BennuFC, a Distributed System for Document Management

Pedro Miguel Afonso Completo Bento

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Examination Committee

Chairperson: Prof. Mário Rui Fonseca dos Santos Gomes

Supervisor: Prof. Fernando Henrique Côrte-Real Mira da Silva

Supervisor: Prof. Luís Manuel Antunes Veiga

Members of the Committee: Prof. David Manuel Martins de Matos

June 2013

Abstract

Document management systems are crucial in any organizations. The storage infrastructure required by these systems assumes the existence of a shared file system with selective access by users and groups. In most document management systems there are web interfaces which can be used to add, update or delete files. Although functional and client independent, these interfaces require the user to access a web page to transfer documents, usually one at time. This type of interaction may become tedious when one has to manage a large number of files and documents. An alternative solution for file sharing systems within workgroups and organizations are distributed file systems (DFS). They allow a reliable storage by using high performance storage servers and a transparent access to user's files at different workstations.

The goal of this work is to develop a generic client application for a specific document repository at IST based on the Bennu framework. The client application must provide transparent communication with the data repository, access control at user and group levels, confidential access and must be seamless integrated with the existing authentication infrastructure and identity management subsystems. As this system will be used over unknown network conditions, it incorporates a delta encoder in order to ease the bottleneck constraints often imposed by the network.

Keywords

Client, File, system, delta, encoding, IST

Resumo

Os sistemas de gestão de documentos são cruciais em qualquer organização. A infra-estrutura de armazenamento necessária por estes sistemas pressupõe a existência de um sistema de arquivos partilhado com acesso seletivo por usuários e grupos. Na maioria dos sistemas de gestão de documentos, existem interfaces web que podem ser usadas para adicionar, atualizar ou excluir arquivos. Embora funcionais e independentes do cliente, estas interfaces exigem que o usuário aceda a uma página web para transferir documentos, geralmente um de cada vez. Este tipo de interação pode se tornar tediosa quando se tem que gerenciar um grande número de arquivos e documentos. Uma solução alternativa para sistemas de compartilhamento de arquivos dentro de grupos de trabalho e organizações são sistemas de arquivos distribuídos. Eles permitem um armazenamento confiável usando servidores de armazenamento de alto desempenho e um acesso transparente a arquivos do usuário em diferentes estações de trabalho.

O objetivo deste trabalho é desenvolver uma aplicação cliente genérica para um repositório de documentos específicos do IST com base na estrutura do Bennu. A aplicação cliente deve fornecer uma comunicação transparente com o repositório de dados, controle de acesso ao nível de usuário e grupo, acesso confidencial e deve ser integrada com a infra-estrutura de autenticação existente e subsistemas de gestão de identidade. Como este sistema será usado em condições da rede desconhecidas, será desenvolvido um *delta encoder* a fim de aliviar as restrições de largura de banda geralmente impostas pela rede.

Palavras-chave

Cliente, ficheiro, sistema, delta, codificação, IST

Index

Figure Index

1 Introduction

Back in time when the computer technology age first start taking over, some started to consider that computers could help companies to address a number of specific tasks in the business goals. Such ideas came from the fact that, although computers could not handle complex tasks or jobs that they were not programmed for, they excel at repetitive and redundant tasks. As so, computers could make workplaces more efficient by eliminating the errors from the human factor: they would never get tired, they would never need coffee breaks and they would certainly meet schedules. As computers became more prevalent there would be no necessity for additional costs in workmanship. Some would even integrate teams that repair and recalibrate those machines.

Also as computer became a norm it would be possible to reduce the need of paper, thereby cutting unnecessary costs and contributing to the protection of the environment. Another advantage would be the ease of access and organization of information without needing cumbersome installation filled with archives that are kept without knowing if they will ever be needed. Also Coopers and Lybrand study showed that an average office [1] [2]:

- Makes 19 copies of each document
- Loses 1 out of 20 office documents
- One file cabinet costs $25,000 to fill and $2,000 to maintain
- Handling paper documents consumes 90% of the typical office tasks
- Companies spend:
  - $20 on labor to file one document
  - $120 on labor to search for a lost document
  - $250 on labor to recreate a lost document
- Workgroups lose 15% of all documents they handle
- Workgroups spend 30% of their time trying to find lost documents

Those beliefs turned out to be truth. For many years, document management systems consisted in: management of files and physical filing and retrieve of information in their documents. But with the ascension of the first word processors in 1980, digital documents became a reality. With the increasingly usage of electronic documents, it established the necessity to organize all the data produced. Over the next decade, a number of electronic document management systems were developed. They supported electronic documents, but they were complex, expensive, hard to manage and a required lot of effort of users to be able to index documents. Also they had some limitations, like they only supported proprietary file types or only a limited number of file formats. Later, they evolved to the point where they could manage all kinds of data and with the growth of the Internet and improvements in development of computer networks, it became progressively easy to access information remotely.

Today, document management systems are crucial in any organizations. These systems are designed to keep all information of an organization and make it accessible to whoever is allowed to ac-

cess it. To ease the access, they assume the existence of shared file systems with selective access by users and groups. In most document management systems, there are web interfaces which can be used to add, update or delete files. Although functional and client independent, these require the user to access a web page to transfer documents, usually one at time. Likewise it can become tedious since web interfaces usually require the user to stay at same page for some time until the transfer is completed, particularly in large files

The most common architecture to transfer files is the traditional client-server model. The initial distributed file systems were developed sharing the same network infrastructure. However, with the increasing network bandwidth, it is now possible to implement reliable file systems over the Internet, while allowing them to be scalable. The most usual approach to maintain these systems scalable is by adopting multiple replicated servers. Still, this thesis tries to go a little further studying and implementing a way to help the system to reduce the burden of wide area network file transfer.

An alternative to web interface for sharing data files within workgroups and organizations are distributed file systems (DFS). They provide a reliable storage by using high performance storage servers and a transparent access to user's files at different workstations. This can be accomplished with the development of a client system that keeps a local copy replicated at user's workstation. Wherever the user modifies a file, it will automatically synchronize with the storage servers. Once the synchronization is complete, the new version is available to any other workstation controlled by that user or to the user's workgroup. In the same way, if someone else in the user's group updates a file, everyone one else in the workgroup will receive the update. Success uses of this approach are Dropbox and Google Drive, for example.

1.1 Objectives

The main goal of this work is to implement a client that is able to access in a transparent way the IST document management repository. In other words, build the client system to access an already existent repository which right now only has a web interface. The client should synchronize files automatically without any user intervention, except if there is a file conflict. In these cases the client will automatically rename the user's conflicting file and transfer the version from the server, so that later the user can merge the conflicting data manually.

The IST repository is currently coupled with a web interface that allows accessing it. As this is the only way to communicate with the repository, in order to be able to complete the main objective is critical to first create an API that allows the communication with the client.

The final goal is to create a solution to improve file transferences over the Internet, therefore improving scalability. There are two main options: build a solution based on peer-to-peer file transferences on the local network, so wherever a user needs a file from the repository, he would retrieve it from another user in the local network that has that file, instead of going to the repository. The second option is build a solution based on delta encoding of files.

Although both could be implemented, due to time constraints only one was chosen. The peer-to-peer solution could take away some load from servers, but it could also fail its purpose, if there are no sources in the local area, since it would still transfer the entire file to the repository.

On the other side, the second option would bring improvements in file transferences as long as the file contents do not change completely from version to version, which usually they do not. Its main disadvantage is the additional computation that is needed to encode and decode the file. As we believe that relying in the possibility of another user to join our local network with the exact files we needed could be unlikely, we decided to opt the second option, where the overhead in file transferences depends uniquely in file differences between versions. And as a further work, we propose the peer-to-peer communication between users in the same local network.

In sum, specifically the objectives are:

1. Study existing distributed file systems and its methods to share files;
2. Design and develop the client system similar to commercial solution, such as Dropbox [3];
3. Client should operate even when unable to communicate with server;
4. Integration with the IST's identity provider and document management repository;
5. Development of a solution based on delta encoding to improve file transferences;
6. Evaluate and test the solution in a simulated environment.

The objective 1 is the core of the related work and it will involve the study of some distributed file systems used nowadays and its methods to maintain all files synchronized, along with performance, high availability, scalability and techniques to reduce storage and data transmission.

Objectives 2, 3, 4 and 5 are the actual implementation using some of the methods previously studied. It will include a mechanism of persistent cache in client to support offline operation, similar to Coda disconnected operation, and file encoding. The integration with IST repository and identity provider is explained in section 2.1.

In the 6th objective, the system will be evaluated in a controlled environment. Results will then be analyzed with the fulfillment of the objectives.

1.2 Requirements and restrictions

Since this work is being made to a specific system and client, there is no total freedom when it comes to choose which technologies to use. As in the world of ICT, we are often limited to technologies that the client uses. For this reason, and because the system is being developed for IST, we are limited to the technologies used by the institute and the ones that it approves. As the work is being developed for a system that will go into production, after being tested, it was important to use only technologies that the institution adopts, so that the client application could go into production as quickly as possible.

The first requirement imposed for the client application was the portability. IST wants to reach as many systems as possible and with reduced changes to the application. To deal with this constraint,

we kept in mind to use a programming language that could offer such benefit. Since the IST repository was already programmed with Java, the choice was Java. Not only to reuse as much code as possible, to speed up production, but also because the client application will be maintained by the same team that works in the repository, which work mostly with Java.

The second requirement was relative to the authentication methodologies. The IST supports three authentication methods: Kerberos ticket system, LDAP and CAS. Although the institution offered some freedom on the choice of authentication method, he showed a particular preference for CAS, because the repository already supported it natively.

The third requirement was to use the IST repository. This system will be described ahead, as well, but its main constraint is the use of just one replica. This would invalidate the usage of solutions of server replication or cloud storage, but the institution does not put it aside, in future. In the current state, the repository does not offer any API to communicate with it, except locally, so there was some freedom to elaborate and implement an API that would allow the communication with it, over the network.

In sum, since the repository it is a recent development of the institute, it opted for taking a more conservative approach, in terms of technologies, to produce a solid and functional solution and eventually evolve it to match or even outtake existing solutions.

1.3 Summary

In this chapter was described the main problem around the document management systems and the motivation to solve it. Then were defined the objectives and requirements that the developed prototype must attend and demonstrate in the final solution.

2 Related work

This section presents the state-of-the-art containing the four major research and architectural topics relevant to this project. The IST Fenix and Bennu Framework for proper background, distributed file systems, cloud computing and efficient data transfer.

In the section of IST Framework it is made an overview of the tools provided by the institute that will be used in the project.

The distributed file system section covers distributed file systems and their advantages and disadvantages. Next it describes different distributed file systems each one presenting different concepts and ideas.

Finally, the efficient data transfer section describes mechanisms to shorten the amount of data needed to transfer to synchronize data across different hosts.

2.1 IST Framework

As we will see ahead DFS are here for a long time ago and there are some commercial solutions, then why not use those solutions? This question can be answer with multiple answers, but probably the most important is that files are often stored under an undefined jurisdiction. This can end up in files being leaked, because these solutions often have flaws in security, even the most popular ones such as Dropbox [4] [5]. Also there is an opportunity to improvements in DFS and integration with specific and already existing data structures.

2.1.1 IST document management repository

As stated before, this client is designed to work along with the IST document management repository. This repository is integrated with the IST Bennu Framework and it serves as an interface to access the repository file system. Right now, the only graphical user interface it provides is a web interface where after the user is authenticated, he can manage his repository data. The user is allowed to create, delete and rename folders and inside them to upload, download, rename and delete files. A user can also share those files with other existing users. Additionally, the user can also add extra information to his files using existing metadata templates, which eases the user search and simplifies the repository's organization.

The Bennu Framework [6] is an infrastructure being developed by the IST Fenix Team and its purpose is to ease the development of web-applications built as modules. In its actual implementation, Bennu is still in development, and it is not available to public. The entire framework is built in java, so it can run on any platform in which there is a Java Virtual Machine available.

The core features of this framework reside in standalone independent modules that are joined together with maven [7]. Among the various modules in Bennu's core features, there are two which are the most import for this work: the Authentication, which provides the API to communicate with the IST CAS server and the File Storage module, who maintains the server file system. This framework stacks over the Fenix Framework [8] which allows the development of Java based applications that need a transactional and persistent domain model. The following image represents the architecture of a web application using the Bennu Framework:

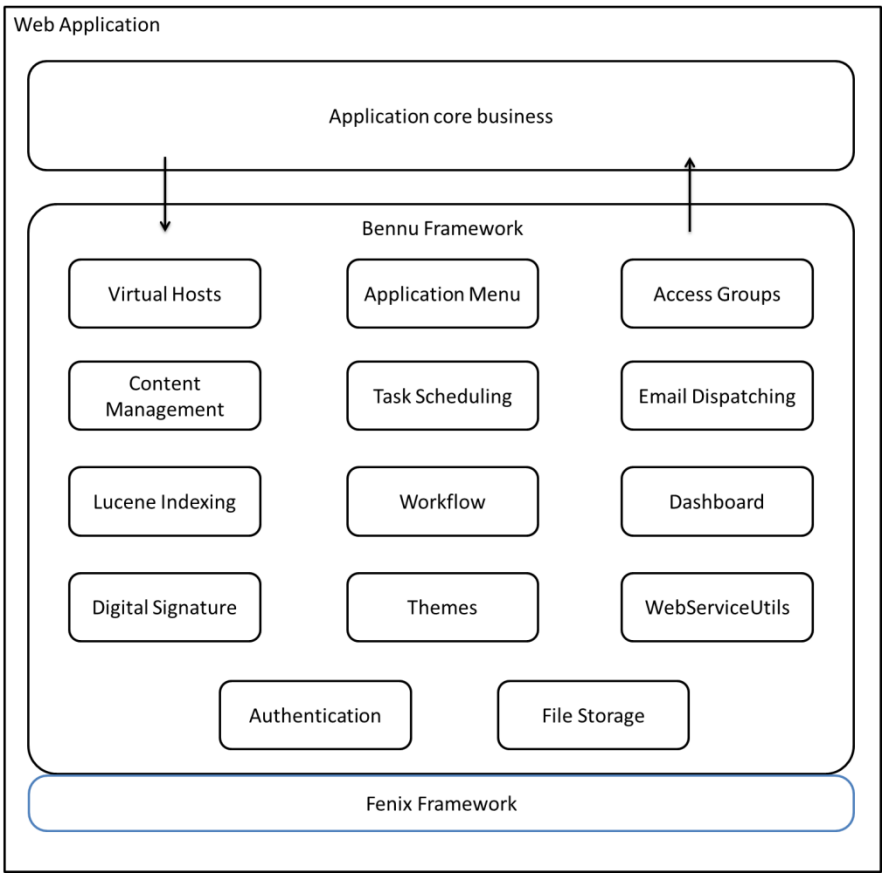

Fig. 1. Bennu's architecture

File Storage

The File Storage module is responsible for maintain an abstract File System over any operating system file system, for this reason the module is built in Java. The Fenix Framework plays a critical role on the internal structure of the repository, in such way that the repository structure was initially generated through a Domain Model Language (DML) file and then the functionalities were modeled over it. The DML file used is similar to the representation of a relational database schema, which generates the base classes and the related functionalities to interact with the database. This library gives yet the advantage that the database is completely hidden from the programmer, so he only needs to

worry about all the application coding, while the library provides the persistence. As so, the repository internal FS structure can be viewed as an E/R schema:

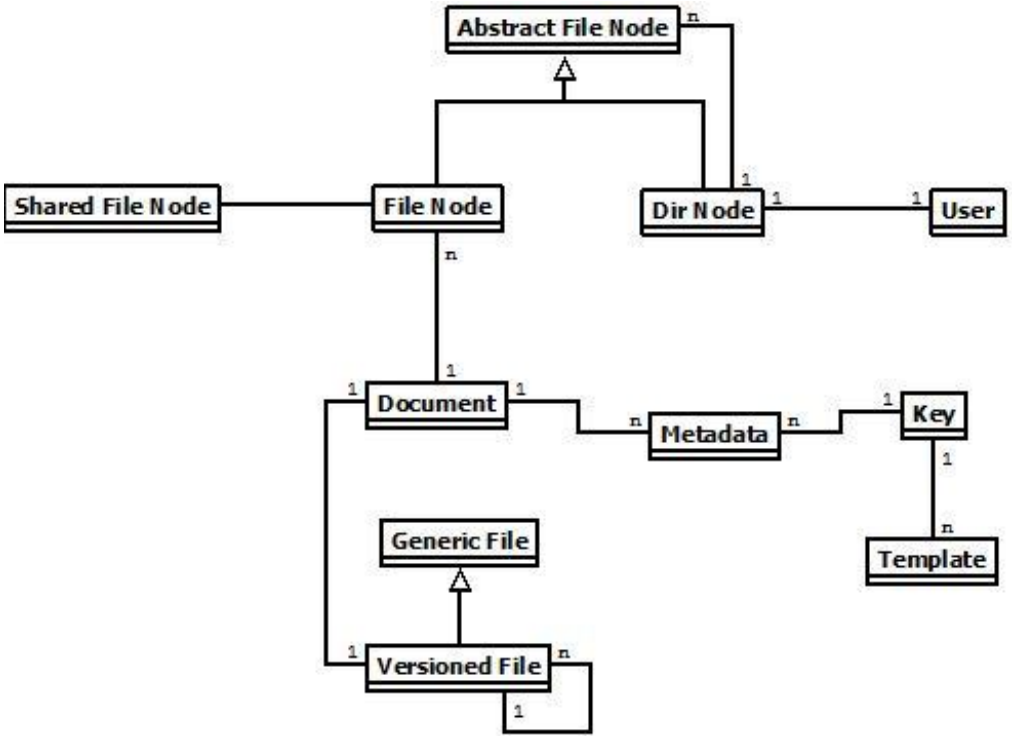

Fig. 2. IST Repository file system schema

The idea of this FS is to support multiple users, in a way that the FS content managed will depend on the user being authentication. Like so, the file system root directory is retrieved from the User. At start, that directory will be empty, but the user can create other directories or upload files, which create new Directory Nodes (Dir. Nodes in the image) or File Nodes, respectively. So each Directory Node represents a directory in the file system and a File Node a file. Every time a file is uploaded with the same name, it is added a new Versioned File to the list of versioned files in the Document. The document represents the contents of a file and can be retrieved from the corresponding File Node. By default, the contents retrieved correspond to the last version of the file, but it is also possible to get an older version. As said previously, every file can have extra information, which is kept in the metadata file that can have more than one template, for example a file can have the template information of a book and an article.

The File Store also has two options to keep the physical contents of the files. The first option is the database which stores the contents of the files as a binary object of a variable size (blob [9]), which can hold until 4 Gigabytes of data. The only limitation in file size is imposed by the MySQL implementation of LONGBLOBS [9]. The second is the physical disk who creates a directory for each user and that directory will be the user's root directory for the FS.

Regarding the file conflict resolution, when two users upload the same shared file at same time, the system will use the “first come, first served” policy. In this case the files will be uploaded and each will get a different sequential version in relation the previous file. The last version of the file will be the one of the last user who uploaded the file (last write wins). In this system this policy is acceptable as the contents of every file are kept in the repository and they can be recovered.

Additionally the File Store also supports a logging system. This is especially useful if a user is sharing a file with others and wants to know when the file was modified and who performed the modification. The logging system also covers the other file and directory operations. Another use of the logging system is to rollback unintended modifications or recover a previous version. Just as the repository file system, this system is also mapped in the database and its internal structure can be seen as an E/R schema:

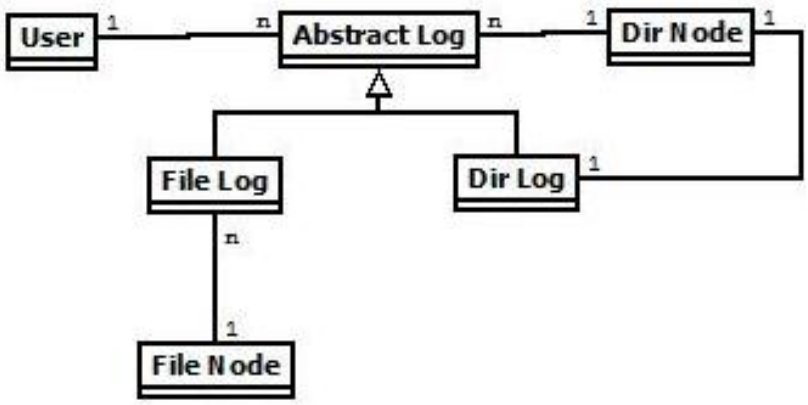

Fig. 3. File Store logging system

In this system, every user has a set of Abstract Logs which describe the events occurred over his folder and files and shared files. Alternatively, they can also be retrieved from a user Directory Node, but they will only contain information about events occurred in that directory. Each of these Abstract Logs can represent a File or a Directory Node and has the event associated: create, delete and modify. Each node also keeps the information about the state when the changed occurred.

In the context of the client application being developed, these logs are of extreme importance as it is based on them that the client knows what files and directories were modified.

2.1.2 IST Identity Provider

Due to the variety of applications that can be deployed at IST it was decided to support more than one authentication protocol. Currently, it supports Kerberos authentication protocol and Central Authentication Service (CAS).

2.1.2.1 Kerberos

Kerberos [10] [11] is an authentication protocol developed by MIT that provides a strong authentication for client and server application, mainly through the exchange of cryptographic secure tickets over a non-secure network. For this reason, both the end-user and the service provider do not trust each other, but they do trust a third-party common server, the Key Distribution Center (KDC). The Kerberos authentication protocol is based on shared symmetric key cryptography of both the user and service which are known only by the trusted server and the entity itself.

As previously stated, the KDC is a third party server, which in this protocol is considered a trusted server. Its only function is to issue tickets and depending on the specification, it can also be divided in two subcomponents:

- Authentication Server (AS) – Its function is to issue tickets for user authentication, the ticket granting ticket (TGT);
- Ticket Granting Service (TGS) – It provides the Service Tickets (ST), which the user will use to request services.

Protocol

In the Kerberos protocol [4] there are two types of tickets:

- Ticket granting ticket (TGT) – A long term ticket;
- Service Ticket (ST) – Short term ticket.

The TGT is issued in the beginning of the protocol when the user wants to authenticate itself. Once the user has the ticket, he can request specific services from other servers. To do this the user authenticates itself in KDC using the TGT and the KDC delivers him ST for that specific service. The ST can then be used to authenticate the user to that service.

For the following protocol description, we will assume that both AS and TGS are joint in just one entity, the KDC.

User authentication

The protocol initiates with the user authentication to the KDC. In order to do this the client sends his User ID (or username) to the KDC, so it can issue a TGT. The message that goes to the KDC is in clear text and never contains the user password. The KDC checks if the user is registered in the local database and, if it is, it responds with two messages.

The first message contains the Client/KDC session key that the KDC attributed to that user and then it is encrypted with a secret key that only the user and the KDC know: the user password.

The second contains the TGT which holds the client identification and the Client/KDC session key encrypted using the secret key of the KDC. This way, the user will not be able to decipher it.

When the user receives the first message, he must decipher it using his password to extract the Client/KDC session key. If the message has been modified or the user does not have the password, he cannot extract the session key. From now on, wherever the client needs to communicate to the KDC or otherwise, the messages will always be ciphered using this session key.

Requesting access to a service server

To ask for permission to a network service, the client must request at KDC for a ST. The request is made through two messages:

- The TGT received in the previous step and the identification of the service.
- The client identification and timestamp encrypted with Client/KDC session key.

Also the requested service must be registered at the KDC previously. This second message it is also used to validate the freshness of the message and authentication of the KDC.

The KDC answers with two messages:

- The client's identification and a Client/Service session key, encrypted with the Service key. For this reason it is necessary that the service is registered previously to the KDC.
- The Client/Service session key encrypted with the Client/KDC session key.

Accessing to the service server

At this step, the client it is ready to access the service server. At first the client sends two messages:

- The client's identification and a Client/Service session key, encrypted with the service key. This was the first message that the KDC responded to the client, in last step.
- The client identification and timestamp encrypted with Client/Service session key.

The service must decipher the first message with his own key to retrieve the Client/Service Key. Using this session key, the service deciphers the second message and responds to the client with the timestamp on previous message plus 1, encrypted with the Client/Service key.

The client deciphers and if it matches, it means that the service server it is trustful and it requests the service. Finally the service server provides the requested service to the client.

This protocol uses strong cryptography, allowing that a client proves his authenticity to a server (and otherwise) in a non-secure network. But it has a single point of failure: The KDC. If it fails, it is impossible to authenticate. Also if the KDC is compromised, an attacker can impersonate any user.

