, *Clean Code: A Handbook of Agile Software Craftsmanship*. Robert C. Martin Series, Prentice Hall, 2009.

[26] A. Cockburn and L. Williams, *Extreme programming examined*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

TRITA-CSC-E 2013:006
ISRN-KTH/CSC/E--13/006-SE
ISSN-1653-5715

www.kth.se