2.1.2.2 Central Authentication Service

Central Authentication Service (CAS) [12] [11] is a service which uses a single sign-on authentication process permitting a user to provide his credentials once to have access to multiple applications. There two components and roles in this protocol: the CAS client and CAS server.

The CAS client could be any web application that uses the CAS protocol. It is the responsible to provide the services requested, but only to authenticate users in the CAS server

The CAS server is the centralized authentication server which authenticates the users. This server is the only entity that knows the user authentication credentials and for that reason only it can issue tickets and verify identity of authenticated users to CAS clients. In the backend, CAS is supported by different authentication services, namely LDAP [13] and Kerberos.

Protocol

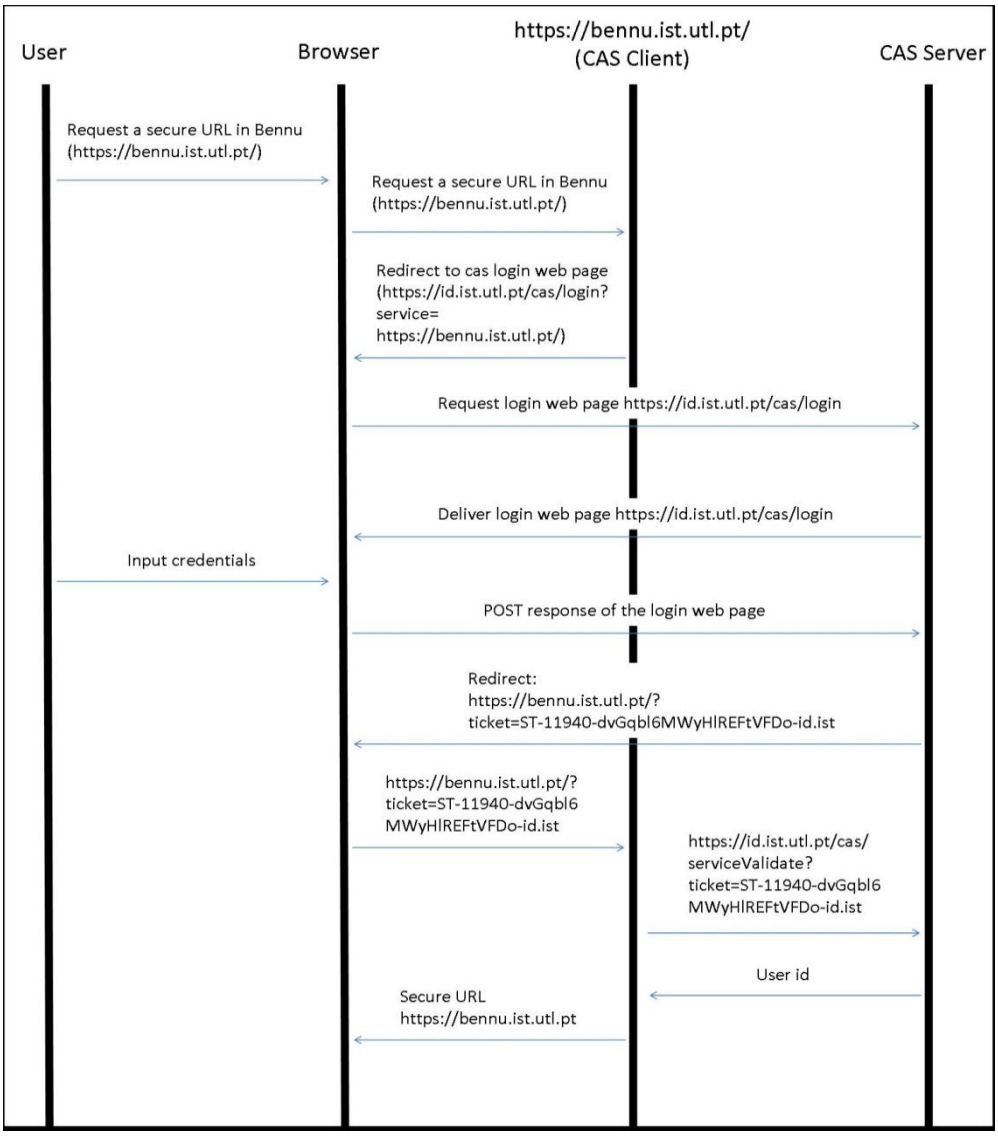

Fig. 4. CAS authentication protocol

1. The user starts by trying to access the URL of Bennu application (https://bennu.ist.utl.pt).
2. As it is a resource protected by a CAS client, the Bennu can only be accesses by authenticate users.
3. Seeing that the user is not authenticated, the CAS client redirects the user to the login webpage (https://id.ist.utl.pt/cas/login?service=https://bennu.ist.utl.pt/). Additionally, it is also passed the service that requires authentication.
4. Request the login web page https://id.ist.utl.pt/cas/login
5. Deliver the login web page https://id.ist.utl.pt/cas/login
6. The user inputs its credentials in the login webpage (https://id.ist.utl.pt/cas/login)
7. Post (HTML method) the credentials to the CAS Server.
8. If the credentials are correct, the CAS Server provides a link to the service that previously requested the authentication and the service ticket (ST).
9. From the previous link the user is redirected to the CAS Client by the browser, delivering the ST.
10. The CAS Client sends the ticket to the CAS Server to confirm the validity.
11. If the ticket it is valid the CAS Server returns the User Id.
12. Finally, it is created a cookie with information about the authenticated user and the protected content is returned to the browser.

In the further communication between the browser and the application it is always used the cookie to identify the user, so there is no need to re-authenticate through the entire process.

Single Sign-on

After the successful login, the single sign-on is enabled, because in the previous algorithm there were in fact two authentications.

The first was in steps 9 to 12 when the user delivered the ticket. Here it was also generated another cookie, which bounds the browser to the CAS Client and controls the access of the user to the web application.

The second was in steps 7 and 8 when the user delivered his credentials to the CAS Server. Here it was generated a cookie which bounds the browser to the CAS Server.

Suppose now that the user, already authenticated with the previous CAS protocol, now seeks to access another protected page: https://www.ist.utl.pt.

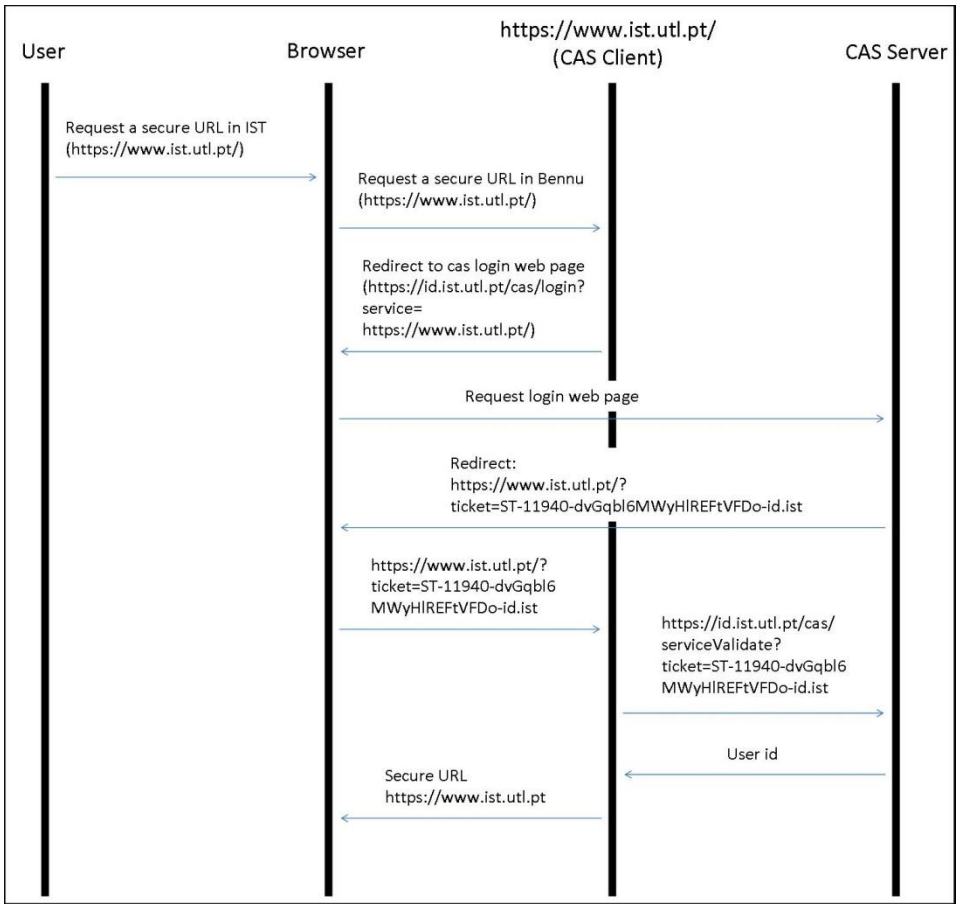

Fig. 5. CAS single sign-on feature

It will follow the same flow, but at step 4 when the browser requests for the login web page it will also send the previous generated cookie. As the user already sent previously his credentials, the CAS Server does not return the login web page, as previously, but instead, a ticket ready to be delivered to the application. From here the flow it is exactly the same as in previous.

Another advantage of this protocol is that it allows an application to authenticate a user without ever knowing user's security credentials.

2.2 Distributed File Systems

A distributed file system [14] is composed of components located at network computers which communicate and coordinate their actions using messages. The spread of components that at first seem to be a bad idea, allows a system to share different kinds of network resources and to be easily extensible and scalable. However, such systems share the same design issues of traditional local file systems and, additionally, users are spread over the network. This characteristic produces added latency, due to internode communication, and can limit the availability of the system. The main area of research in distributed systems is how to overcome these limitations so that distributed systems be-

come as fast and reliable as local file systems. Specifically, the problems that distributed systems deal with are concurrency, the need of a global clock for synchronization, failure handling and scalability.

A distributed system is composed by components that are executed concurrently. These, however, do not perform in the same way, i.e. the components can differ in type, for example printers, hard drives, which have their own access and operation times; and components of the same type can also have different performance depending on hardware. Also some of these resources cannot be accessed concurrently as it is the case of printer or a write of file. As such these resources have to be specifically coordinated by system components.

Another problem is the coordination of actions by the various components. Each system has its own clock. While this problem can be solved synchronizing the clocks through the exchange of messages, its accuracy is limited by the round-trip-times (RTT). For this reason the state of other components actions must be coordinated via messaging.

Distributed systems are composed of network systems which in its turn are made of distributed components. Like any system, these systems can also fail. However the detection of failures is complex due to the nature of network faults. For example, when a system stops responding it may be because there was a network failure, leaving the system unreachable to some or the whole system has crashed. It is difficult to know if the network has failed or it became unusually slow. It must also be taken in account that each component can fail while others continue to function. As so, each operation should be acknowledged to verify the operation was indeed completed successful.

One of the motivations to use distributed systems is resource sharing. With the increasing number of users it is important that the system is able to scale to maintain the performance. Usually this is accomplished in two ways: replicating resources by adding more resources to allocate among users, while also keeping some as a backup, in case of failures; or by using decentralized algorithms moving some of the load away from critical systems. Examples are peer-to-peer systems where the data can be transferred from closer nodes. This not only contributes to a faster transfer but also to decrease the amount of traffic at the original data store servers.

As seen in previous paragraphs, distributed systems present a number of problems that are not found in centralized ones. The latency, changing network conditions and the rogue node makes it challenging to develop efficient large scale systems. So why solve all these problems if a local storage system is able to do the same thing and in a much easier manner? Several motivations justify this approach.

Mobility

Nowadays, with the increased amount of available devices capable to connect Internet and the advances in wireless data communications, Internet is a resource available almost everywhere. These devices range from the traditional desktop computers to laptops and mobile phones. Distributed file systems can provide a generic interface to access large amounts of data even to portable devices whose storage is very limited, as it happens with the mobile phone.

Another point is that usually in an office, work or home environment users do not have a dedicated workstation. Distributed file systems can facilitate the user mobility since the information is stored physically in a server and made accessible to the user in another workstation, or as mentioned before in another device.

Data sharing

A distributed system is one of the easiest ways to share large amounts of data among users. A user can take the advantage of his own data being stored in a server and to allow access to a restrict number of users. As a distributed file system always provides an up-to-date view of the shared data, once a user updates a files everyone else can see those changes. However it should be noted that not all distributed file systems support or are designed to keep consistency guarantee when a group of people are modifying the same files, at same time.

High availability

Nothing is more frustrating than spend days and months writing and formulating documents and then suddenly our data is lost due to software or hardware failure, especially hard disk devices. Distributed file systems are computer systems designed for high availability with much reliable software and controlled software. And even if a server node fails there are always other replicas with the most up-to-date data to replace the failed system. Given this implicit backup systems it makes more sense to use these highly reliable file systems than our own computer systems to keep data. Also should be noted that not all distributed file systems use replication.

Scalability

In order to keep the reliability in distributed file systems, it is usual to have replicated file server systems to share the load among them. Still, the typical bottleneck in these systems, and the limit to transfer rate is the Internet. An alternative to replication that maintains scalability is peer-to-peer file transfer. This method has the advantage of

The following section will present several distributed file systems and some techniques that can improve the design of a distributed file system. Coda file system is much more detailed in this section than the others as its solution is similar to the system that will be developed in the scope of this thesis.

2.2.1 NFS

Network File System (NFS) [15] [16] uses the client-server paradigm to allow multiple distributed clients to access shared files. However there is almost no distinction between clients and server. A server exports file systems or subdirectories that can be mounted by clients. The same way, a client can also export his own file system to be mounted by other clients. The NFS clients have a local

cache that is connected by a network to a file server with a disk and a local cache. At start, clients cache all file attributes and only small blocks of files from the servers. In order to read the contents of a file, the client look for data in its local cache. If the data is in the cache, the client does not need to communicate with the server. Otherwise the client has to request the contents from the server. If the request is not stored at server's cache, then server will have to retrieve the data from the disk, updates it to his cache and send data to client which then will update its cache.

One of goals of this file system was to be simple. This brought some problems such as cache consistency and performance. The cache consistency problem is caused by the nature of NFS server: they are stateless. This means that they do not keep the state in main memories and in particular they do not gather information about which clients are using which files. This led to consistency problems, because since the server does not have that information, it cannot inform the clients when a new update is done by another client. So there is a possibility that some clients may continue to use the old data. To address this problem, clients poll server periodically to check if they have the most recent updates. This creates another problem that is performance. With the constant polling, it creates an extra load on server side and adds latency to accesses of client's files. Another problem of performance is that most implementations of NFS use write-through on cache policy. So when a client closes a file that was modified, it sends the new updates back to the server. A relevant problem is that the application blocks until all the data from the file is written in the disk of the server.

2.2.2 AFS

Andrew File System (AFS) [17] was created at Carnegie Mellon University (CMU). Later it became a commercial product by the Transarc Corporation, who was later purchased by IBM. AFS is today used by hundreds of organizations, continuing to gain more worldwide support. The original idea was to be a computing facility to share data among users with the ability to scale well and be secure. Recently this File System became an open source project named OpenAFS.

As the previous FS, AFS uses a client-server model. However there is clear distinction between the clients and servers. These workstations use a file name space divided in two areas: a local name space that is used for temporary and system administration files and a shared name space that is where the user's personal files are stored, under /afs directory. Security is assured using encryption in transfers between clients and servers and making server physically secure. When a user opens a file in his area (shared name space) the client copies the file in 64kb blocks from the server to his local disk and it is treated from now on as a normal local file. In opposite to NFS, where the servers were stateless, the AFS servers are stateful, so they record that a client has cached a specified file and it is created a callback. While this callback exists, the file is valid and it is the most up to date version. On close if the file has been modified, its contents were written back to the server. This way all callbacks related to that file become invalid and others clients had to reopen the file to get the new version. Although this process guarantees consistency, it does not ensure that one user will not overwrite another

user changes and if two users close the file simultaneously, it can lead to inconsistent and unpredictable results. Since the AFS clients only interact with servers when a file is open and closed this allows taking much load from the servers increasing the scalability.

2.2.3 Coda

Coda [17] [18] [19] is a direct descendent of the previous distributed file system, AFS. Over the period of development of AFS, several versions were developed. Coda implements AFS-2 focusing on its best features: scalability, performance and security and, additionally, Coda also provides high availability. This is accomplished by two of the main features in Coda: server replication and disconnected operation. This later feature was mainly driven by the appearance of mobile devices. These two features and some others are discussed below.

Scalability

Coda follows the paradigm of client-server considering that there is a small set of servers and a much large number of clients. The group of servers is designed Vice and is dedicated UNIX file service which is physically separated from clients. On the clients, the operating system is responsible to intercept system calls, such as open, modification and close files, and sends requests to a process called Venus. This process is a cache manager that communicates with Vice. In order to achieve the scalability meter, most of the work is done by the clients.

Performance

One of key elements in Coda for performance it is his extensive use of cache. Besides caching the directories and symbolic links, to resolve pathnames locally, Venus also caches entire files instead of blocks. This removes the additional overhead that is generated by the transfer of blocks as well simplifies the cache management.

Like AFS coherence of cached objects is maintained using callbacks. So when the client caches an object from vice, it is created a callback which is a promise that if the file is changed on Vice, Venus will be informed that the object has been modified. These callbacks are arranged in a table that maps the object to a list of clients that are interested in it. When the object is updated, this entry is removed from the table and the clients are informed of the change. Yet since it is possible to occur network partitions, Venus can no longer be notified by Vice of the callback break. It will be discussed in disconnected operation the solution to this problem.

Security

Security it a major concern in distributed file systems given that the unauthorized access or unwanted modification of files could be disastrous. To prevent this security problems are addressed in several ways. First, clients and servers are physically and logically separated, therefore servers are

considered secure. They are located in physically secure areas and run trusted software. On the other hand we have clients that are considered unsecure. They are not physically secure, can run modified software and hardware and the connection between clients and servers can be object of eavesdrop. To sum up, the first measure is to settle servers in a secure location and take in consideration the possible attacks of unauthorized access coming from clients. Second the client-server connections established must be encrypted. This is accomplished using a variant of the Needham and Schroeder private key protocol. And finally, it must support directory access control lists. In case of an unauthorized intrusion only the objects which that user has access can be harmed.

High availability

High availability of data is crucial in this system and one of the great improvements over AFS. This feature allows the system to provide uninterrupted service to clients through data replication. Coda uses two complementary replication mechanisms: server replication and disconnected operation. The server replication involves storing copies of data at multiple servers. Having multiple copies of data over several servers increases the probability that at least one replica is available at any given time. However even if there are no servers available or there is a network failure, Coda still allows the client to continue operate. This is due to his second mechanism, disconnected operation. The following sections will discuss these mechanisms in detail.

Server replication

As stated before, server replication decreases the probability of data is unavailability by storing it at several servers. In this process, servers replicate volumes (aka file sets) which are distributed among several servers. The servers which contain the volumes required by a specific client are called volume storage group (VSG). However at a given time, a client might not be able to contact all the VSG, because of server or network failures. The set of servers that the client is able to reach is called the accessible volume storage group (AVSG). This AVSG can be different from client to client even if they are using the same volume, given the problems mentioned before.

Coda uses an optimistic replication scheme in which replicas are allowed to diverge. This strategy is the opposite of pessimistic replication where systems try to guarantee that all data among all replicas are identical to each other and maintaining the same state. The idea of optimistic replication is to provide high availability by trading off consistency in replicas. However, this replication is based on eventual consistency, meaning that replicas are guaranteed to converge after a period of time.

The protocol for communicating with the servers can be described in three steps: First, read the status of file from all AVSG to verify that the file is consistent. Second, if the information shows consensus, one of the servers is selected as the preferred server and transfers the data from that server. If this is not the case, servers that need to be updated are notified, the server with most updated data is selected as the preferred server and transfers the file from this server. And finally when a file is updated, Venus communicates with AVSG and the update occurs in two additional steps: the client gener-

ates a stamp (a version vector) of the object (store ID) and updates it to AVSG. Second, Venus transfers the files to the AVSG list who update successfully. Once the data is cached on a Venus (or it is made an update) it is established a callback with the preferred server.

Disconnected operation

Coda uses disconnected operation when the AVGS becomes empty. The disconnection from the servers can be involuntary or voluntary. Involuntary disconnection occurs due to network or server failures and voluntary when a user forces the disconnection, for example, move out of the range of network (cases of wireless networks). While Venus is working in this mode, there is no overhead of multiple replicas or the decrease of performance due to the replication protocol. But the user is only limited to access cached files. When the disconnected operation ends, modified files and directories volumes are propagated to the AVSG and the operation is reverted to normal. Resuming, Venus has three responsibilities: While connected, it must cache files that will be necessary during disconnect. While disconnected, it must use the files cached to serve the file requests; lastly, on reconnection, it has to propagate the updates to the AVSG.

These responsibilities can be represented as states:

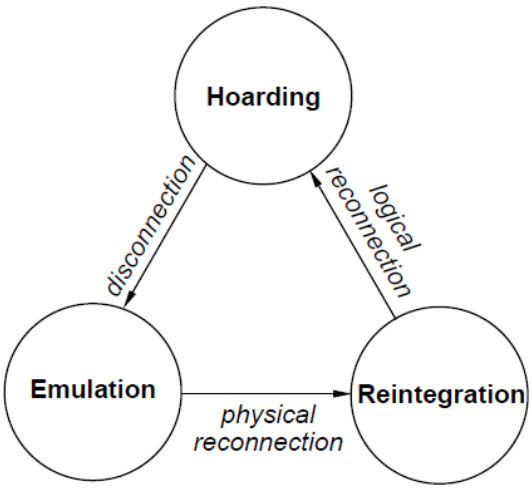

Fig. 6. The various states of Venus.

Hoarding is the state where Venus will spend most of time. It is also the state that represents the connected state, which it serves requests from system calls with AVSG, maintains cache coherence using callbacks, probes AVSG regularly for modifications and more importantly caches files. Files are cached using a normal LRU caching policy. However, it also offers the option to specify a prioritized list of files and directories to retain in cache, called hoard priority list. The highest priority level files and

directories must be retained all the time. Venus uses a prioritized algorithm that balances both the references of LRU and hoard priority list in Venus cache.

When a disconnection occurs, Venus moves to the emulation state. Here it emulates the server locally serving requests using only cached files from the previews state. Of course that if a file it is not in cache, Venus cannot provide that file. In addition it records file operations locally in a structure called the client modify log (CML). Venus has a CML for each volume that it is maintaining in cache. When it is generated an update during disconnection it appends the operation in the log. The objective of CML is upon reconnection Venus should be able to replay the operations at server.

On reconnection, Venus moves to Reintegration state. The objective is to perform operations at CML in servers and then move to hoarding state. Venus starts by processing the CML, for example, removing entries of store records of open files. Then server retrieves CML from the client, write-locks objects that are referenced in log, performs the operations and if all went well, finally transfer files. However if some operations fail or files conflict they are aborted in server side and Venus moves the involved objects to local file called closure. Later these files can be recovered using a repair tool provided by Coda.

2.2.4 Sprite

Sprite network file system [20] was developed at University of California at Berkeley from 1980 until 1992. The goal was to create a distributed file system where each client could read and write to a single and shared file system with performance comparable to traditional local disk file systems. In order to achieve this, Sprite provides Unix file system interface adapted to a distributed environment and unlike other Unix file systems, Sprite was also implemented at the kernel level, as opposed to user-level (like AFS). To improve performance it implements caches, not only on client, but also on server. This FS caches file blocks, instead of the whole files and uses the system memory as cache in its place for the disk. Using the system memory has the main advantage of high access speed, but has a very limited space. So that the system could gather as much memory as possible without interfering with other applications, it implemented a cache size to dynamically grow or shrink. A delayed write policy was implemented so that write operations by clients are made in its cache instead of immediately on server. Then every 30 seconds blocks that have not been modified are written back to server.

A requirement on servers is to control which files clients have open for read and write. While there are no writers, Sprite uses caching on client to achieve high performance. Otherwise, caches on clients are disabled and reads and writes are performed directly on server. When on this scenario only the server cache is used and all write are done with write sharing.

When these cases happen, the system performs poorly, due to his write sharing and network delays. Also Sprite has poor high availability since once servers are unreachable, the system cannot operate.

2.2.5 Hadoop File System

The previous seen distributed file systems are target at specific public: normal users, and for that end, they serve well. But what if we need to store several gigabytes? Such needs have arisen in the scientific and government field, for example in the Large Hadron Collider. This machine it is the world's largest energy particle accelerator, which is expected to address some of the most fundamental questions of physics and human understanding. This project can produce an amount of data corresponding to 100.000 dual-layer DVDs every year [21].

These data intensive computing applications have forced the development of new computing techniques of parallel computation and enhancement computer grinds. An example of a much known successful grid is the one used by Google. This company has a grid of thousands of computers in each warehouse [22] interconnected which can search, store and manipulate huge amounts of data in matter of seconds. This could only be achieved by optimizing the distribution of the workload in the system to a point that it would perform better with the addition of new cores than replace the previous with a better performance cores.

The main evolution of Google over the existing data intensive computing application is its MapReduce model [23]. In this model there are two different phases:

- Map –In this phase the data to be processed is split between the grid nodes that will compute it. The data is split in a form of a key/value pair. This allows processing each pair independently from the others, and therefore it requires minimal coordination between processing nodes. The result is then stored in another key/value pair, as an intermediary result.
- Reduce – This second phase, as in the previous, it is also an assignable task, which can be split between the nodes. The phase consists in read the results stored in the key/value pair on the nodes that were processed in previously and produce the final output.

After the initial success of this model by the Google search engine, there were some attempts to implement this model. One of the most popular implementations is the Hadoop [24]. This system is an open-source application, which is written in Java to support the portability between systems. In the actuality, the model is used widely in commercial and academic applications, for example Amazon, Facebook, Yahoo and others, to support their systems.

Hadoop Architecture

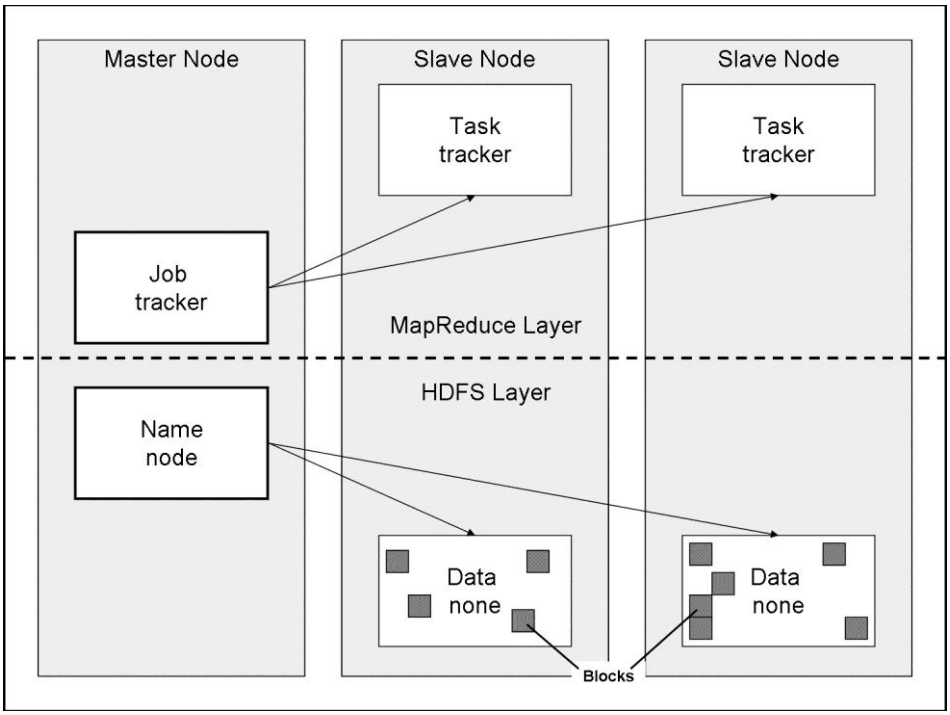

Fig. 7. Hadoop Architecture

The Hadoop architecture can be seen in a higher level of abstraction as a Master-Slave model, where the Master commands the entire operation and assigns tasks, while the Slave is responsible to execute the work that is asked for and report back the progression to the Master. Both, the slave and the master, can be divided in two distinct logical layers: the MapReduce and the Hadoop Distributed File System (HDFS).

MapReduce Layer

It is responsible for implementing the MapReduce model, which is the greatest feature of this system. This model is composed of two services: Job tracker and Task tracker.

The **Job tracker** is located in a dedicated node, in the grid, and it is responsible for monitor the entire operation of the MapReduce. Specifically, it should split the data into pieces to be processed at the slave nodes, schedule, assign and monitor the tasks at each slave node, and finally rerun the tasks that failed.

Each slave node runs a **Task tracker**, which accepts the assigned map, reduces work and executes it. At same time the task tracker will also report periodically to the Job tracker its progression in the task by sending a heartbeat. Additionally it can also request for work, while idle.

Hadoop Distributed File System

Previously, it was mentioned that when the Job tracker assigned tasks, it split the data between the intervening nodes in the operation. In fact, the data it split, but that data can be located anywhere in the HDFS, i.e. in the grid. So, in a map operation, the input data it is retrieved from the HDFS and the reduce operations outputs the results to the HDFS. However, the intermediate results, between the output of map and the input of reduce are not stored in the HDFS. Instead they are stored temporarily in the local file system of the node. In HDFS, files are split in blocks of 64MB and for balancing the accesses, each block corresponding to a file can be scattered across other nodes. Additionally each block is also itself a file. The HDFS is composed by two entities: the Name Node and Data Nodes.

The **Name Node** holds the information about the File System hierarchy (it does not contain any data related to the contents of the files in the HDFS) and, just as the Job tracker, it is only instantiated in the master node. When a file needs to be accessed, the client should contact the Name Node, not only for the authorization of that operation, but also to know which blocks compose the file and at what nodes in the grid are they stored at.

Every slave node packs a Data Node. This entity is responsible to hold blocks of files, which can be generated both from the local or remote slaves. The generation or deletion of blocks is always issued by the Name Node, but when writing or reading a block, it is the slave's HDFS client that manipulates the data directly.

In Hadoop, a read operation, at a higher level of abstraction will be seen as just a regular read from a java stream. However in the background, the stream, instead of reading from the native file system, this operation will be performed by the HDFS Client. First it will try to contact the Name Node to request permission to read the file. If it was granted, it requests the block ids and the list of Data Nodes that have the needed blocks. Then the client will contact the Data Nodes, starting by the nearest one and reads the blocks.

A write operation, as in the previous operation, it will be made through a stream and therefore it is transparent to the user. In the background the HDFS client will split the data to write into blocks of 64MB. This system attains replication by storing multiple copies of the same block in several Data Nodes. By default the HDFS makes three copies of the same block: two in the same rack and another in a different rack. This replication system works as follows: the client sends the data to a Data Node and that Data Node also sends the same data to another, and it goes on until the predefined number of copies has been reached. When all the data has been transferred successfully to all Data Nodes, the write operation finishes. This operation is double checked by the Name Node when the Data Node sends the list of updated blocks to it. If there are not enough copies of a block the Name Node issues its replication.

Although this algorithm is widely known, there have been some problems namely, Facebook [25]. In the scope of the project being developed, this algorithm is specific to one problem: massive computation over large amounts of data. While my project can require large storage, depending on the amount

of users, Hadoop requires specifically high bandwidth, which is not available in my project. Also it does not support the offline operation in the case of Data Nodes are not available. For this reason, Hadoop might still be used as a source for ideas, but not as part of the solution.

2.3 Could Computing

Previously we have seen HDFS. This distributed file system uses dedicated clusters of computers to manipulate huge amounts of data. While some large companies can afford to buy these costly infrastructures, it is expensive to maintain them, especially when all resources are not at 100% usage. On the other side, small companies do not have money to invest in such infrastructures, not only because of the initial investment, but also the maintenance. From the combination of both needs it was born the cloud computing concept.

The cloud computing is the result of the formation of computer clusters with the main purpose of providing a large amount of services in a virtualized way. As it is possible to maximize the usage of computation resources by renting the unused resources and reduce the costs. The small companies or users also benefit as they only pay for the resources that they are using.

The virtualization is one of the key features, as it allows users to access the cloud resources and see them as its own. This isolation is achieved with the usage of virtual machines (VM), as they allow a better control over the cloud resources, like CPU, network, storage… Also at same time it is possible to provide elasticity, as the limitations imposed of the access to the resources in VMs can be changed to match the user needs. Another advantage of the isolation provided is that in a case of a failure by a VM, it will not affect the remaining VMs running in the same physical infrastructure.

Therefore, the cloud resources offer great elasticity as they are provided on-demand and scale with the needs of the user. These services are becoming more and more popular due to the flexibility offered and the simplicity of access to data from anywhere. The costs to the user only depend on the amount of consumed resources in the cloud.

The services provided by the cloud can vary in functionality and size. While some clouds can focus only on services to storage data, there are others that offer more general services or a wider range of personalization. For this reason, services provided by the cloud are divided in layers [26]:

- Software as a Service (SaaS) – This layer can be thought as the higher layer of cloud computing. The user it is only allowed to use the provider's applications from various interface clients, but cannot manage or control the underlying cloud infrastructure.
- Platform as a Service (PaaS) – The PaaS layer it is more permissive. It allows the user to deploy its own application in the cloud using the provider's tools, libraries and services. Again the user cannot manage or control the underlying cloud infrastructure, but he has direct control over deployed applications.

- Infrastructure as a Service (IaaS) – This final layer it is the most permissive. Here the user can deploy and run arbitrary software. He can take control of operating system, storage, deployed applications and network components. Still, he cannot manage or control the underlying cloud infrastructure.

While cloud computing has numerous advantages, most of them already mention above, there is still an important issue: security. The idea of cloud computing it is a total abstraction of the underlying cloud infrastructure from the user. The user can use the cloud resources, but he does not know where the physical resources are neither where his data is stored, and more importantly, if his data is safe. For example there were some situations: phishing attack on Salesforce.com which duped an employees and customers into revealing passwords [27] and Dropbox successful attacks [4] [28]. Another important security aspect when using cloud storage it that cloud services usually work with third parties, so the customers, not only have to worry about the possible compromise of data in the cloud service that they hired, but also from the third parties, that they did not even know it was involved [29].

2.3.1 Amazon Web Services (AWS)

The AWS it is a set of services that offer some cloud based services, being the most popular the Amazon Simple Storage Service (Amazon S3) and Elastic Computing Cloud (EC2). These two services are very distinct: while Amazon S3 provides a storage cloud service, the EC2 it is more oriented to cloud computing. Still, they both share a common cloud characteristic: they offer the elasticity on-demand. So that when the user needs more space or more computing resources, the resource pool it is automatically enlarged.

The Amazon S3 [30] provides web interfaces to allow users to store and retrieve any amount of data on the internet. The data is organized in objects that can be written, read or deleted and contain up to 5 terabytes of data. More explicitly, each object it a binary object. Each of these objects is identified by a unique key and a bucket. The bucket is a set of keys where each unique key has an associated value, the object.

To store or retrieve the objects, Amazon provides a set of standards such as REST and SOAP interfaces, but it also possible to build additional protocol layers, as the Bit torrent protocol.

The S3 service it is powered up by, among other Amazon technologies, the DynamoDB [31]. This technology was developed by the Amazon itself as a “highly available key-value storage system”, which manages the trade-off between cost, consistency, durability and performance, while maintaining high availability. The DynamoDB it is based on NoSQL, which it is an alternative to the limitations of relational databases [32].

2.3.2 Dropbox

Dropbox is a popular [33] internet file hosting service which offers cloud based storage to store, retrieve and share data among users. The idea was to create an application that could share the same files across multiple desktop computers and laptops. At that time, the idea was not completely new, but the existent products had problems with internet latency, bugs or they were too complex to use. To solve this problem, Drew Houston created a demo of Dropbox and with the help of a fellow, Arash Ferdowsi, they created Dropbox [33] [34].

What makes this cloud storage service so popular is the client application which combines a set of unusual features that did not exists at that time: automatic synchronization, versioning of files, delta encoding and web-interface [35].

File Synchronization

Dropbox supports two ways to access the user own workspace: by the web interface and client application. The web interface allows the user to make simple operations, like download, upload and rename a file. While the user is uploading the file, he has to stay in the same webpage until the upload is complete.

The second way requires a user to install the Dropbox client who will autonomously synchronize all file and folders present in a special directory. What makes this directory special is that it will mirror all the server side data (so that all the data present in the server side will also be present in that directory) and if any of that data it is changed, that change will also be performed on the server side. This way, Dropbox allows the data to be modified offline and resynchronized later. In addition, if the file synchronization involves shared data among users or other machines, their clients will also update the corresponding data to the most updated copy.

As mentioned before, one of the bottlenecks of distributed file systems it is the network bandwidth. Therefore, it is essential to optimize the usage of this resource. The Dropbox client deals with this limitation by optimizing both the data transferences and data storage. When a file it is updated, the client will analyze the data and compare it with the remote copy on the server side, if there is one. The comparing algorithm used it is unknown, as it is a closed source product, but some users tests point to a mix usage of a deduplication algorithm [36] [37] [38] and “binary diff” [39]. When uploading a file, the client will send the hashes of the file and try to find a matching hash within the already indexed hashes. If it finds any matching parts of the data, those parts are not transferred. Else, it will execute a “binary diff” which will mark the portions of the data that have been modified and only those changes are transferred to the server side allowing this way to keep the low bandwidth usage, especially in large files.

Dropbox also employs another tool to remove load from the main storage: the Lan sync [40]. When this tool is enabled, it allows a user sharing a data with another user, in the same network, to retrieve data updates directly from other user, without the needing of get it from the Dropbox servers.

As Dropbox has a very permissive system of automatic synchronization, it might leave some space for the user to make mistakes, wherever he is updating or deleting a file. For this reason it incorporates a versioning system, which allows users to keep multiple versions of a file as a backup. Also in a case of a “rollback”, it is possible to save bandwidth, just by sending back to the user the modifications with binary compression.

In order to run this whole cloud system, Dropbox makes use of other cloud services to offer its own cloud services, namely Amazon EC2 and Amazon S3 cloud services [30]. EC2 is used to run the server applications, as it is a cloud computing service, and Amazon S3 serves as their primary data storage. The combination of these cloud services provides a very complete solution system and which is now a reference to other companies [41].

2.3.3 Google Drive

Since the appearance of Dropbox and given its success, many other cloud based file hosting arose. Google Drive [42] [43] is the Google’s file storage and synchronization service which was released at 24 April 2012. It offers cloud storage, file sharing and collaborative tools. Just as Dropbox, Google offers a web interface which allows the user to upload and download files and as an alternative also provide a client which synchronizes the files autonomously. However, Google Drive’s client does not support any kind of binary compression as it was pointed out by Nicolas Garnier (Nivco) [44]. This way, if a there was a change in files Google re-uploads the entire file, instead of just the changes within the file. The same thing happens in versioning, instead of just saving the deltas of each version to spare space, it keeps the entire file.

The main advantage of Google Drive is the integration with the service Google Docs. This service is a web-based office suite which allows users to create and edit documents online concurrently with other user, which is particularly useful when more than one coworker is writing in the same document. Google Docs also includes other features such a powerful search which allows the user to search for text or images and a viewer in the browser that allows to open over 30 file types in the browser including HD video, Adobe Illustrator and Photoshop.

2.4 Efficient data transfer

In this section we will discuss some methods that can be used to reduce data storage and network transmission bandwidth. However, this comes at cost of some additional processing. The great benefits of eliminating extra storage space and transmission overhead are the increased scalability of the system, which is imperative in distributed file systems due to the large numbers of clients compared to servers.

2.4.1 Compression

Compression techniques can be divided in two groups: lossless and lossy. Lossy compression algorithms can be found in audio, video and image compression. They are used in such objects, because the main goal is not to reproduce the exact copy, but a way to obtain the best fidelity for a given amount of compression. However in the context of this thesis, the object has to be reproduced exactly as the original once decompressed. The algorithms used for such compression are called lossless. They can be divided in two categories: dictionary based methods and statistical based methods.

2.4.1.1 LZW

LZW [45] (Lempel-Ziv-Welch) method is a lossless data algorithm which is an example of a dictionary based approach to compression. The original idea was created by Abraham Lempel and Jacob Ziv, and was first published in 1977 called LZ77. The next year they published a new algorithm called LZ78. Later, Terry Welch made some refinements to LZ78 and published it in 1984 by the name LZW. This algorithm compression is based on replacement of strings by a single code. These strings organized in a dictionary and every string is a reference to a LZW code word. The dictionary is initialized with an entry for every byte (characters of 8-bit) and the outputted as fixed-length 12-bit codes. Subsequent strings are built from the input stream. The code word that will reference the string is given by the next available code number and the entry is added in the dictionary. Compression occurs when a single code is output instead of the corresponding string. Since the combinations in one byte are 256, the first 256 codes are assigned by default to every byte. The remaining codes are assigned as the dictionary is constructed, up to 4095.

Encoding/decoding

The input stream reads 1 byte at time and at each step of the encoding process input byte is concatenate in strings until it finds a combination that is not present in dictionary. Then LZW adds the new string to dictionary; generates the new code word and outputs it. The string that will be tested in next step will be the last read bye concatenating the cycle's byte. The decoding process is the inverse operation, however instead of read one byte; 12-bit codes are read. The dictionary is initialized, as previously, only with the 256 entries and is constructed as the algorithm proceeds.

2.4.1.2 Huffman

Huffman coding [46] is another lossless algorithm, but it is an example of a statistical based approach. The algorithm was designed by David Huffman while he was still a student at MIT in 1950. This algorithm is used nowadays mostly as a complement to other compression algorithms such as GZIP, JPEG and many other utilities. In this algorithm for each symbol in the text that will be com-

pressed it is assigned a prefix code. This code is a sequence of bits which is different for each symbol. The length of the sequence is based on the probability of the symbols, i.e. shorter for most common symbols and larger for less common. To calculate the probability it is built a binary tree where for each node it is assigned a weight and the sum of a node sibling's weight is equal to the weight of parent node. To obtain the prefix code of a symbol, the tree is searched top-down and the code it is the sequence we used until reach that symbol. The most common symbols are shallower in the tree, than the less common, resulting in a shorter sequence.

Example for "this is an example of a Huffman tree":

Character	Frequency	Code
space	7	111
e	4	000
a	4	010
f	3	1101
t	2	0110
s	2	1011
h	2	1010
i	2	1000
n	2	0010
m	2	0111
o	1	00110
l	1	11001
x	1	10010
u	1	00111
p	1	10011
r	1	11000

2.4.2 Chunks and Hashing

An alternative to compression is exploiting the similarities of different versions the same of file. The idea comes from the fact that usually a file does not change completely between two versions. By exploring these similarities it is possible to save time and bandwidth, since those parts are not needed to be transferred. Usually this is accomplished by dividing a file in chunks, or fragments, and sending only the new or modified chunks over the network.

2.4.2.1 Rsync

Rsync [47] [48] is a software application and network protocol that synchronizes files and directories while minimizing data transferences using delta encoding. Delta encoding it is a form of representing the differences between two files rather than the complete files. The original Rsync was designed by Andrew Tridgell and Paul Mackerras and it was first announced in 1996. Let's consider two files A and B which are "similar". The Rsync algorithm proceeds as follows: First, the receiver splits file B into non-overlapping, contiguous and fixed-sized blocks of size S bytes. Of course, that the last block can be smaller than size S. Then for each of those blocks Rsync calculates two checksums: a weak "rolling" 32-bit checksum and a strong 128-bit MD4 checksum (the new version 30 of Rsync uses MD5, instead of MD4). These checksums are sent to sender. Next, it computes all overlapping blocks of the file A and searches for one block that matches both hashes. This can be done in just a single pass due to a special property of rolling checksums. Finally the sender sends only the data of sections that were not found and instruction to the receiver rebuild file B to a copy of A.

2.4.2.2 CVS

Concurrent Versions System (CVS) [49] is a software that keeps track of all changes in a set of files and allows several developers to collaborate. The transmission it is also done through delta encoding as well for storing the differences between files.

2.4.2.3 LBFS

LBFS is a Low-Bandwidth Network File System [50] that is designed for networks with low bandwidth and high latencies. Like Rsync it exploits the similarities between files, however it goes a bit further and also exploits the similarities between different files (for example temporary or auto-save files which are usually used by text editors). This technique used together with techniques of other distributed file systems, such as Coda, can decrease substantially the amount of data transferred by avoiding transfer duplicated data that data is already present on client's cache. LBFS uses a large cache at client and server. Client as well as server divide files into chunks and index them by calculating their hash and storing them on a local database. The chunks are of variable size and theirs boundaries are determined by Rabin fingerprints. This technique ensures that any modification to the content in a block will affect that block and not the boundaries of the neighbor blocks. LBFS indexes files using SHA-1 hash which is an extremely low probability of collision hash and assumes that if two blocks have the same hash, then those blocks are equal. So in a transmission, which is done block by block, if the block has the same hash, only the hash is sent, otherwise the entire block is transmitted. The data sent is compressed using Gzip. Just like AFS, files are flushed to server when they are closed. In

order to ensure that writes never result in inconsistent files, LBFS uses temporary files which are then renamed.

2.4.2.4 DFS (Microsoft)

The first appearance of Microsoft DFS [51] [52] was in version 4.1 integrated with Windows NT 4.0. This system allowed administrators to join resources present in different servers into a single namespace as a hierarchical file system [53].

So if a company had 5 servers, each sharing its resources through the UNC:

- \\server1\aplications
- \\server2\payrolls
- \\server3\commons
- \\server4\users
- \\server5\folders

The DFS could add an abstraction layer to the physical network points by combining them in unique a DFS namespace (directory tree). Each server would be represented as directory in the namespace. This way the users just needed to remember the name of the main server (the DFS root) to access all others. The resources could be browsed with the Windows Explorer, just as if the resources were on the local storage.

But this system had a huge flaw: the DFS was not fault tolerant. So if the DFS root server went offline there was no way to access the DFS, unless the users knew the specific location each resource.

This feature was added in the DFS 5 with windows 2000 Server and allowed to replicate the DFS namespace in multiple servers.

Another feature added was Windows File Replication Service (FRS). It enabled the replication of the shared resources in the DFS. This way each resource of the DFS namespace could point to multiple servers, but it will still be present to the user as one resource. The FRS detects when a file was created, modified or deleted and replicates it to other servers that also shared that resource. If there was a conflict i.e. if both files were modified, the file with the most recent date and hour would prevail.

Later in Windows Server 2003, the FRS was enhanced by the DFS Replication (DFSR). The main advantage of this improvement is that, while previously the entire file was transferred if changed, this new file synchronization service aims to reduce the amount of used bandwidth to transfer a file by transferring only the differences between them. The Remote Differential Compression (RDC) [54], as this method was named, is different from the Binary Delta Compression, which is designed to operate only on known version of a single file. The RDC computes the file differences on the fly, so there are no assumptions about both file differences. To compute the differences it divides the file in blocks of variable size. The block size is determined per block by computing the local maxima of the block using a fingerprinting function. This function it is a rolling hash function that will be computed incrementally over the block. When it reaches the local maxima, the current byte position is chosen as a cut block

boundary. After the division of blocks, for each one it is computed a stronger hash (MD4 hash). The signatures can then be used to compare the contents of another file. The matching ones are assumed to be the same blocks and therefore, they do not need to be transferred again. As the file containing signatures grows linearly with the file size, if the original file size is huge, it can also generate a huge signature file. For this reason, the RDC algorithm can also be applied to signature file to reduce even further its size. Microsoft estimates that "if the original file size is 9 GB, the signature file size would typically be about 81 MB". And if the RDC is applied again over that signature file, that size would be reduced to 5.7 MB [54].

A recent update to windows server 2012 introduced even another improvement to storage services. Data deduplication [55] aims at finding duplicated file blocks and replaces them with a reference to a single copy of the block. To apply this algorithm the blocks are divided in variable size blocks, between 32 and 128KB, analyzed, as described above, and the blocks of a file are reorganized into special container files in the System Volume Information folder. This method allows saving a file with same contents but at reduced size, but as in the previous algorithm, at expense of CPU usage.

Binary Delta Compression is a method which is ideal when trying to find differences in two files. By having both the original and the new files, it can compute the differences between them. While this method seems to be ideal to my project, because by transferring only the differences between files it would allow to save a lot of bandwidth, there is a problem. Both the original file and the new are in different places of the network, so it automatically invalidates the usage of this method, because it would require the transference of the new file to be analyzed. For the same reason, this method was not used in the RDC. Still, the RDC is a viable option to reduce the bandwidth usage in file transferences, but must be kept in mind that the saves in bandwidth come at cost of CPU usage.

2.5 Summary

This chapter presented the existing solutions and introduces the main systems and ideas that were relevant in the conception of the main solution. It starts by introducing the concept of distributed file systems and relevant systems that use this concept: NFS, AFS, Coda, Sprite and Hadoop File System. They are followed by an explanation of the popular cloud computing concept and some known technologies in the field: Amazon Web Services, Dropbox and Google Drive.

Finally was presented the "Efficient data transfer" sub-chapter, which explains two distinct ways to achieve compression when transferring a file: Compression and Chinks and Hashing, along with the existing technologies.

3 Architecture

After studying and reviewing the algorithms and tools related with this thesis, this chapter brings the description and presents the solution developed, as well the main problems that appeared during the development process.

The next section (3.1) will present a quick overview of the system, while the subsequent will describe the modules used that together form the final solution.

3.1 Architecture overview

This chapter is organized in the following contents:

- Section 3.2 describes the Authentication Module. This module is the link between the client and the user browser (which is used for authentication purposes).
- Section 3.3 describes the Cache, which is mainly used for storing remote and local data.
- The section 3.4 provides the description of File System Watcher component. It is responsible for monitor and capture events that occurred in the Cache of the local file system.
- Those events are treated by the Event Queue and stored until they are ready to send to the repository. This module is described in section 3.5.
- Finally, we have the Client Dispatcher which manages all the communications with repository and decide the resulting measures to be taken over the files. As this is a complex module it is divided in 3 sub-components: Synchronization Resolver at section 3.6, Delta encoder/decoder at section 3.7 and Repository Connector at 3.8.

The objective of this work is to develop and implement BennuFC. It is a client which can synchronize autonomously with the Bennu, the IST repository. The actual working system of Bennu offers an interface to the repository through a web portal. However, it is not capable of autonomously synchronize the files present in the user workspace and the repository. The point of the client is to ease this process by a complete separation of duties, in such way that, the user can work freely in his workspace, while the client takes care of the whole process of managing updated files either from repository to the workspace or otherwise.

Furthermore, whenever a change is made in a file, it implies sending the entire contents of the file even if the change was small. This work also aims to develop a method to synchronize files by exploiting the similarities between different versions of the file. If a file was modified, why not just transfer the new contents? Through this method is possible to achieve high compression rates in transferences, as only the changed parts are transferred.

The system is divided into modules where each one plays a different role. The sections presented in this chapter follow the flow since when the client detects a modification of a file in the local user workspace until it reaches the repository. The overall architecture is depicted on figure 8.

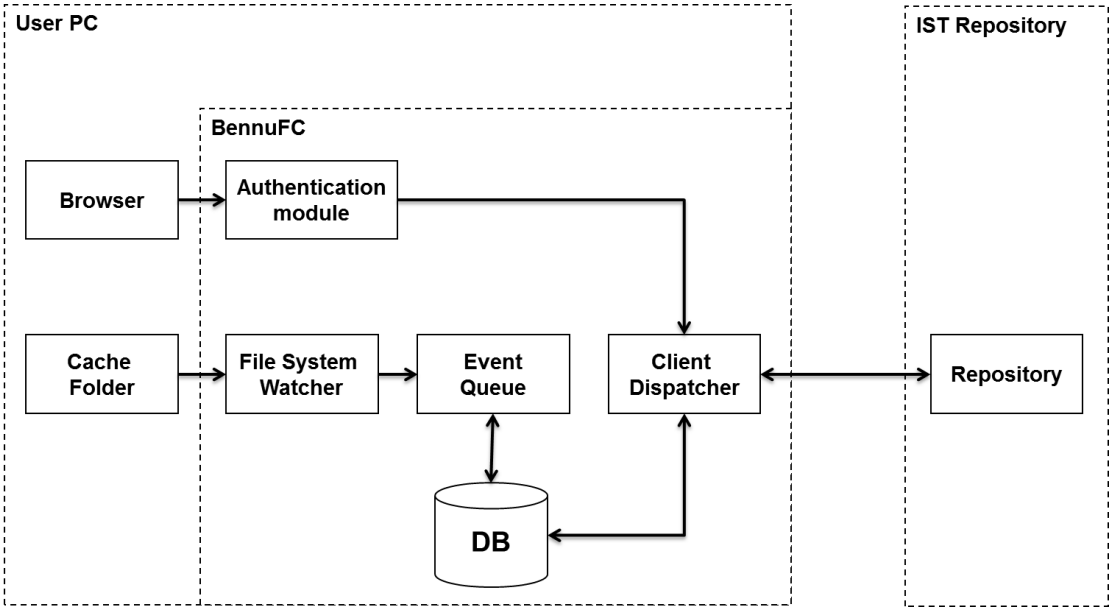

Fig. 8. Architecture of the system

The cache folder is a regular folder in the user's PC, where the user can put files or other folders with the objective of synchronize that data with the online repository.

This folder is being permanently monitored by another component, the File System Watcher (FSW). So any change made by the user at the cache folder will notify the FSW. As the notification system only works when the client is running, FSW also implements a method to know if there was any change since the last time it ran.

When the FSW detects any change it will send the event to the Event Queue. This component is mostly responsible to receive notifications about changes in the cache and store them in a persistent way using a database.

As soon as the Event Queue starts receiving events, the client dispatcher will retrieve them in order to send them to the repository. The ability to communicate with it will depend on two factors: If the client is able to communicate with the online repository and if the user is authenticated. The authentication is attained by using the browser which will communicate with the client's authentication module by delivering him a CAS authentication ticket. Then the authentication module validates it with the repository and if it is valid, the user is authenticated. From this point, the client dispatcher is able to freely communicate with the repository and is able to upload files to the repository or download updated files to the user's cache folder.

3.2 Authentication module

One of the requirements of this project was to use the same authentication services that IST provides. The current framework supports Kerberos and a single sign-on solution based on CAS with an authenticated backend based on LDAP and Kerberos.

Kerberos is a widely used computer network authentication protocol. This option could be in practice used and in terms of implementation would be simpler: the user would input his credentials in client, the application sends them to the IST repository and it would authenticate the user. But as the user authentication credentials are provided by IST, it has strict rules about security and tries to avoid solutions that require the user to input both username and password directly in a 3rd party programs. For this reason, Kerberos should not be used in this context.

Finally we have the CAS service. The details of this service have been explained previously in the related work chapter, particularly its single sign-on feature, which allows a great integration with all IST web application. Also, this authentication method is widely used in authentication of the users for IST web applications and, for this reason, the algorithm's implementation is already mature enough and considered safe. Still, so far this protocol has only been used in the authentication between the user's web browser and IST web applications. The coupling of the CAS system with a client application could be done easily with the already existing libraries, but just like the Kerberos, the credentials could not be used on 3rd party application, other than the user browser. The main problem here is that if the user performs the login through the browser, the client application does not have any mean to communicate with it and therefore, it cannot access the CAS ticket which would allow the client to authenticate itself.

The solution was found with a method similar to the one used in magnet links of torrent clients. When the browser founds an URL protocol that it cannot handle, it will search on the OS for a suitable application. In the Windows, the user browser will search on the Windows Registry for a previously registered application associated with that protocol. If it finds one, that application is executed using the browser link as the argument. In this case, when the browser opens a link starting by "bennufc://" protocol, the browser will look for a suitable application registered under that protocol and executed the client using the CAS ticket as the argument. Once the client has the CAS ticket, it completes the rest of the CAS authentication protocol. This way the client never has access to the user credentials, but it was able to authenticate itself using the CAS ticket.

3.3 Cache

One of the requirements of the client is to ease the access to the remote repository files, even if it is not reachable. The solution to this problem has been studied in some of previously mentioned File Systems. They cached files of the remote repository locally so that when the client is disconnected from the Internet, files are still available.

In computation a cache is used as faster memory that stores the data transparently so that future requests are served faster. Generally, the data stored in caches can be data previously computed or duplicated data from somewhere else, for example, remotely. When there is a data request and that data is present in cache, the data is retrieved immediately from the cache, which results in a faster access time. On the other side, if the requested data it is not present, there is a cache miss and that data has to be retrieved from its original source or recomputed.

The same concept can also be used in remote communications. If the requested data is available locally, then it can be retrieved immediately without further communication with the repository. Otherwise the data has to be retrieved remotely, which increases its access time.

The cache concept used in this project is very simple: Instead of creating an additional entity to manage data, the cache is just a regular folder of the OS created in the user's computer which can contain other folders and files. The cache's files can be read or written by the user's applications or by the repository, through the client application. This way it is possible to take advantage of OS functionalities and features, such as search and OS interfaces to access the cache, which are more familiar to the user.

The success of this component is, however, highly related to the available storage space. The larger the information we hold in cache, the faster it is its access, but it comes at cost of storage space. Some of the DFS studied, such Coda, used a variable cache size due to constraint in storage space at that time. This lead to the development of complex hoarding algorithms which sometimes still lead to frequent cache misses. While it made sense those days because of the high storage space prices, today most of computers have large storage space, which will be more than enough to keep all files in the cache. For example, Dropbox allows a user to either store the entire repository or just some folders.

The major advantage of this decision is that when the client is unable to communicate with the server, the user can still access to all files he had stored on his remote space, because they were saved previously on the cache.

3.4 File System Watcher

While the cache on its own can bring huge improvements in access times, it does not suit us well if the coherence between the local cache and the remote repository is not maintained. For this reason it was created the File System Watcher. This component is responsible for the connection between the local cache and the client. When a user makes modifications on his workspace, i.e., in the local cache, it is important that the application is aware of such modifications to take further measures. To solve this problem, it is necessary that the client monitors the cache's folder to look for any changes that might occur. There are two methodologies to work out the issue: an extensive file search method or register the application to receive directory notifications.

The first approach is based on a constant search in the directory (or directories) checking every file for any modifications or a modified attribute. When it finishes the search, it starts over. Despite being an easy method to implement, it is very resource demanding and decreases the system performance because of its exhausting search.

The second solution is to register the application to receive modification notifications of a directory. This tactic is much more efficient than the first method since it relies on OS notifications [56]. The theory behind this method is the capability of File Systems (FS) to register modifications in the so called Journals [57].

For example, in Microsoft NTFS [58], as files and directories are added, deleted or modified the file system writes change journals records in streams. Thus, it is possible for applications that previously registered for directories or files notifications to receive these events. Another benefit of these journals, which is its actual main purpose, is to log modifications in file systems before the OS commit them to the main file system. So that in case of a power failure or a system crash, file systems recover quickly and are less likely to become corrupted. However, in terms of receiving the notifications, this method has a major downside. If the application is not running, it will not receive the events. So in the context of my solution, just relying on this method means some events would not be caught especially when the client crashes or if any events happen before the application is started. It is important to note that the journaling system it is not specific of NTFS and exists in other file systems, such as Linux ext3 [59] or Apple HFS Plus [60].

The resulting solution is a hybrid algorithm of both methods: directory monitoring and extensive file search. When the client starts, it registers the application to receive notifications of the cache directory. After this point any notifications made in files, folders or sub-folders will be received by the client. The notifications caught are sent to the Event Queue. This algorithm will continue its execution until the program finishes.

However, since the client cannot receive notifications that occurred before it was started, it will run a polling phase at startup. During this phase, it will scan the cache directory, and sub-directories, for any previous changes using the extensive file search method. In practice this method will check the previous last modified date that the client knows of with the actual “last modified date” of the file. If changes are found, File System Watcher will generate the corresponding events to those changes and sent to the Event Queue to be processed. Once the search finishes, the polling phase ends, but the monitor remains active.

The notifications received can be of three types: creation, modification and deletion. Each of these events corresponds to an operation made in a file or directory.

3.5 Event Queue

This component is responsible for keeping the persistent state of the client along the various executions. To preserve the state of the application we used a local database. Its objective is to mirror the

local file system, but with additional metadata information about synchronization state of each file and directory. Although that information could be stored in hidden files, just like CVS or SVN does, we wanted to keep the user workspace as cleaner as possible. In order to keep the database updated and reflect the changes in the existing local file system, as Event Queue receives events generated by the File System Watcher, they are processed and stored in the database. Additionally, it also stores additional information about the directories and files. So, the exact objective of the database is to keep the same structure and organization as the cache, without the actual contents, but storing additional metadata information about the cache entities (files and folders). The stored supplementary information falls in the following categories:

Paths

The information kept in this category is related to folders existing in the user's workspace. Every time the event corresponding to folder being created or deleted is received, that information is reflected on the database. Besides registering it, it also provides a faster way to rename and index folders and associate them to the files.

Files

In the same way as paths, when a file is created, modified or deleted physically, that update is reflected on the database. Additionally to that information, it also keeps the information about the date of the last modification and version number.

The date allows the client to know when it saw the last modification on that particular file. The main use of this information is during the polling phase of File System Watcher, when the client is not running, but the user can modify any file in the cache. When the polling phase runs, it will compare the date of the last modification in the database with the one present on the file, managed by the OS File System. If they are different, then the file was modified. Also, if a file exists on the database and not on the cache, it means that the file was deleted. In the same way, if the file is present on the cache, but not on the database, then that file was created. For each of these cases, an event is created, just as if it came from the FSW.

The version number of a file is used in file transferences and to detect conflicts. This number is only updated by the repository, which means that, if a file is modified in the cache, the version number will remain unchanged. When the change is committed with success to the repository, the version will change to the one returned by the repository.

Events

As the File System Watcher captures events, they are sent to the Event Queue. Once they arrive, they will be stored in the memory to be processed. If the application crashes with remaining events to be processed, they are lost. But as the File System watcher will always run the polling phase when the client is started, it will recreate those missing events.

The events sent to the Event Queue can be of three types: create, modify and delete. When those events are processed, the database is updated with the corresponding actions to keep the database cache structure identical to the cache itself. For example, if there is a delete in the cache, the reference to that file in the cache is also deleted. But as the client needs to keep the remote repository synchronized with the local workspace, it will be also created an event to send to the repository, notifying of this action. These events, unlike local events (which are processed first), are stored directly in the database. If the client has connectivity to the server, the Client Dispatcher will try to send the event to the remote repository, otherwise it will just keep it on the database, until there is connectivity. More details on this algorithm are explained in section 3.6.

Besides these events, there is also another type of event. Just as FSW, the remote repository can also send events to the client. This occurs if the remote repository suffers a modification of any type. For example, when the remote repository sends an event of a modified file to the client, that file will, in perfect conditions, overwrite the cache's file with the most updated file. But if that file is in use by the user, the client cannot overwrite it. So to prevent skipping the event, it is saved in the database and from time to time, the client will try to perform the event. This event will only be erased in two cases: if the event is executed with success or if the file is deleted remotely or locally.

3.6 Client Dispatcher

The client dispatcher is the responsible for the connection between the client and the repository. The whole process of file synchronization involves several stages and in a higher level of abstraction it can be divided in two main steps.

The first is to decide which files should be transferred from and to the repository. This is decided by retrieving the events of modified files and folders from the repository and crossing that information with the local changes previously stored on the database. The result of this stage is a unified list of operations either from or to the remote repository.

The final stage is all about executing the operations that were fetched in the first stage and transfer the files needed to and from the repository. The transfer of files is made using file encoding, which is built on delta encoding. The idea is that files with same name and in the same folder can have similar contents. To take advantage of this similarity, the encoder analyzes both files and transfers only the differences between them. The encoding process will be explained in detail in section 3.6.2.

This Client Dispatcher is constituted by three sub-components: the synchronization resolver, delta-encoder and repository connector. The synchronization resolver is responsible for the first stage, while the last two for the second stage. In the following sub-chapter they will be explained in detail.

3.6.1 Synchronization resolver

The synchronization resolver is responsible for the first state of synchronization between the client and the repository. As previously stated, this stage uses two lists of events which are based on events that happened locally and remotely. The local operations are present in the local database, which have information about the file name, path name and version. When this list is retrieved, it contains the history of events that were observed in the local file system and, for each file it can contains more than one entry. Also some operations, like a modification of a file, can generate more than one “modify operation”. When this list is retrieved, it needs to be parsed, in order to eliminate duplicated entries and get the final operation for each file. The final operation is the resulting operation by following the ordered operations that are in the list for file. For example, if a file was created, modified and then deleted, the final operation will be deleted. If a file was modified, deleted and then created, the final operation is created. A file is identified uniquely by its path name and file name.

The remote list is also based on the history of the events, but this list contains the full history of events. So in order to do not process the same events over and over again, when the client request for updates, it also sends the timestamp of the last update that was seen by the client. This value is retrieved from the database and represents the timestamp of the most recent event present in the previous request. Like so, the repository will only send the events that occurred past that time.

In the next figure will be showed an example of the polling process in the repository with various requests:

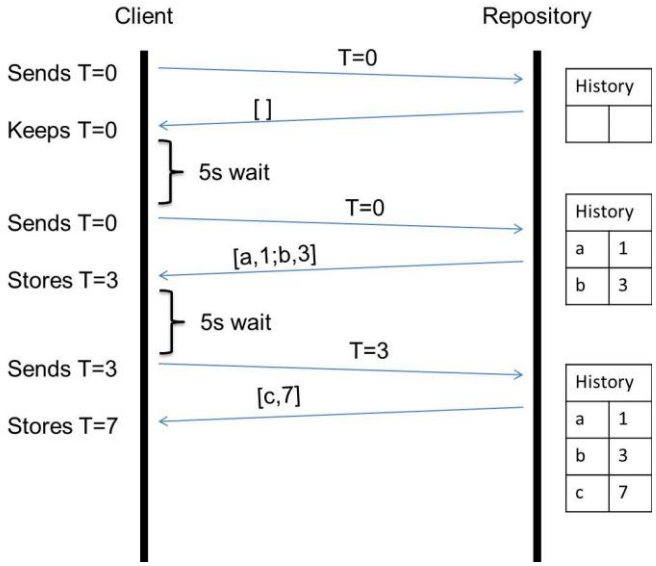

Fig. 9. Retrieval of updates from repository

When the client starts its update cycle it will poll the repository for updates and sends the last update timestamp. If the client never updated anything, it will send 0; otherwise it will send the stored value. In this case as the client did not have any previous contact with repository, it will poll with a timestamp (T) of 0. The repository checks its local history of events for any updates that happened after that value. Since the repository history was empty, there were no updates on repository. So, the repository replies with an empty list of events. The client checks the list, but as it is empty, the client keeps T with its value. Before each poll the client will make a wait period of 5 seconds, so that it does not overload the repository.

In the subsequent poll the client will send again the last known timestamp. The repository will check the history list and select which events happened since timestamp 0. In this case, there were two files created at time 1 and 3 and will reply with a list of files with the corresponding timestamps, version, file name, folder name and the event which occurred over that file. Just as in the client, the events can be of three types: creation, modification or deletion. Upon receiving the list, the client will parse the information and extract the highest timestamp. This timestamp is stored to be sent in the next poll.

Similarly, in the next subsequent polls the client will send the last highest timestamp seen to the repository, which will return the events in history posterior to that timestamp.

After these polls the client has the list of events that occurred remotely. The next step is to merge this list with the local event list. This is made by executing the synchronization protocol to resolve the possible existing conflicts and create a unified list of final operations to perform locally and remotely.

Synchronization protocol and conflicts

After getting both lists of events from local and remote workspace the synchronization is simple. If there are any operations in the remote list, then they must be applied locally. In the same way, operations in the local list are applied remotely. For example, if in the local list there is information about the creation of “File1.txt” and in the remote list about the creation of “File2.txt”, the synchronization will consist in uploading “File1.txt” to the repository and download “File2.txt” to the local cache. Update operations will be equivalent to the creation of a file which results in the upload or download of the file. Finally, the delete operation erases the file from the local cache, while if it is performed in the repository, it will erase the reference to the file, but keeping the file contents in the case of the user wants to restore it.

Also when there is an update to a file coming from the repository, it is important to make sure that versions already present in the client are not transferred again from the repository. This can happen because when a client uploads a file to the repository, the repository registers it in its history with the actual timestamp. This registry will have a greater timestamp than the last time that the client checked for the events. So when the client requests again the remote list, it will see its operations in the event list. If the client finds out that the local copy of that file was not modified and that the version in the remote list matches the local file version, then the operation on the remote list was just an echo of a previous operation performed by the same client and ignores it.

The Synchronization is as simple as explained above when the same file is not referenced in both lists. But in this case there is a conflict. An example of a conflict is when the local and remote files have been modified. Usually this happens when a file, which is being shared with several users, is modified by more than one user, at same time. The conflict is identified when the client retrieves the remote event list and tries to merge it with the local list. As the same file was modified locally and remotely, the client will solve this conflict by duplicating the local file and renaming it with a prefix plus the filename. The file is backed up in the same folder and from now on, it is a normal file, just as if it was created by the local user. Then, the remote copy is downloaded and replaces the user file. This way, it is kept the consistency with the repository and the user changes. The user can later review the backed up file and merge its contents with the new file.

Although this solution works, in practice it may not be always available. Consider again the same case where both the local and the remote have modified the file, but the local user is still doing changes in the file. In this case, the client will never be able to acquire the lock for the file. So it is not able to duplicate the file. The way around is to locally log this update and try to reapply it in the future. Every time that the client starts a new update cycle, it checks for any of these failed events. If they exist, it will add these failed updates to the remote list as if they were received from the repository.

Besides the concurrent write mentioned above, there is also another concurrent modification that may generate conflicts. This conflict is related to delete a file on one side while modifying the same file on the other. Let's see an example: If during the synchronization round the "File1.txt" was deleted on the repository, but modified locally, then there is a problem, because if the client deletes the local copy on the cache to keep the consistency with repository, it will also delete a version of a file that was modified by the user, which means that the updates to that file are lost. Since this might lead to undesirable effects to the user, the client will ignore the delete coming from the repository and instead, send the updates to the repository as if the file was just created. Still, if it happens the other way around, it is different. That is, if the file was deleted locally and updated on the repository, the remote copy will be deleted. The difference in this case is that if the file was deleted by mistake, then it is still possible to go to the web interface and recover the file and its updates, which is not possible in local updates. With the use of this procedure in both cases there are no updates lost.

There is however an exception in which it might be possible to lose the updates. If a folder is deleted remotely and there were local updates to files in it, the client will erase the folder and every file in it. So the user updates are lost.

3.6.2 Delta encoder/decoder

One of the challenges in distributed file systems is the network, as its conditions can change often. Since this client should provide mobility, it will face unpredictable conditions which can lower the user experience, especially in long file transfers. To reduce the impact of this bottleneck, it is vital to reduce the amount of communications with the remote repository. One way to achieve this reduction is by

exploiting the similarities between file versions. If a file was already transferred previously to the remote repository, then it is possible to exploit the file similarities by only transferring the changes of the new file. Also, if a user mistakenly saves the file without changing a bit, the client will know that there was no change, so it will not transfer it. Let's consider the following case:

We have two files which contain generic data composed by numbers:

- Data1 is '112233445566778899'
- Data2 is '112233445**5**6778899'

The Data1 is the original, while the Data2 is a modified version of Data1. If we analyze them, they are different. Just by looking at them, we can check that in Data2 one of the 6 was replaced by a 5. So we could easily transform Data1 in Data2, just by knowing that we had to replace the 6 by a 5.

In file transfer what usually happens is that, one user has Data1 while the other has Data2, but they do not know the contents of the other. So, if the user of Data1 wants to transform it into Data2, just as in the previous example, he will have to transmit the entire Data2 to analyze it and see the actual differences, which would be the same as transfer the entire file.

While transferring small files like in the above example does not consume much bandwidth, when they have several MB it is a different case, especially if the changes are small. To solve this problem we implemented a file transfer protocol based on Rsync algorithm [48]. The aim of this algorithm is a protocol that allows the transformation of Data1 into Data2 by only sending the minimum information possible over the network.

3.6.2.1 Algorithm overview

In this algorithm it is assumed that each entity (for example, the client and the repository) has its own version of file and only one of them has the final version of the file (i.e. the version in which the file should be transformed into).

In summary, the algorithm is processed in three steps:

- Generate a description of the previous file version (File 1)
- Detect changes between files
- Transformation of the file (File 1 into File 2)

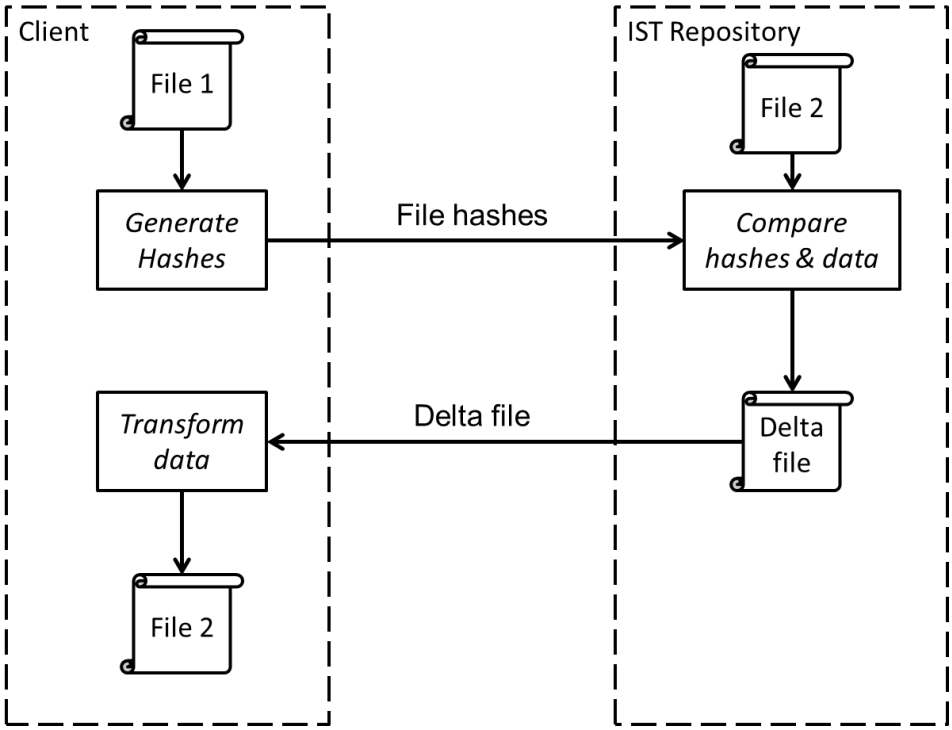

Fig. 10. Delta encoding process

Suppose that the repository has a modified version of the File 1 present in client, the File 2. When the client detects that there were modifications it will send a resumed description of the contents of File 1 (file block hashes) to the repository, over the network.

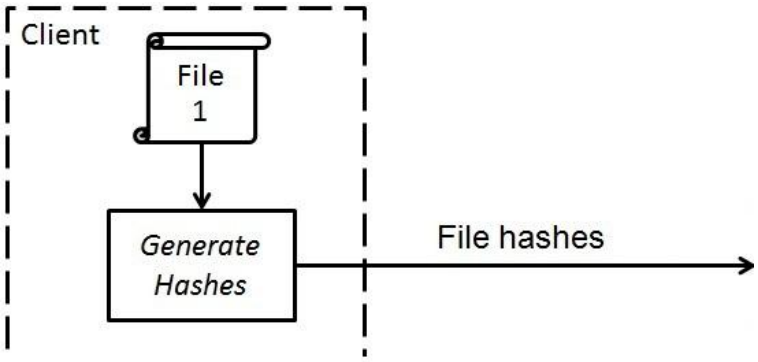

Fig. 11. Generation of the resumed description of File 1

The repository will detect the modifications between both versions and will generate two temporary files. One file contains the new contents found in the File 2, if there are any, while the other references the structure of the resulting file, i.e. instructions to transform the File 1 (on the client) into File 2. For example, it can contain information about where to put the new contents or leave the existing ones. These two files are compressed in a unique file, resulting in a delta file.

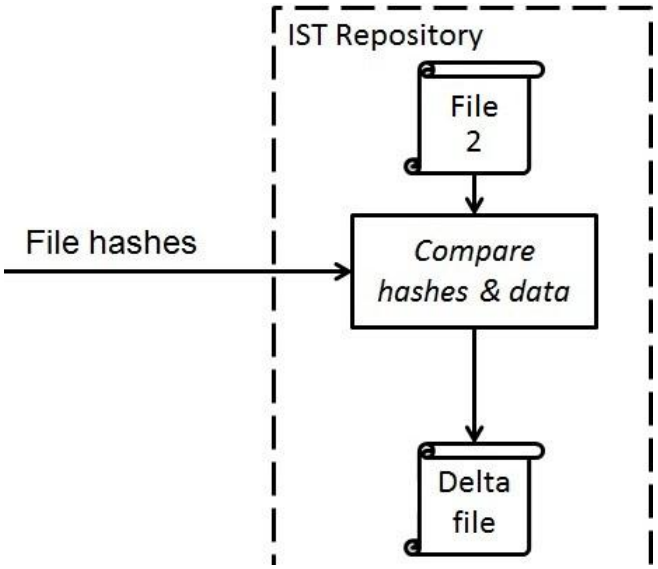

Fig. 12. Detection of changes between File 1 hashes and File 2

In the final step, the delta file is sent through the network to the client. The delta file is read and by merging the existing contents of the File 1 in the client with the delta file that came from the repository, the client reconstructs the File 2.

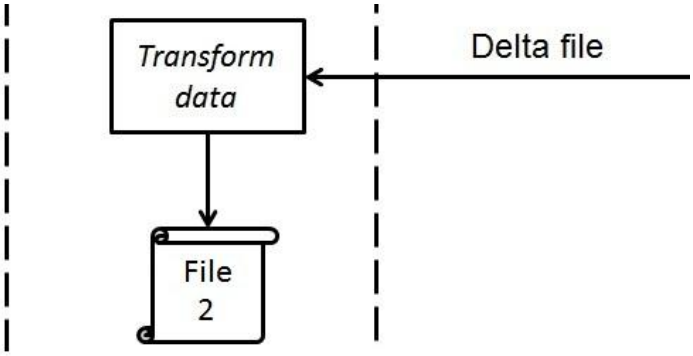

Fig. 13. Transformation of File 1 into File 2

Each of these steps of the algorithm will be explained in detail in implementation chapter.

3.6.3 Repository Connector

Finally, when the client needs to communicate with the repository, it is made through the repository connector. The communication occurs in cycles of 5 seconds and always in the same order.

First, the client requests the repository for the timestamp of the last update performed on it. If the timestamp retrieved is different from the one in known by client, then there are updates. Else, the client waits another 5 seconds.

When there are updates, the client requests the list of updates performed in the repository until that timestamp and compares them with the local changes, just like the algorithm described in the Synchronization Resolver (3.6.1). Depending on the updates needed, if there are any folders to be updated, they are created and/or deleted. The reason to update folders before the files is simple: if there is a new file created on a new folder, it is necessary to put it in the corresponding folder.

Once the folder updates are done, the algorithm moves to the files. Each file in the list is tested according possible conflicts between them. If the file passes and no conflicts are detected, it is ready to be updated. Consequently, the file it is locked, so that the user it is unable to change it during the operation. In a file transfer from the client to repository, the file it is simply locked and when the upload is finished, the lock is released.

If the transfer is from the repository to the client, the file locked and the new copy transferred to a temporary folder. When it finishes the transference, the temporary file is moved to the corresponding folder in the cache. Despite being a transference from client to repository or otherwise, it will always use the delta encoder. The only exception is when the file is new, because in this case is impossible to compare it with another. In these conditions the transference will consist in sending the entire contents of the file, as a regular file transfer.

When the list of updates is empty, the timestamp of the client is updated to the one retrieved previously. It is important to update it only after processing the entire list, because if there was a network error or the client crashes during the operation, the client will still be able to retrieve the same list of operations and none of the updates is lost. But in this case the updates will be reapplied.

The details involved in this operation will be discussed in the implementation chapter.

3.7 Summary

In this chapter was presented the main solution. The objective of this work is to develop and implement a client that can synchronize with the IST repository, making it more complete. The actual working system of Bennu is capable of maintain by itself a repository through a web portal. To achieve its goal, the system was divided into modules where each one plays a different role.

The authentication was designed to use the same authentication services that IST provides. The reason is that, as the user authentication credentials are provided by IST, it has strict rules about security and tries to avoid solutions that require the user to input both username and password directly in a 3rd party programs. The chosen authentication service was Central Authentication Service (CAS), as it offers the possibility to start the authentication through the browser, and once it gets the CAS ticket, delivered it to the client, which finishes the authentication process. This way, the client never has access to the user authentication credentials, but is able to authenticate the user.

The cache entity is physically a regular folder of the OS created in the user's computer which can contain other folders and files. The cache can be read or written by the user's applications or the repository (through the client application). While the cache on its own can bring huge improvements in

the access times, it does not suit us well if the coherence between the local cache and the remote repository is not maintained. For this reason it was created the File System Watcher (FSW). Its goal is to monitor the local cache using a hybrid algorithm of directory monitoring and extensive file search.

The Event Queue is responsible for keeping the persistent state of the client along the various executions. For this task, it uses a local database to mirror the local cache structure, but with additional metadata information about synchronization state of each file and directory. In order to keep the database updated, the Event Queue receives events generated by the FSW, they are processed and stored in the database. The events sent to the Event Queue can be of three types: create, modify and delete.

Finally, the client dispatcher is the responsible for the connection between the client and the repository, and the whole process of file synchronization. During the file synchronization, it is possible to transfer only the changes of a file, if it was already transferred to the remote repository previously. This concept is introduced in Delta encoder/decoder.

4 Implementation

This chapter describes the implementation details of the model introduced in the Architecture chapter.

This chapter is organized in a first section which will explain the technologies used to implement the project and the following ones will provide additional implementation details, which will complement the ideas presented in the Architecture chapter.

4.1 Implementation details

This project was implemented using several technologies. The repository, which was provided by IST, is mainly developed in Java 1.6 and uses MySQL to keep the system persistence. Additionally it uses Fenix Framework [8] which allows the development of Java based applications that need a transactional and persistent domain model. This framework can map the objects directly to a relational database while hiding it completely from the programmer. So, the programmer just needs to worry about the normal programming tasks. The repository's services are hosted in Apache Tomcat.

The BennuFC client was developed entirely in the scope of this project. Just like the repository, it was developed in Java, which provides great portability. The choice of the language was also influenced because this project is mainly targeted to IST and Java is the programming language mostly used by developers of the Fenix team. To keep the persistence, it uses SQLite which provides most of the features needed of a SQL relational database but does not requires any installation. The connections between the repository and the client were made using Jersey RESTful Web Services 1.7 [61]. The client was implemented and tested in Windows 7.

4.2 Authentication module

We have seen previously that in order for the client application to be able to login, it had to use the browser as the intermediary to obtain the CAS ticket. This way, the client fulfills the requirement of authenticate the user without the need of knowing its credentials. So when a user needs to perform a login with the client, he uses the client application to launch the login webpage in his web browser. The login webpage is by default IST login webpage (https://id.ist.utl.pt), but additionally, will be passed the service argument, for example:

https://id.ist.utl.pt/cas/login?service=bennufc://

The service argument is used to redirect the user to that URL, after a successful authentication, containing the new CAS ticket. When the user inputs his credentials in the login webpage, it sends those credentials to the IST Authentication server (https://id.ist.utl.pt) that will forward the request to the CAS server. In this message, besides the user credentials, it is also sends information about the

service that the user is trying to authenticate, which usually is a service endpoint. For security reasons, the service endpoint is validated to be sure that it is an authorized service. In this case, it was necessary to previously register the “bennufc” service in the CAS Server; otherwise it would reject the request. Upon a successful authentication, the CAS redirects the browser to the given endpoint with the CAS ticket as an argument, for example:

bennufc://?ticket=ST-12555ohHiImanOnionTicket353454o5bk-id.ist

Authentication protocol

Until now, the usual requests for IST CAS authentication were made from web applications, such as the Fenix application (https://fenix.ist.utl.pt) or the DOT application (https://dot.ist.utl.pt). In these cases the browser could handle the URL protocols (https) and the authentication was done fully through the browser.

But when it comes to a custom URL, the browser cannot handle it all by itself, like it did in previous web applications. Instead, it will execute the authentication protocol just as illustrated in figure 14.

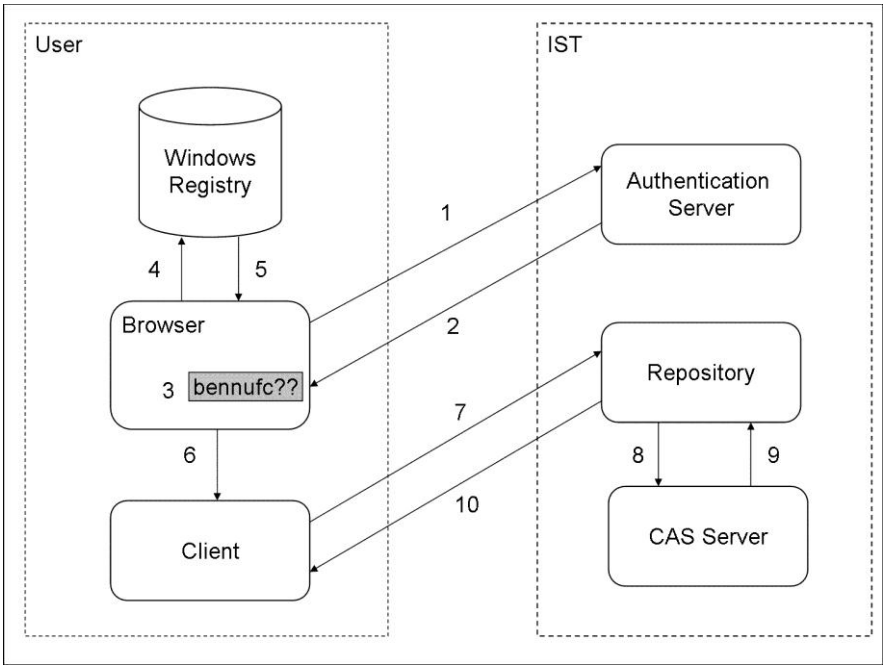

Fig. 14. Authentication protocol v1

After the user send his credentials through the browser (step 1), the Authentication Server will validate them. If they are valid, the Authentication Server will return a URL (step 2) containing: the service (which was passed as argument previously) and the ticket, as argument. The URL will be similar to:

bennufc://?ticket=ST-12555ohhi1manOnionTicket353454o5bk-id.ist

As “bennufc” is protocol that the browser does not support natively (step 3), it searches in the Windows Registry for that specific URL (step 4). If it finds an entry (step 5), the browser will execute the designated application (step 6) using the URL as the application argument. This way, the client will be

started with the ticket as argument. To validate the ticket, the client sends it to the repository (step 7), which redirects the ticket to the CAS server (step 8). If that ticket is still valid and was never used, then it returns the username corresponding to that ticket (step 9). Internally, the repository associates that username to the session established previously with the client (step 7) and informs the client of the successful authentication (step 10). From now on, the client is authenticated and allowed to execute operations over the user workspace using the repository API. Still, in order to keep the current session information, the client must keep the cookie information returned at step 10, more specifically the "JSESSIONID", and resend it every time he performs an operation at repository.

Inter-Process Communication

The previous authentication is always valid, as long as there is no client running at the moment of the login. Otherwise, it has the main problem that every time that is necessary to perform a new login (for example when session expires) the browser would launch a new client. This would generate conflicts in the access to common resources, such as database or user workspace. To address this issue it was created new Authentication protocol v2, which only changes the previous step 6 of the protocol.

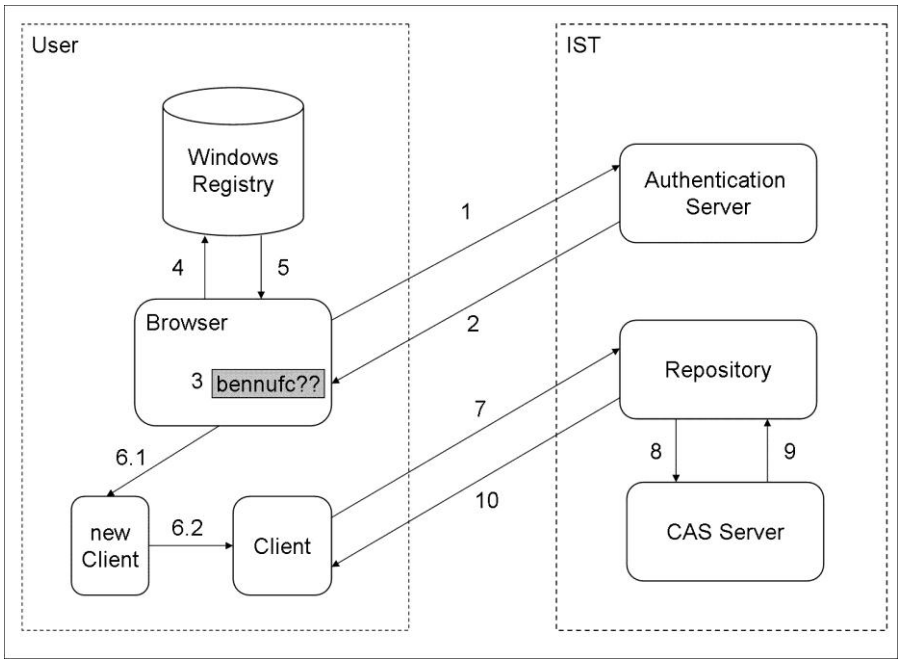

Fig. 15. Authentication protocol v2

This problem was therefore split in two sub-problems: detect if there is a client already running, and if there is, deliver the new ticket to the already running client, so he can re-authenticate. Also the solution, must keep the portability of the application and also do not rely on any specific browser.

The solution was accomplished using a common Inter-Process Communication method: TCP Sockets. They keep portability of the application, offer a flexible communication protocol and are easy to use.

Regarding the first sub-problem, when the client starts, it will bind a specific port and create a server socket. This way, if a new client is started and tries to bind that port, it will fail, meaning that probably there is a client running at that port. The downside of this solution is that if there is another user defined application binding that port then the client will always be tricked into thinking that there is another client running, when in fact there is not.

As for the second sub-problem, we restricted to the methodology used by the browser to handle custom URLs. So whenever the browser encounters a “bennufc” protocol URL, it will launch a new client application, as it did before (step 6.1). But instead of communicating right away with Repository (step 7), the new client will check if there is a client already running, by creating a new server socket. If there is, the new client will deliver the ticket through the socket to that client (step 6.2) and exits. The remaining authentication process is performed, as previously, but by the already running client. If there is not any client running, the new client executes the rest of the authentication protocol, as in Authentication protocol v1.

4.3 File System Watcher

This component was implemented using the new features of directory monitoring of Java 7. Although the idea of implementing a shared cache between the local client and the repository greatly simplify the system usage, it was not so easy to implement. The main difficulty of implementing the monitoring of a folder, just by using polling events, is that Java 7 library does not provide the source of those events. Therefore either if it was the user or the client (in behalf of the repository) who modified a file, it is generated the same event so, the FS Watcher does not know who made the modification.

Example:

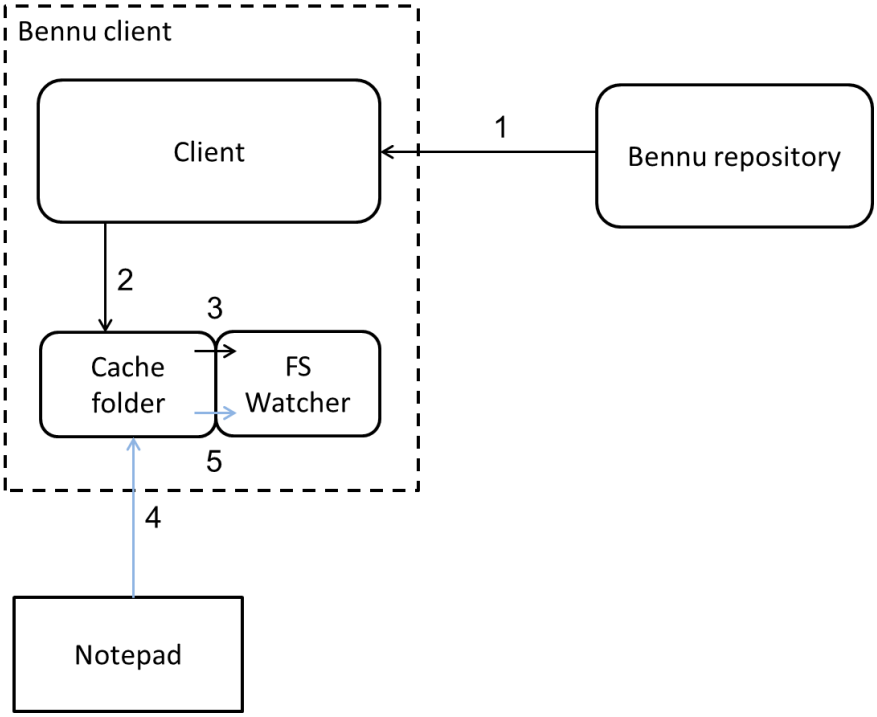

Fig. 16. Various event sources

When a user modifies a file in its local FS (4), the client receives notification of that operation (5). But when a remote modification happens, for example by another user sharing the same file, our client is notified (1) and the modified file is transferred to the local cache (2), which also generates the same event: a modify event in the cache (3).

This problem is particularly relevant because the events received locally must be propagated to the remote repository to keep cache persistence. So, if the origin of those updates is the user, for example, a modified file in its workspace (4), the client will receive a notification of that modification (5) and propagates that update to the repository. But if the origin of this modification is a remote modification (2), the client would re-propagate the updates of the repository to the repository again, which in its turn would generate another notification of a modify file, starting a non-stop cycle of updates. As so, it is absolutely necessary to differentiate the source of the events.

The final solution developed involved several attempts to fix this problem. One common aspect to all of them was to create a temporary file in a temporary folder when the file is being retrieved from the repository, reconstructed with delta encoding and then move it to the final destination. This way, there were not generated any events while it was being reconstructed. The events were only generated when the file is moved from the temporary location to the cache.

The first attempt was to use the file attributes by hiding the file. If the file came from a repository it was hidden and then moved to the cache. So when the File System Watcher received the notifications, it analyzes the file and if it was hidden he returned him to visible and ignored the event propaga-

tion of that event to the repository. But there were two additional problems: if a user moves a hidden file to the cache, it would ignore that file, until another event related to that file occur. The second was related to OS API to modify file attributes. If these functions were used it would compromise the platform portability, as they are different from OS to OS.

The final solution found was to keep a list of recently transferred files in the memory with the filename, pathname and event type, so that when the notification is received, it is ignored. Each element of that list also has a five seconds timeout, so that if the event never occurs, that element is cleared to prevent ignoring future events from the user. It must also be noted that when a file is modified, the same modification can result in receiving more than one modify event. This is highly related to the way that OS handle writes in files. For example when copying a file from the temporary folder to the cache, Windows generates a create event and a couple of modify events. To overcome this difficulty, it was created a custom write. When the file is ready to be moved from the temporary directory to the cache, it creates an entry in list which notifies the FSW that the file is going to be moved. Then the cache's file, which is going to be modified or created, is locked. Finally the contents of the file are moved into the cache. If any events related to that file happened then they are ignored. When the move ends, the file is closed, but not unlocked. Instead we create a timeout and any remaining events are ignored. When the timeout expires the lock is released and the user is now finally able to modify the file. The method to lock the file ensures that the user does not update the file while it is being moved from the temporary directory, so the only events received are coming from the repository and they must be ignored.

4.4 Delta Encoder/Decoder

The choice of the delta encoder algorithm was not easy. There were several attempts to find a suitable algorithm, but most of them worked only with local files. For this reason it was developed a delta encoding algorithm from scratch.

The implementation is based on the same algorithm used by remote Rsync [48] to generate the delta files and reconstruct them. One of the main differences between the delta encoder developed and Rsync is the hash algorithms used. Rsync uses MD5 and adler-32 checksum, while the new algorithm uses SHA1 and Rabin fingerprints. The decision of using stronger hashing algorithms was made to ensure that the likelihood of further collisions is smaller than in Rsync.

The Delta encoder encodes and decodes files in three distinct steps:

- Generate a description of the previous file version
- Detect changes between files
- Transformation of the final file

The description of each step is described below.

4.4.1 Generation of a description of a file

As we have seen before, if the changes were calculated with the exact contents of the file, the contents transferred would be the same as the entire file. The solution found was to generate a resumed description of the contents of the file. The resumed description is computed by dividing the file in blocks of a fixed size and each block is processed with two lossy compression algorithms.

A particularity of lossy algorithms is that they can generate a small value (aka checksums, digest or hash) given a data large block. The problem arises when the generated values of the block are too small and there is a probability that for different data blocks are generated the same checksums. This is occurrence is known as a hash collision. The solution was to perform a checksum of each block with two checksum algorithms with different purposes.

The first is Rabin fingerprints, which was also used in LBFS. Its purpose is to do a fast comparison between blocks. What makes it faster than others is his rolling hash property. This feature allows that given a large input of bytes and a block size set, to quickly calculate the hash of the defined size block for each byte in the sequence, by reusing the last calculated value. The values calculated due to its relative low resistance to collisions can generate some false positives.

To address this issue, it was necessary to use a second checksum algorithm that could reduce as much as possible the probability of collisions, while still providing relative fast computing. So in a normal block comparison, wherever the first hash algorithm generates a positive match of a block, it is confirmed using the second hash algorithm.

Another important aspect while considering the hashing algorithms is also the block size chosen, because for each block will be generated two hashes. If a file of 100MB is split in blocks of 4KB, there would be 25600 hashes to calculate and send over the network. As produced hash file is so long that will be reflected in its transference.

The choice for the second algorithm was SHA1, which offered a considerable resistance to collisions 1/ (2^69) [62], fast computing and hashes of 160-bit.

By combining these two checksum algorithms we can have the benefits of both. To check for similar blocks we use the first algorithm which is faster. If this algorithm finds any matching block, the contents of block are confirmed by computing the SHA1 algorithm over it. To help understand the method used to detect changes between files, let us return to the example of Data 1 and Data 2: The client has Data 1 and the repository has Data 2, which is a most updated version of Data 1. When the client detects that the repository has a new version, it will try to transfer it using the delta encoding algorithm.

To generate the file descriptor, the client divides Data 1 in blocks of a fixed-size. In this example let us consider that the block size is 2 bytes. For each block is generated the corresponding checksums:

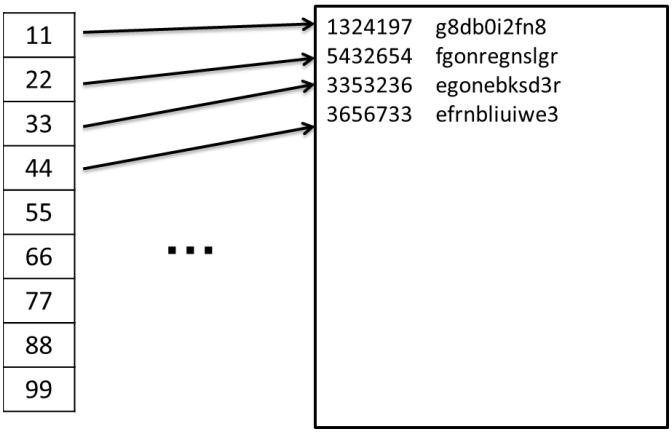

Fig. 17. File hashes generated by Data 1

When all the hashes are computed, the final result is output to a file. For each block found, it is created one line containing the Rabin Fingerprint and SHA1 hash. The order of insertion in the file is the same as the blocks that were found on the file. Line 1 will represent the first block of the file, line 2 will represent the second block, and so on.

4.4.2 Detecting changes between files

The resumed descriptor is forwarded to the repository, which will compare the descriptor against the Data 2. The objective of this phase is to find similar or changed blocks and report it to two new files: the instructions file and the binary file.

This procedure starts by reading the first block of Data 2 with length of the predefined block size. Over that block it is calculated the Rabin hash, which is then compared with the values stored in the resumed description file. If any of them matches, then it computes the secondary hash and attempts to match it against the SHA1 of that entry. If it also matches, the application assumes that it is the same block and is added an entry in the instruction file with “Insert #” where # is the reference to the number of block present in the resume description received, i.e. a reference to Data 1 block.

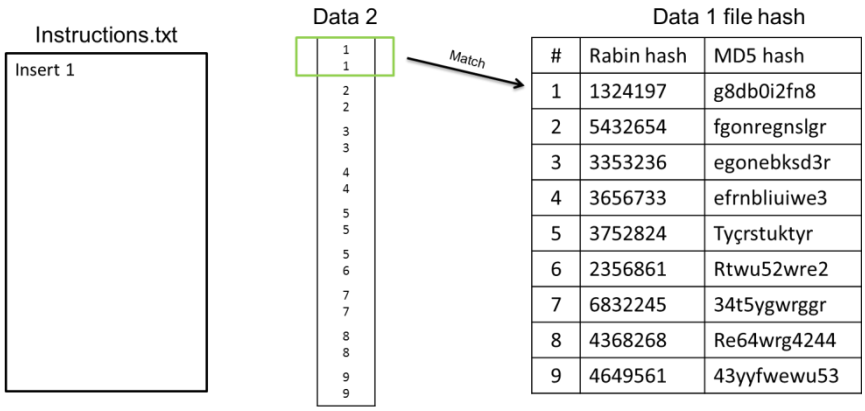

Fig. 18. Comparison of blocks of Data 2 against the Data 1 resumed description with a successful match.

When a block is found, it is read the next one. However the behavior is different if the Rabin or SHA1 hash do not match any of the entries in the resumed description. In this case, it is set a pointer to the first byte of the block which was unmatched and reads the byte after the analyzed block.

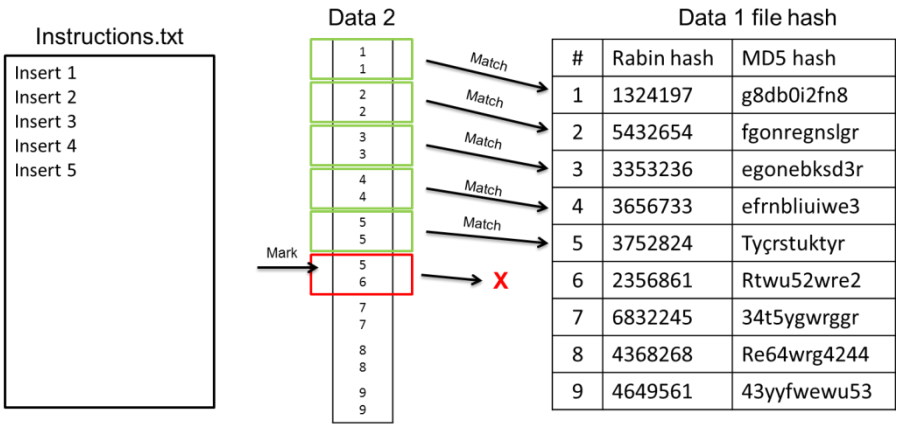

Fig. 19. Comparison of blocks of Data 2 against the Data 1 resumed description with an unsuccessful match.

The new Rabin hash is recalculated to try to match the new hash with the values on the resumed description. Here, the property of the rolling hash comes pretty handy because it will recalculate the value of the hash just based on the previous value calculated and the new byte read. If it still does not find any matching value, the algorithms repeat the same steps until either file ends or it is found a block.

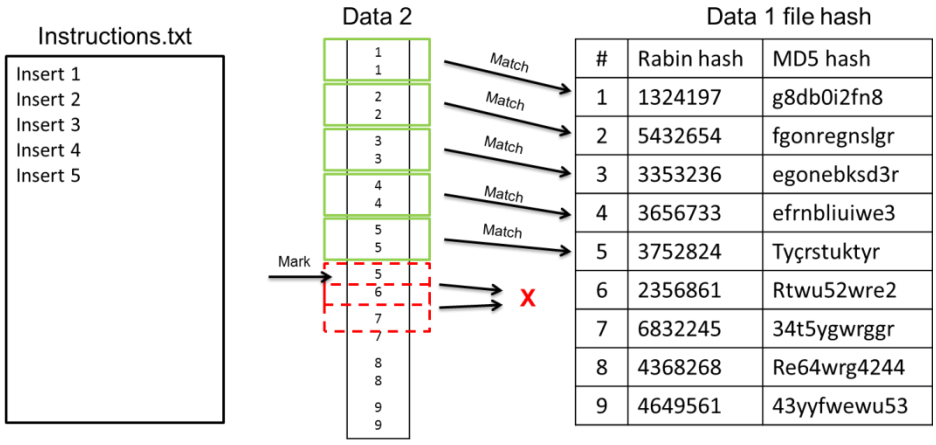

Fig. 20. Comparison of blocks of Data 2 against the Data 1 resumed description with two unmatched blocks

When such happens, the bytes, corresponding from the first marked byte until the actual, are read and appended to the final of the binary file. This file will store all the data that is considered to be missing in Data 1. Also it added an entry to the instruction file with “Read #1, #2” which will give an instruction towards the new bytes to be added in Data 1. The #1 is a reference to the first byte of the binary information and the #2 is its length.

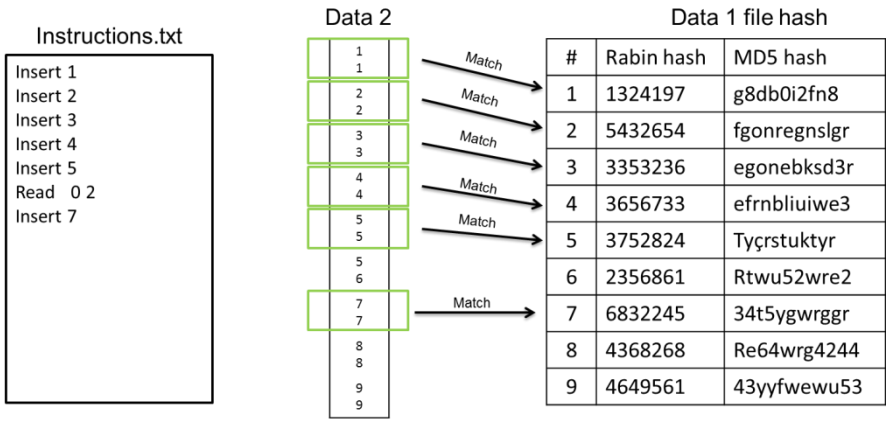

Fig. 21. Comparison of blocks after the two unmatched blocks

The result of this comparison is the two files generated through the execution. These files are merged in a delta file and sent to client so he can transform his Data 1 in the repository’s Data 2.

4.4.3 Transformation of the file

When the client starts the transformation it has three files:

- The client’s Data 1 file;
- Delta file, which contains the instruction and binary file.

This process involves the creation of a brand new file without any contents in a temporary directory. The goal is to given the three file mentioned above recreate the Data 2 in an empty file. To execute the transformation the client starts off by reading each line of the instructions file. As there are only two instructions, there are only two operations that can be performed.

If it is “Insert #”, the client will read an array of bytes from Data 1. The read pointer is set on the Data 1 file and it reads starting from the # (which is the number of block) * block size, and with length corresponding to the block size. Then, the array is concatenated with contents of the new file.

If the operation is a “Read #1, #2”, the client will read an array of bytes from the binary file. The file pointer is set to start in #1 (which is a position in the binary file) and reads until the length is #2. As in the previous operation, the array in appended at the end of the new file.

When the client reaches the end of the instruction file, the Data 2 is already reconstructed and ready to be moved to the final destination. However the move operation is quite different than a normal move of a file. The reason is that the client has to move the file to the cache, but as it is being monitored, if the file was moved using the OS move file operation, the client application would interpret this move as a change made by the user. So before the file is moved, it is needed to first register this move with the File System Watcher, so that the event will be ignored. This addresses the problem discussed in the section of Event Sources, where the File System Watcher would be tricked into thinking that the modification caused by this move would be an update from the user, when in fact it is an update from the repository.

4.5 Summary

This chapter was created as a complement to the Architecture chapter.

In the authentication section was introduced the Inter-process communication protocol, using TCP protocol. It is mainly used when is necessary to perform a new login (for example when session expires) and there a client already running. As the re-authentication using the browser would launch a new client, it would end up generating conflicts in the access to common resources, such as database or user workspace. To prevent it, after the browser authentication, the newly created client will deliver the CAS ticket to the older client using an already existent TCP socket and exits. Then the older client completes the authentication.

The sub-chapter is followed by a detailed description of FSW and its relation to the Cache. Also it explains the main problems encountered in its development and how they were addressed.

Finally, is described in detail the delta encoder/decoder and how it works.

5 Evaluation

This chapter aims to describe the test procedures used to test Bennu FC client application with the IST Repository. The procedures were split in two parts.

The first procedures are functional tests. The objective is to explain and test the overall behavior of the client according to the defined functional requirements of the project. The verification will be checked using casual use cases, which will cover the step by step operation performed in the whole system.

The second part will consist in performance tests. The tests will check on the gains in data transfer using the delta encoder algorithm versus the standard and the tuning of delta encoder algorithm using different hash algorithms and block sizes.

Finally we have the scalability tests. This test will consist in benchmarking the time that takes to complete a download, using delta encoding, when under load of several clients. This operation was chosen because in the previous test results it was the most demanding operation.

5.1 Test environment

The overall system is composed by the BennuFC client and IST repository, and was tested in a single machine. This allowed testing the simplest case where there is one client connecting the repository and eliminate the possible network latency variations. The test machine has the following hardware setup: Intel i7-2670QM CPU @ 2.20 GHz, 6GB Ram, Windows 7 64 bits and HDD 5400 RPM.

For the scalability test, it was added a second machine to simulate concurrent clients, while the previous was hosting the repository. Each client will be executed in a different thread.

5.2 Use cases

The following tests were executed using the single machine setup. Regarding the first use case, only for this test specific test it was enabled the authentication module. The reason is that when this module is enabled the IST repository uses the IST CAS authentication, but the endpoint of the repository is “localhost”, instead of IST domain. Unfortunately this endpoint is not allowed to use CAS services and like that it is impossible to login at the web interface of the repository. Nevertheless the clients are still able to authenticate and perform operation at the repository. On the other cases it was chosen default a user in the repository configurations and the authentication was disabled.

5.2.1 UC_001 – Authenticate a user

Primary Actor:

User

Pre-conditions:

Authentication must be enabled in the repository
Network connection to the repository

Scope

Authentication

Story:

1. The user starts the use case by running the BennuFC client application, which will display an icon in Windows task bar.
2. The client checks the cache folder looking for modification in the cache and contacts the repository to request its updates. But as the client is not authenticated, the Repository returns an error.
3. The client shows an error message to the user saying the user need to authenticate itself.
4. The user Right-click in the client's task bar icon and select "Login". This action shows another message informing the user that the client will open a browser window so that the user can authenticate. Once the user clicks "OK", it will open a window with the address: https://id.ist.utl.pt/cas/login?service=bennufc:"
5. Once the user fills his credentials in the browser and if they are correct, the browser will ask the user if he wants to execute the BennuFC.
6. The user clicks on allow and the client is now authorized to synchronize the local cache with the repository. The use case ends.

5.2.2 UC_002 – Synchronize a file from the local workspace to repository

Primary Actor:

User

Pre-conditions:

BennuFC must be running
Network connection to the repository

User must be authenticated in the repository, as in the UC001

Scope:

File transference

Story:

1. The user is working in an application and decides to save a file in the cache folder.
2. The client is notified of its creation. As the file is new it sends the entire contents of the file to the repository and the use case ends.

Alternate paths:

2a. The file already existed in the repository, so instead of sending the entire contents to the repository, the client uses delta encoding to transfer the file and the use case ends.

2b. When the client tries to open the file to send to the repository the file is in use by the user's application. Consequently the client waits until the file lock is released to transfer it. The use case ends.

2c. When the client tries to transfer the file to the repository, it is notified of a remote notification in same file. The client backup the local user modified file and transfers the remote copy to the cache.

5.2.3 UC_003 – Synchronize a file from the repository to local workspace

Primary Actor:

User

Pre-conditions:

BennuFC must be running
Network connection to the repository
User must be authenticated in the repository, as in the UC001

Scope:

File transference.

Story:

1. The client is notified of a remote modification in a file.
2. The file does not exist in the user cache and the entire contents of the file are transferred from the repository. The use case ends.

Alternate paths:

2a. The file already existed in the use cache, so instead of receiving the entire contents, the client uses delta encoding to transfer the file and the use case ends.

2b. When the client tries to open the file to receive the modification from the repository the file is in use by a user's application. Consequently the client waits until the file lock is released to transfer it. The use case ends.

5.2.4 UC_004 – Delete a file in the repository

Primary Actor:

User

Pre-conditions:

BennuFC must be running
Network connection to the repository
User must be authenticated in the repository, as in the UC001

Scope:

File transference.

Story:

1. The client is notified of a remote delete in a file.
2. As the local copy in the cache does not have any modifications in comparison to the one on the repository, the file is deleted.

Alternate paths:

2a. The file did not exist locally, so the remote delete is ignored. The use case ends.

2b. The local copy in the cache was modified, so instead of performing a local delete, instead the file is transferred to the repository, as a new file. The use case ends.

5.2.5 UC_005 – Delete a file in the client

Primary Actor:

User

Pre-conditions:

BennuFC must be running

User must be authenticated in the repository, as in the UC001

Scope:

File transference.

Story:

1. The user deletes a file in the local cache.
2. The client deletes the file remotely.

5.3 Delta encoding performance tests

In the previous section, we presented the tests to verify the overall behavior of the system. This section is going to present the performance tests performed to the file transference algorithm developed.

To test the delta encoder algorithm, we created a separate application which had as input two files and a file block size to execute the algorithm. The main advantage of isolating the algorithm from the rest of the system is that, the benchmarks will reveal only the costs involved in the computation of the algorithm. The final objective of the application is to transform the original file into modified file by executing the delta encoding algorithm.

In this algorithm the block size plays a critical role, because it will determine the precision in finding the modification in a file. The smaller the block size, the more accurate is the algorithm in detecting the modification, which means that it will transfer smaller parts of the file. Still, it comes at cost of computational power and a larger hash file. The smaller the block size is, the bigger the hash file transferred initially will be and lengthier will be the comparison between the hashes and the modified file.

To evaluate how these variables affect the algorithm's performance, special conditions were simulated to emulate the best, normal and worst case scenarios.

5.3.1 Test setup

The file sizes, in bytes, used in tests were:

File1	File2	File3	File4	File5	File6	File7
16 KB	1 MB	5 MB	20 MB	50 MB	100 MB	500 MB

Table 1. File sizes used in tests

These files were generated using a random file generator and will correspond to the original files in each test. They are used by the algorithm to generate the block hashes with the corresponding block size and in the reconstruction step.

The block sizes, tested were:

2 KB	4 KB	8 KB	16 KB	500 KB	1 MB

Table 2. Block sizes used in tests, in bytes

The number of block hashes generated can be calculated using “file size” / “block size”, rounded up. So in summary the number of hashes generated for each file for each block size is:

Block size (bytes)	File1	File2	File3	File4	File5	File6	File7
2048	8	512	2560	10240	25600	51200	65536000
4096	4	256	1280	5120	12800	25600	128000
8192	2	128	640	2560	6400	12800	64000
16384	1	64	320	1280	3200	6400	32000
524288	1	2	10	40	100	200	1000
1048576	1	1	5	20	50	100	500

Table 3. Number of hashes generated for each file and each block size

Each of these hashes is formed by a Rabin fingerprint hash and MD5 or SHA1 Hash. As these hashes will be computed over the same files, the hash files will have similar sizes. In the table below it is presented the file hash sizes (in bytes), which would be transferred over the Internet:

Block size (bytes)	File1	File2	File3	File4	File5	File6	File7
2048	496	31677	158381	633703	1584172	3168177	15841199
4096	247	15843	79208	316821	792098	1584257	7920758
8192	123	7921	39597	158417	396026	792041	3960146
16384	62	3961	19817	79187	198047	396029	1980136
524288	62	123	620	2475	6183	12383	61890
1048576	62	62	310	1237	3095	6186	30930

Table 4. Size of hash file for each file and for each block size, in bytes

The results for each scenario are presented in two different graphics. The first graphic will show the performance tests measured for each file transferred, using delta encoder with different sizes and a normal file transfer. In the normal transfer, the value presented corresponds to the expected transfer

time with a static connection of 200 KB/s. In delta encoding transfer, the value corresponds to total operation time plus the expected transfer time with the same connection of 200 KB/s.

The second graphic will present the total data exchanged for each file transferred, using delta encoder with different sizes and a normal file transfer. In the normal transfer, the value presented corresponds to the file size. In delta encoding, it represents the hash file plus the delta file transferred.

5.3.2 Best case scenario

Objective

This case was created just to demonstrate how the algorithm operates in optimum conditions, i.e. when all blocks match the ones on hash file and are ordered.

How it was performed

To evaluate this scenario the files mentioned above were used to compare them with themselves. This ensured that every block matched to the other on the other file, which is the best case scenario.

The results, in seconds, are the following:

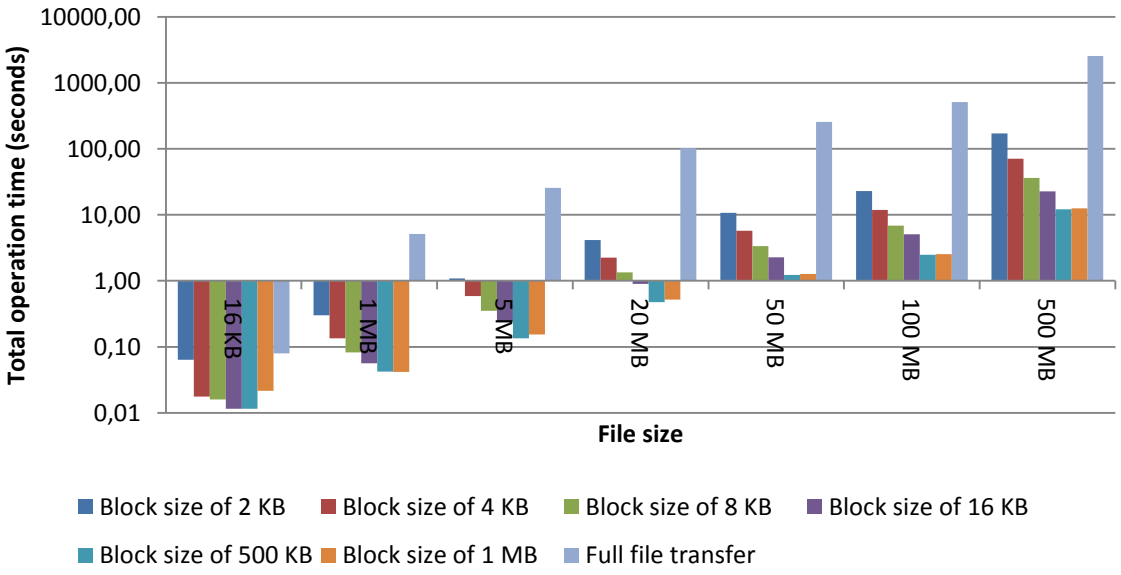

Fig. 22. Performance tests when transferring each file through delta encoding, using different block sizes

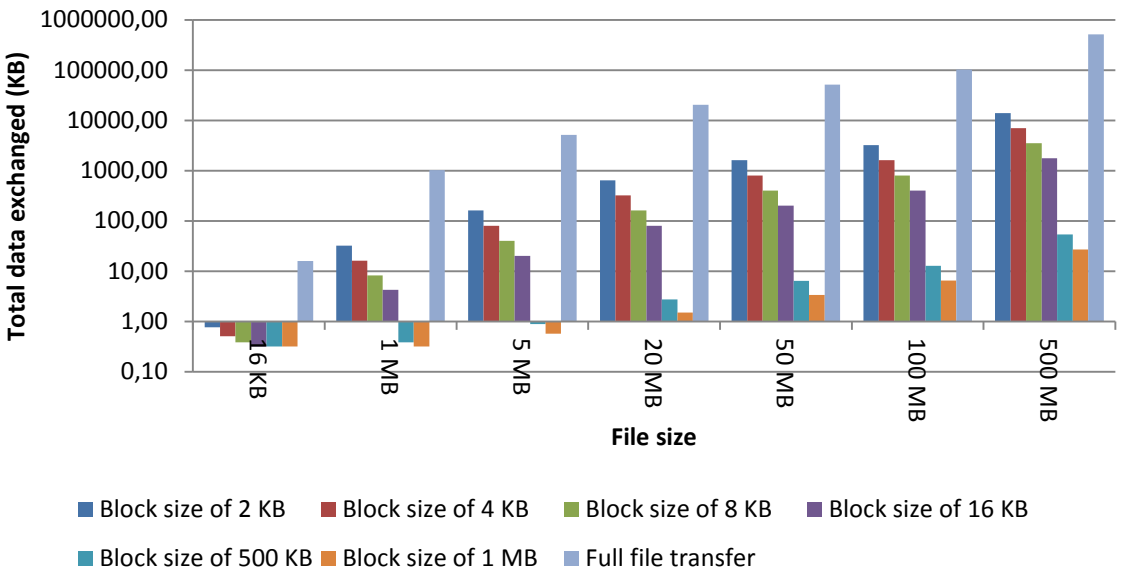

Fig. 23. Total data exchanged when transferring each file through delta encoding, using different block sizes

The figure 22 presents the time corresponding to the total time of the process of encoding and decoding. Overall the most time consuming operations were detected in smaller block sizes, because they generate more block hashes, which requires a longer search when trying to find a matching hash. Still, when comparing to a full file transfer, the algorithm performs much better.

5.3.3 Normal case: file with 2 insertions

Objective

This case corresponds to the case where the user made two insertions, of two bytes, in the original file.

How it was performed

The evaluation in this instance was done with the support of random file generator which inserted two bytes in two specific points of the file original file. The insertions were made dividing the file in two parts, appending the two bytes in the beginning of each part and finally merge them. The algorithm then computed the deltas between the original and the modified file generated. The results, in seconds, are the following:

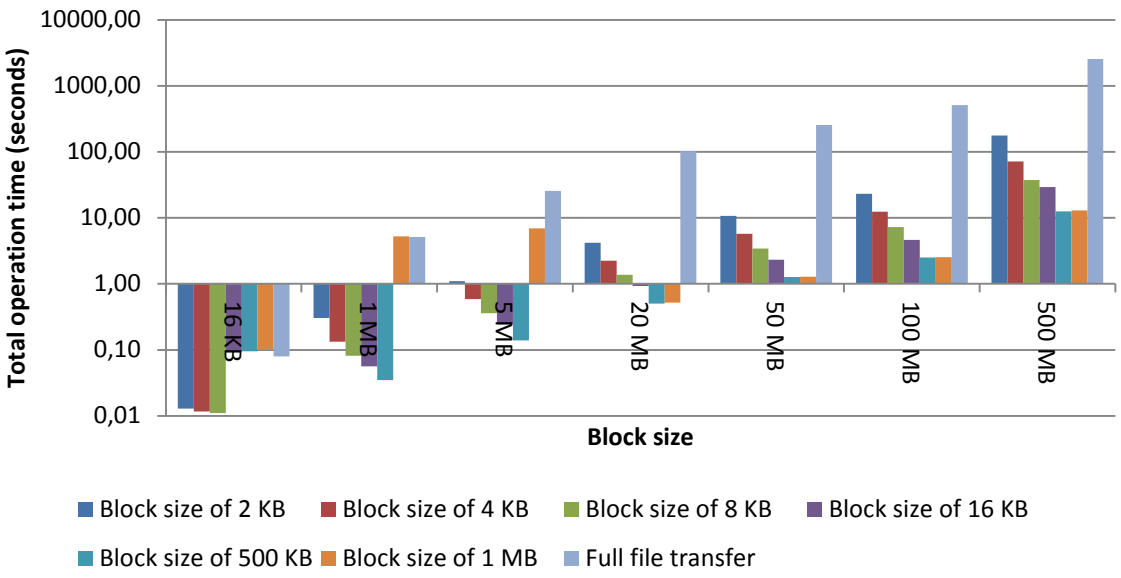

Fig. 24. Performance tests when transferring each file through delta encoding, using different block sizes

In terms of performance, the difference is not much from the best case. Still when comparing the original file size with the transferred size using delta encoder, the difference is huge. The total transferred size of each file using delta encoder is presented below:

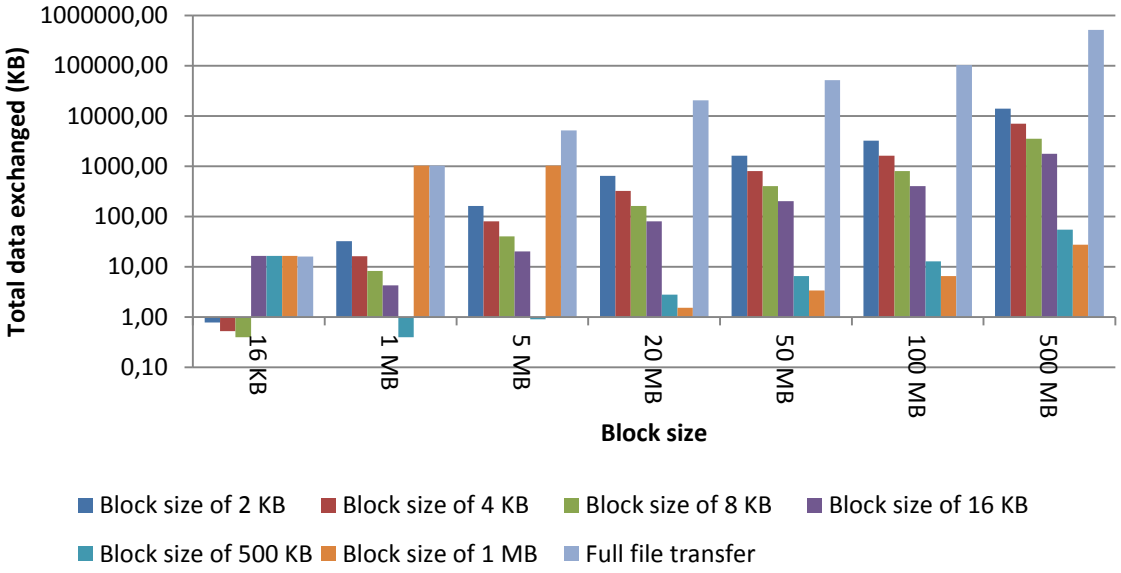

Fig. 25. Total data exchanged when transferring each file through delta encoding, using different block sizes

Algorithm behavior (explained by the test of File 1 with block size 2048 bytes)

In this test, the algorithm will start by computing the Rabin fingerprint between the 1st byte and 2048th byte. As there were 2 bytes inserted in the beginning of the file, the generated hash between those bytes will be different from any other of the known hashes. So the algorithm reads the next byte

(2049th) and computes the hash between 2nd byte and 2049th. Just as in the previous case, there is no matching hash. But when it pushes the next byte, the hash between the 3rd byte and 2050th byte will match to the first block of the original file. This happens because with the insertion of two bytes in the beginning of the file, all the blocks of the original file were "shifted" two bytes to right in the modified file. So only those new bytes are added to the binary file, while the matching blocks are referenced in the instruction file.

A common collateral effect of referencing all blocks in the instruction file is that, as the size of the analyzed file increases, the instruction file will increase as well. This can be clearly observed in the 500 MB file, where the instruction file occupies almost 571 kilobytes (KB). Still, when the block size is increased to 16 KB, it is possible to transfer only 72 KB of delta file, because there are fewer blocks.

5.3.4 Normal case: file with 2 modifications

Objective

This case corresponds to the case where the user made two modifications, of two bytes, in the original file. As so the original file size is maintained, but the blocks affected with the modifications will be different.

How it was performed

The evaluation in this instance was done with the support of random file generator which modified two bytes in two specific points of the original file. The modifications were made dividing the file in two parts, modify the two initial bytes of each part and finally merge them. The algorithm then computed the deltas between the original and the modified file generated.

The performance results, in seconds, are the following:

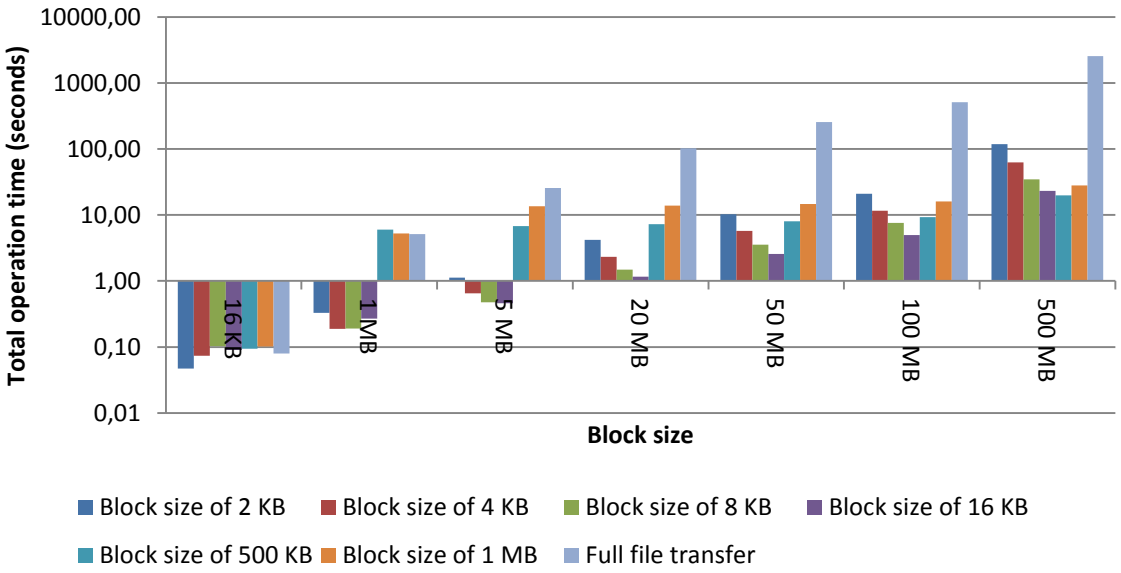

Fig. 26. Performance tests when transferring each file through delta encoding, using different block sizes

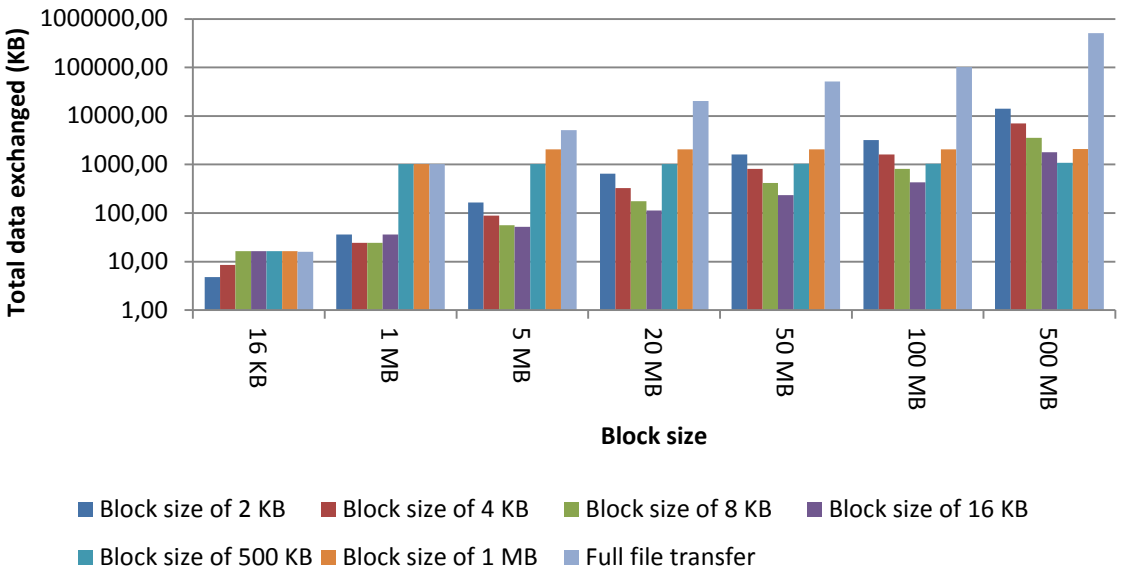

Fig. 27. Total data exchanged when transferring each file through delta encoding, using different block sizes

Algorithm behavior (explained by the test of File 1 with block size 2048 bytes)

In this test, just as in previous, the algorithm will start by computing the Rabin fingerprint between the 1st byte and 2048th byte. The main difference between this case and the previous is that, while in the previous case the contents of the blocks were intact, which generate known block hashes, in this test, the actual contents were modified. So the first block of 2048 bytes will never get a matching hash, but as the algorithm could not guess it, it will calculate the hashes from byte to byte until finds a known block hash. Only when it calculates the hash from the byte 2049th to 4096th byte, it will find a known block, which corresponds to the second block of the original file. So the first block, which did not match any hash, is put in the binary file and the ones found are referenced in the instruction file.

For this reason, as there were two blocks modified, the delta file size will be about the size of two block sizes, because only two of blocks of the original file were modified. In smaller size files, like File1, as some block sizes are larger than the size of the file itself, the entire file is put in the delta file.

Just as in the previous test, larger files will generate larger instruction files. For example in File 7 with block size of 2048 bytes, only the instruction file (included in the delta file) exceeds a lot the size of two blocks: about 571.053 bytes (this value was retrieved from the insert case results, where the delta file almost only contained the instruction file). So 571.053 bytes + 2x “block size” (2048 bytes) is around: 575.122 bytes, which was the size found.

The performance times seen in this test are slightly higher than in the previous, because instead of four unmatched blocks, one per each byte inserted, in this test there were 2048 unmatched block for each modified block. So, about 4096 unmatched blocks, which mean 4096 computed hashes and for

each an exhaustive search for any matching hash. For example in File 7 with 2048 bytes block size we have 4096 computed hashes and for each hash, a failed comparison with 65.536.000 original file hashes, which totals in 268.435.456.000 comparisons.

5.3.5 Normal case: file with 2 deletions

Objective

This case corresponds to the case where the user made two deletions, of two bytes, in the original file. As so the original file size is maintained, but the blocks affected with the modifications will be different.

How it was performed

The evaluation in this instance was done with the support of random file generator which deleted two bytes in two specific points of the original file. The modifications were made dividing the original file in two parts, delete the two initial bytes of each part and finally merge them. The algorithm then computed the deltas between the original and the modified file generated.

The performance results, in seconds, are the following:

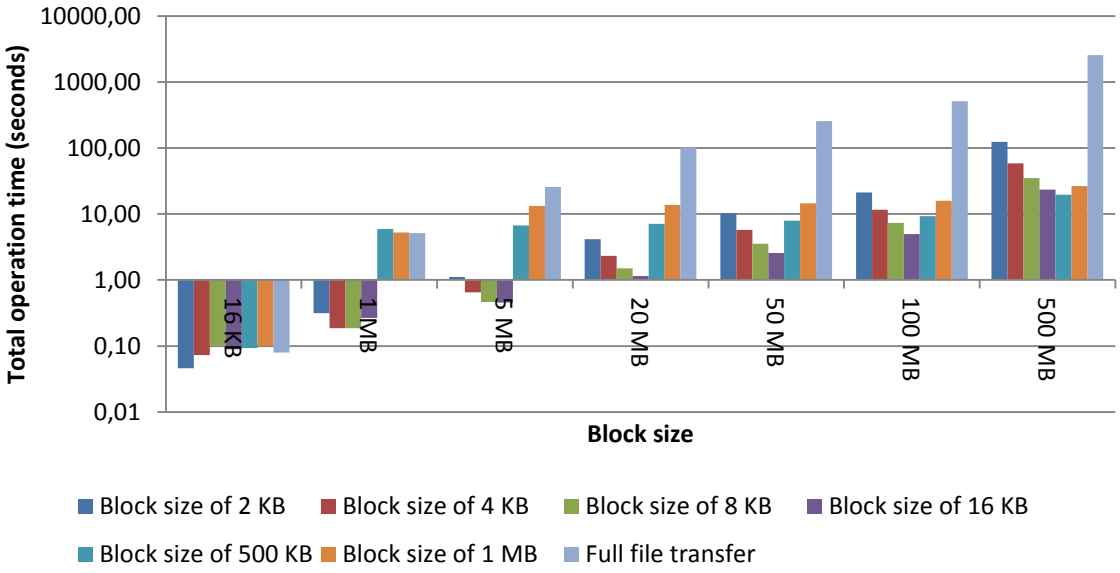

Fig. 28. Performance tests when transferring each file through delta encoding, using different block sizes

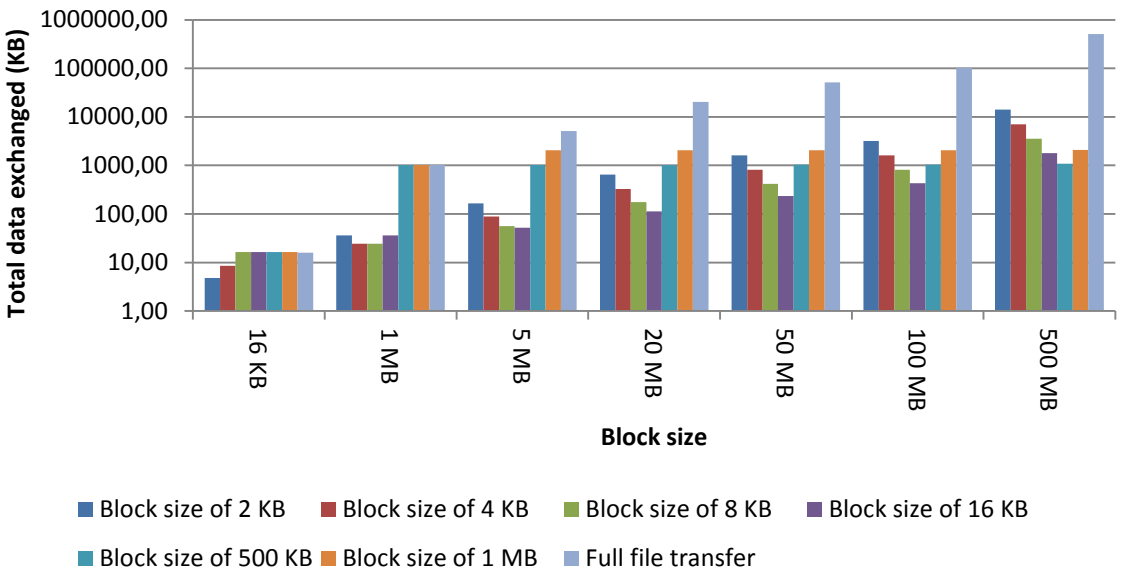

Fig. 29. Total data exchanged when transferring each file through delta encoding, using different block sizes

Algorithm behavior (explained by the test of File 1 with block size 2048 bytes)

The algorithms behavior is the same as in the previous test. The contents of some of two blocks were changed, so these blocks will never match any hash of the original file hashes. The algorithm only ran slightly faster than in the previous test, because the modified blocks were shortened in two bytes per block, instead of the replacing two bytes per blocks. This fact leads to fewer comparisons between hashes and to a faster computation.

5.3.6 Worst case scenario

Objective

This case will only happen when the user replaces entirely the contents of the file.

How it was performed

This case was reproduced by the random file generator by inverting every byte of the original file. The performance results are displayed below, in seconds:

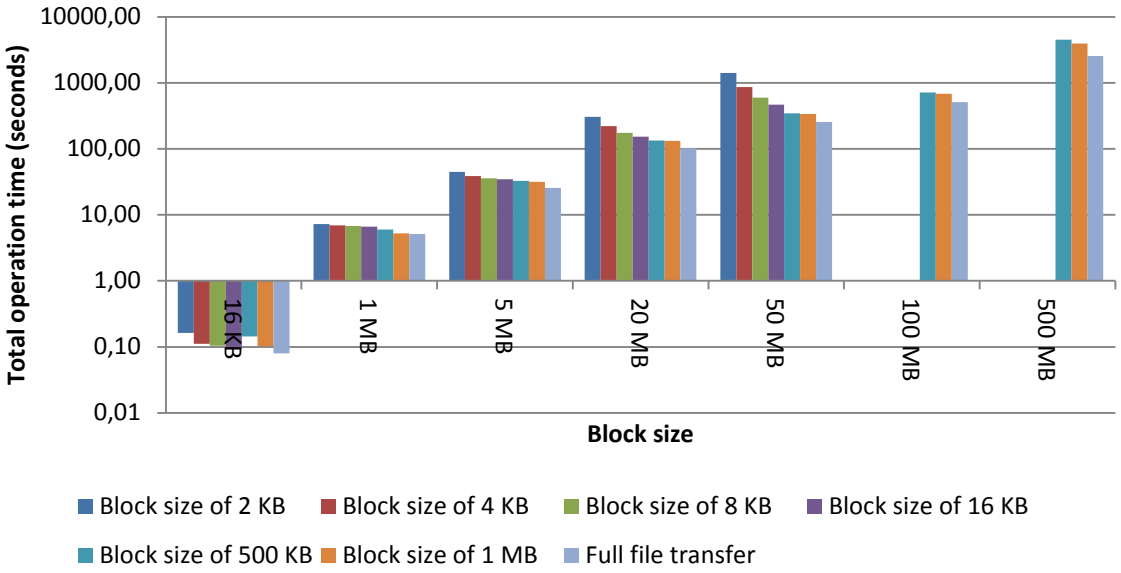

Fig. 30. Performance tests when transferring each file through delta encoding, using different block sizes

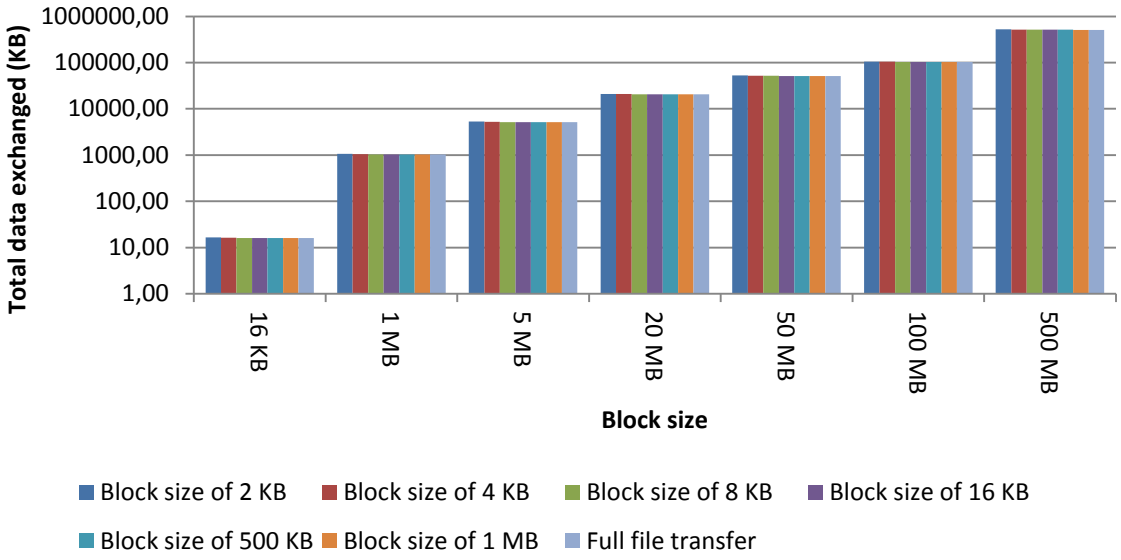

Fig. 31. Total data exchanged when transferring each file through delta encoding, using different block sizes

Algorithm behavior (explained by the test of File 1 with block size 2048 bytes)

In this test, just as in previous, the algorithm will start by computing the Rabin fingerprint between the 1st byte and 2048th byte. The main problem in this case is that there are no matching blocks, because the content of all blocks was changed. So the first block of 2048 bytes will never get a matching hash and it will calculate the hashes from byte to byte until finds a known block hash, but it will never find one.

The results are not impressive at all and for this reason the bigger files were not benchmarked with smaller block sizes, as they were taking too long to compute. The most expensive operation identified in this test was the comparison of the block hashes. For example in the File 5 with block size of 16384 bytes, the detailed performance test had the following results:

1. Generation of block hashes: 541 ms
2. Read hashes to memory: 2 ms
3. Compare files: 204812 ms

Total time spent on rabin = 1587 ms

Total time spent on search + second hash = 130149 ms

4. Compress delta files: 2112 ms
5. Uncompress delta files: 158 ms
6. Reconstruct files: 64 ms
7. Overall time: 207689 ms

This operation will consist in the following steps:

- Compute Rabin fingerprint hash
- Search for it the original file block hashes
- If it matches, compute the second hash

While Rabin's hash is fast to compute, the same cannot be said about the second hash. Another aspect to take in consideration in this test is that there are no matching blocks, so the Rabin hash of every byte is going to be computed. As this hash is weaker than the second, the chance of some false-positives matches occur can be higher. This can lead to a frequent calculation of the second hash, which would slow down the overall operation.

5.3.7 Final remarks

During the development phase there was a discussion about which second hash algorithm should be used. Among the wide range of possible options, there were chosen two, because they have been used in two delta-based systems: Rsync and LBFS.

MD5 is the second hash algorithm used in Rsync algorithm and since the algorithm developed is based on Rsync, it made sense to use the MD5. Still the authors say that they do not put aside the possibility of collisions, although they are very rare and unlikely to happen.

On the other hand there are other hashing algorithms that can achieve even higher reliability in similar times. Among the discussed algorithms there is SHA1. This algorithm was used in LBFS to hash blocks as a second algorithm which also worked together with Rabin fingerprint.

Given the choices, it was made a more conservative choice and the algorithm was implemented using by default the SHA1. Nevertheless all previous tests were performed using both algorithms, but as the space to show results is limited, instead it was presented the results of the best performing hash-

ing algorithm, which was MD5. In tests MD5 always outperformed SHA1, except in smaller files. As the tests were performed under the same conditions, same files and same blocks, it was not found a reason for this fact. The implementations of both algorithms are provided, natively by Java.

The average time performed by both hashing algorithms for each test case is presented below, in milliseconds:

	best case	Insertion case	Modification case	Deletion case	Worst case
MD5	5440	7582	4607	4571	197946
SHA1	6141	12005	8488	5333	201606

Table 5. Performance comparision between SHA1 and MD5 in delta encoder of the total operation time, in milliseconds

Regarding to the reliability of both algorithms, in every test, by the end of the algorithm it was performed a SHA1 checksum between the modified file and the final outputted file. The checksum was not only there to prove the successfulness of the developed algorithm, but also to check for any hash collisions. In these tests, there were no hash collisions, but only more exhaustive and varied tests could provide more details in relation to the reliability of each hashing algorithm.

5.4 Scalability test

The final test is the scalability test. Its goal is to test how the delta encoding algorithm behaves when under heavy load. To test it, it was created a simpler version of the client to invoke specific API related to delta encoding.

Regarding this API there are three possible calls:

- *getFileHashes*(String pathname, String filename)
    - Calculates the block hashes from a specific local file, given a pathname and filename.
    - Returns block hashes.

- *compareFiles*(String pathname, String filename, long originalfilelen, InputStream inputhashfile, int version)
    - Compares the local file against the remote file. The local file is retrieved from pathname & filename. If the local file last version does not match the specified version, it returns an error. The remote file is identified by its size (originalfilelen) and block hashes (inputhashfile).
    - Returns delta file.

- *recontructFile*(InputStream filecontent, String pathname, String filename, int version)
    - Rebuilds the local file based on the delta file received. The local file is stored at the specified pathname & filename. If the local file last version does not match the specified version, it returns an error. The delta file is retrieved from filecontent.

Wherever is performed a delta download or upload operation, all these three calls are made. The only difference is where they are executed.

In an upload the *getFileHashes* is executed at the repository, the *compareFiles* at client and *reconstructFile* at the repository. On the download operation, the sides are switched. *getFileHashes* is executed at the client, the *compareFiles* at repository and *reconstructFile* at the client.

Based on the previous tests and all scenarios, the most expensive operation was the comparison between the local file and the remote block hashes, which corresponds to the *compareFiles* operation. Like so, the scalability tests of the repository will be based on *compareFiles* call which is used when downloading a file.

The test will be performed using two machines, one with the repository (the one previously used machine for tests) and another simulating the specified number of clients using threads. These machines will be connected in the same local network.

Test description

The test results will be collected by benchmarking the elapsed time since a client evoked *compareFiles*, until it receives a response. Each client will download the same file with 50 MB using a precalculated block hash file. On response it calculates the elapsed time. The number of clients used in test will range from 5 to 50.

Results

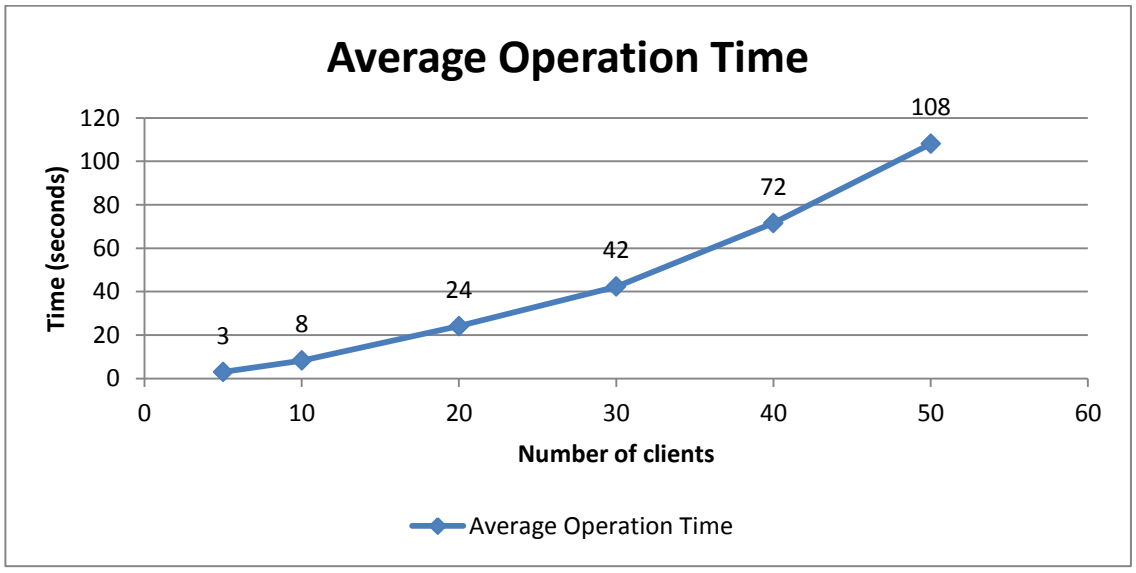

Fig. 32. Average operation time per number of clients

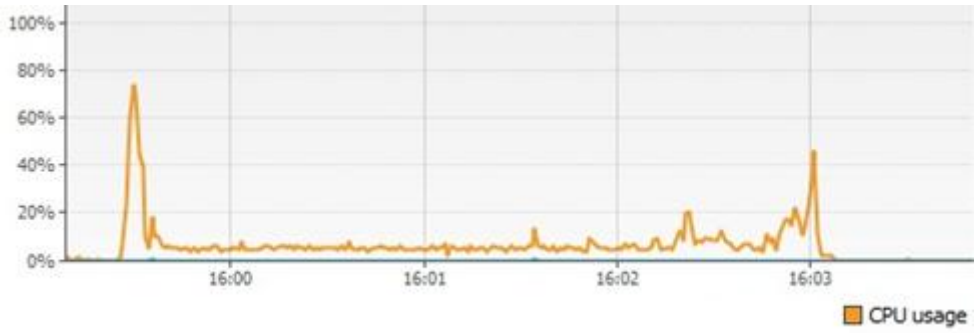

Fig. 33. CPU usage when simulating 50 clients

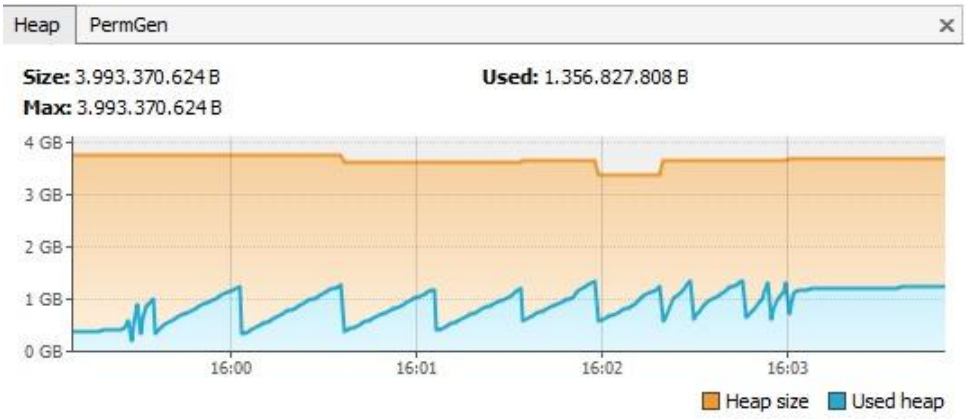

Fig. 34. Memory usage when simulating 50 clients

module.fileManagement.services.filedeltas.FileUtils.**streamToFile** ()		6340614 ms (63,2%)
org.apache.tomcat.util.net.AprEndpoint$Poller.**run** ()		1220503 ms (12,2%)
org.apache.tomcat.util.threads.TaskQueue.**poll** ()		1142586 ms (11,4%)
org.apache.tomcat.jni.Poll.**poll[native]** ()		533042 ms (5,3%)
org.apache.catalina.core.ContainerBase$ContainerBackgroundProcessor.**run** ()		204609 ms (2%)
pt.ist.fenixframework.pstm.TransactionChangeLogs$CleanThread.**run** ()		182795 ms (1,8%)
org.apache.tomcat.jni.Socket.**recvbb[native]** ()		174995 ms (1,7%)
module.fileManagement.services.filedeltas.DeltaFileWriter.**wInstrDataBlock** ()		60443 ms (0,6%)
org.jvnet.mimepull.WeakDataFile.**writeTo** ()		41794 ms (0,4%)
module.fileManagement.services.filedeltas.CompressionUtils.**compressFiles** ()		30680 ms (0,3%)
module.fileManagement.services.filedeltas.DeltaOperations.**compareFiles** ()		27563 ms (0,3%)
module.fileManagement.services.filedeltas.DeltaOperations.**calcMD5Hash** ()		24676 ms (0,2%)
module.fileManagement.services.filedeltas.DeltaFileWriter.**close** ()		14928 ms (0,1%)
module.fileManagement.services.filedeltas.rabinpoly.Window.**getByteBuffer** ()		9195 ms (0,1%)

Fig. 35. Overall CPU time used for each Java method evoked

Results discussion

Based on the collected results, the system does not scale linearly, which would be a quite good result. The main bottleneck in this operation is how repository handles files. When the repository retrieves the contents of a file, it gives us a continuous byte stream. But due to the nature of delta encoding operations, when a block is missing in a file, the algorithm needs to skip backward to copy that block. This would be impossible to achieve in a continuous stream.

For this reason it was created the *streamToFile*. When the contents of a file are retrieved from the repository, they are written to a temporary file. Then the delta operation can proceed. After the operation ends, the file is erased.

However, when under heavy load and with several concurrent accesses, it becomes the main bottleneck of the system (figure 25). When the first clients arrive, we see a high spike in the CPU activity, but as more requests are received, the CPU activity decreases, instead of keeping the high pace. The reason is that as more clients are executing the *streamToFile,* the disk available write bandwidth reaches its limit. So most of the time, the CPU is just waiting for disk. Such can be seen in figure 23.

It is possible to keep the entire file in memory, instead of writing it to the disk, but when handling large files, such as 1 GB, they could fill up the entire Java heap memory. Similarly, even if only the missing blocks were stored in memory, it would be also possible fill up all heap memory, especially when all blocks were missing. In the client, the stream conversion is unnecessary, because native files are accessible directly to the algorithm.

6 Conclusions

This dissertation introduces a solution to an application client who synchronizes user files autonomously with the IST repository. The solution designed is not final, as for the same problem there are many solutions. Still, it was shaped based on the systems studied, my own academic experience and guidance and ideas from my supervisors.

This document starts by describing the main problem around the document management systems and the motivation to solve it. Then are defined the objectives and requirements that the developed prototype must attend and demonstrate in the final solution. Still, as any IT project, the requirements are not always final. For example, since the beginning of the project it was defined that the client application should have a feature which would allow it to speed up the file transfers. As this tool would be used in an organization environment, coworkers would be working together near each other, in the same network. So a tool which also allowed peer-to-peer communication would be great, as users sharing the same files could retrieve the newest version from another user, and as he was on the local network there would be high bandwidth available. This make the system load more balanced. But if for some reason the coworkers did not worked in the same physical space or did not shared files between them, the introduced feature would not bring any advantages. So instead, it was changed to delta encoding. This feature would benefit every user in file transfers wherever they work in their workplace or from home.

After presenting the objectives, the document is followed by a presentation of existing solutions and how they work. This introduces the main systems and ideas that were relevant in the conception of the main solution. Finally, is presented the main solution and the results of its conception.

One of the most challenging parts of this project was surprisingly the cache. The interception of I/O requests is not an easy task, especially when one of the requirements that we do not want to give up is portability. The answer was to use the newest file change notification API of Java 7 called Watch Service API. This takes advantage of native FS support for file changes. When it is not available, the Watch Service will poll the FS, from time to time. However, there was yet another problem to solve. When we performed a modification in a file being watched, the FS can generate more than one event of the same type. Also as the cache uses the OS interface to allow the user to interact with files, the client does not have control how they can be accessed. So, when the client application writes a file in the cache (coming from the repository), it would see several notifications of modify events on that file. However there is no way to verify if all of them were generated by the client modification. The solution was to lock the file while it was being modified by the client. This way, the user could not modify it at same time. After 5 seconds of the file was closed, the lock was released and the user is free to make changes. Meanwhile until the release of the lock all modify events were discarded.

The final addition to the client was the delta encoder. From all the developed work, this was the feature that involved more work, due to its implementation complexity and involved study. On overall the algorithm achieved its purpose and it is possible to achieve high compression levels. For example,

given a file with 500 MB with two modifications, it is possible to transfer only 1 MB and reconstruct the entire file back. If this algorithm was not used, so in normal file transference, the average time to transfer a file with a steady connection of 200 KB/s would be of about 2622 seconds. Instead if for the same file, with same connection, we were using the delta encoding algorithm we have a total transferred size (file hashes + delta file) of 1114 KB and the total transfer time (data transference + overall algorithm time) of 21 seconds.

The worst performance of the algorithm is when it is assigned to compare two totally different files. How badly it will perform will depend on the block size chosen. They can range from 10 times more to 1/3 more of the file transfer time, as seen in tests. For this reason, it is important to balance the block size with the total file size. Smaller block sizes will bring advantages when files are similar, but when they are very different, they can decay the algorithm performance, especially in huge files. In terms of implementation, the algorithm leaves some room for improvement: the actual implementation it is not multi-threaded and does not automatically choose the best block size. Still, I believe that would be the last feature that would make the difference in the algorithm.

In conclusion, the client prototype elaborated in this dissertation meets all predefined requirements. The client authentication system is well integrated with IST CAS authentication system and keeps the single sign-on feature. Additionally as the authentication credentials are inserted in the IST authentication webpage, it offers more confidence to the user. The integration with IST repository was also successful, however it might require some future work, as the repository is still in development phase and a lot can change. Finally the introduced delta encoding algorithm can improve the file transfers, especially under normal conditions. While the worst case scenario can be discouraging, it can be mitigated by tuning the block size.

7 Bibliography

[1] T. McIndoo, “Paperless Office in Perspective,” June 2009. [Online]. Available: http://pt.scribd.com/doc/15686308/Paperless-Office-in-Perspective-A-Document-Management-System-for-Today-. [Last access 7 May 2013].

[2] Edge, “Document Management Return On Investment,” [Online]. Available: http://www.edge.com/downloads/Edge%20ROI%20Document%20Management.pdf. [Last access 7 May 2013].

[3] Dropbox, “Dropbox - Features - Simplify your life,” 2012. [Online]. Available: http://www.dropbox.com/features. [Last access 7 January 2012].

[4] J. Lenhart, “Major Dropbox Security Flaw Let Users Enter any Password,” 20 June 2011-. [Online]. Available: http://technorati.com/technology/article/major-dropbox-security-flaw-let-users/. [Last access 8 January 2012].

[5] D. Newton, “Dropbox authentication: insecure by design,” April 2011. [Online]. Available: http://dereknewton.com/2011/04/dropbox-authentication-static-host-ids/. [Last access 26 December 2011].

[6] Instituto Superior Técnico, “Bennu Framework,” 26 July 2012. [Online]. Available: https://fenix-ashes.ist.utl.pt/fenixWiki/BennuFramework. [Last access 7 May 2013].

[7] Apache Software Foundation, “Apache Maven,” 2013. [Online]. Available: http://maven.apache.org/. [Last access 7 May 2013].

[8] Instituto Superior Técnico, “Fénix Framework,” [Online]. Available: https://fenix-ashes.ist.utl.pt/trac/fenix-framework. [Last access 7 May 2013].

[9] MySQL, “The BLOB and TEXT Types,” 2013. [Online]. Available: http://dev.mysql.com/doc/refman/5.0/en/blob.html. [Last access 7 May 2013].

[10] MIT, “What is Kerberos,” 5 November 2011. [Online]. Available: http://web.mit.edu/kerberos/#what_is. [Last access 5 January 2012].

[11] S. Xiong, “Web Single Sign-On System For WRL Company,” June 2005. [Online]. Available: http://web.it.kth.se/~johanmon/theses/xiong.pdf. [Last access 7 May 2013].

[12] Jasig, “About CAS,” 2009. [Online]. Available: http://www.jasig.org/cas/about. [Last access 8 January 2012].

[13] B. Arkills, LDAP Directories Explained: An Introduction and Analysis, Addison-Wesley Professional, 2003.

[14] J. D. T. K. G. Coulouris, Distributed Systems: Concepts and Design, Pearson Education:

Addison Wesley, 2005.

[15] B. Pawlowski, C. Juszczak, P. Staubach, C. Smith, D. Lebel e D. Hitz, “CiteSeerX - NFS Version 3 - Design and Implementation,” 1994. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.35.6. [Last access 26 December 2011].

[16] Sun Microsystems, Inc., “RFC 1094 - NFS: Network File System Protocol Specification,” March 1989. [Online]. Available: http://tools.ietf.org/html/rfc1094. [Last access 25 December 2011].

[17] M. Satyanarayanan, “A Survey of Distributed File Systems,” February 1989. [Online]. Available: http://www.cs.cmu.edu/~satya/docdir/satya89survey.pdf. [Last access 20 December 2011].

[18] J. J. Kistler, “Disconnected Operation in a Distributed File System,” May 1993. [Online]. Available: http://www.sigmobile.org/phd/b494/theses/kistler.pdf. [Last access 28 November 2011].

[19] M. Satyanarayanan, “Coda: A Highly Available File System for a Distributed Workstation Environment,” 1989. [Online]. Available: http://www.cs.cmu.edu/~satya/docdir/satya-wwos2-1989.pdf. [Last access 29 November 2011].

[20] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson e B. Welch, “The sprite network operating system,” 1988. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.7525. [Last access 29 November 2011].

[21] CERN, “The Large Hadron Collider,” 2013. [Online]. Available: http://public.web.cern.ch/public/en/lhc/Facts-en.html. [Last access 7 May 2013].

[22] R. Harris, “Google's 650,000-core warehouse-size computer,” 23 October 2007. [Online]. Available: http://www.zdnet.com/blog/storage/googles-650000-core-warehouse-size-computer/213. [Last access 7 May 2013].

[23] R. Lammel, “Google’s MapReduce Programming Model - Revisited,” em *SCP*.

[24] Apache, “Welcome to Apache Hadoop,” 2013. [Online]. Available: http://hadoop.apache.org/. [Last access 7 May 2013].

[25] N. Kolakowski, “Facebook’s Corona: When Hadoop MapReduce Wasn’t Enough,” 9 November 2012. [Online]. Available: http://slashdot.org/topic/bi/facebooks-corona-when-hadoop-mapreduce-wasnt-enough/. [Last access 7 May 2013].

[26] NIST, “The NIST Definition of Cloud,” September 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf. [Last access 7 May 2013].

[27] D. Berlind, “Phishing-based breach of salesforce.com customer data is more evidence of

industry's need to act on spam.," 6 November 2007. [Online]. Available: http://www.zdnet.com/blog/berlind/phishing-based-breach-of-salesforce-com-customer-data-is-more-evidence-of-industrys-need-to-act-on-spam-now/880. [Last access 7 May 2013].

[28] N. Perlroth, "Dropbox Spam Attack Tied to Stolen Employee Password," 1 August 2012. [Online]. Available: http://bits.blogs.nytimes.com/2012/08/01/dropbox-spam-attack-tied-to-stolen-employee-password/. [Last access 7 May 2013].

[29] D. Binning, "Top five cloud computing security issues," 24 April 2009. [Online]. Available: http://www.computerweekly.com/news/2240089111/Top-five-cloud-computing-security-issues. [Last access 7 May 2013].

[30] Amazon, "Amazon Simple Storage Service (Amazon S3)," 2013. [Online]. Available: http://aws.amazon.com/s3/. [Last access 7 May 2013].

[31] W. Vogels, "Amazon's Dynamo," 2 October 2007. [Online]. Available: http://www.allthingsdistributed.com/2007/10/amazons_dynamo.html. [Last access 7 May 2013].

[32] C. Strauch, "NoSQL Databases," [Online]. Available: http://www.christof-strauch.de/nosqldbs.pdf. [Last access 7 May 2013].

[33] Dropbox, "DropboxFactSheet," [Online]. Available: https://www.dropbox.com/static/docs/DropboxFactSheet.pdf. [Last access 7 May 2013].

[34] J. Ying, "Meet the Team! (Part 1)," 5 February 2009. [Online]. Available: https://blog.dropbox.com/2009/02/meet-the-team-part-1/. [Last access 7 May 2013].

[35] D. Drager, "DropBox : Review, Invites, and 7 Questions with the Founder," 17 March 2008. [Online]. Available: http://www.makeuseof.com/tag/dropbox-review-invites-and-7-questions-with-the-founder/. [Last access 7 May 2013].

[36] C. Soghoian, "How Dropbox sacrifices user privacy for cost savings," 12 April 2011. [Online]. Available: http://paranoia.dubfire.net/2011/04/how-dropbox-sacrifices-user-privacy-for.html. [Last access 7 May 2013].

[37] A. F., "Deduplication - reports on privacy issues," [Online]. Available: https://forums.dropbox.com/topic.php?id=36365. [Last access 7 May 2013].

[38] Dropbox, "Dropbox Privacy Policy," 10 April 2013. [Online]. Available: https://www.dropbox.com/terms#privacy. [Last access 7 May 2013].

[39] Dropbox, "Does Dropbox always upload/download the entire file any time a change is made?," 2013. [Online]. Available: https://www.dropbox.com/help/8/en. [Last access 7 May 2013].

[40] Dropbox, "What is LAN sync?," 2013. [Online]. Available: https://www.dropbox.com/help/137/en. [Last access 7 May 2013].

[41] Y. Otchere, “DropBox Gets Dropkicked,” 7 May 2013. [Online]. Available: http://www.thinkmythink.com/dropbox-gets-dropkicked/. [Last access 7 May 2013].

[42] Google, “Google Drive,” 2013. [Online]. Available: https://www.google.com/intl/en/drive/start/index.html. [Last access 4 January 2013].

[43] W. Gordon, “File Syncing Faceoff: Dropbox vs. Google Drive,” 24 April 2012. [Online]. Available: http://lifehacker.com/5904731/desktop-file-syncing-faceoff-dropbox-vs-google-drive. [Last access 7 May 2013].

[44] (. N. Garnier, “Download only binary differences to the client with Google Drive SDK,” 25 April 2012. [Online]. Available: http://stackoverflow.com/questions/10312880/download-only-binary-differences-to-the-client-with-google-drive-sdk. [Last access 7 May 2013].

[45] “LZW Data Compression,” 1 October 1989. [Online]. Available: http://marknelson.us/1989/10/01/lzw-data-compression/. [Last access 5 December 2011].

[46] D. Huffman, “A Method for the Construction of Minimum-Redundancy Codes,” September 1952. [Online]. Available: http://compression.ru/download/articles/huff/huffman_1952_minimum-redundancy-codes.pdf. [Last access 5 December 2011].

[47] A. Tridgell e P. Mackerras, “The rsync algorithm,” June 1996. [Online]. Available: http://cs.anu.edu.au/techreports/1996/TR-CS-96-05.pdf. [Last access 7 December 2011].

[48] J. Jenkov, “RSync - Remote Synchronization Protocol,” [Online]. Available: http://tutorials.jenkov.com/rsync/index.html. [Last access 7 May 2013].

[49] D. Price, “CVS - Concurrent Versions System,” 3 December 2006. [Online]. Available: http://cvs.nongnu.org/. [Last access 7 December 2011].

[50] A. Muthitacharoen, B. Chen e D. Mazières, “A Low-bandwidth Network File System,” 2001. [Online]. Available: http://pdos.csail.mit.edu/papers/lbfs:sosp01/lbfs.pdf. [Last access 7 December 2011].

[51] Microsoft, “Distributed File System,” 11 September 2007. [Online]. Available: http://technet.microsoft.com/en-us/library/cc753479(v=ws.10).aspx. [Last access 7 May 2013].

[52] Microsoft, “How DFS Works,” 23 March 2003. [Online]. Available: http://technet.microsoft.com/en-us/library/cc782417(v=ws.10).aspx. [Last access 7 May 2013].

[53] CramSession, “DFS: When, Why, and How,” [Online]. Available: http://web.archive.org/web/20050825131148/http://www.cramsession.com/articles/get-article.asp?aid=349. [Last access 7 May 2013].

[54] Microsoft, “About Remote Differential Compression,” 26 October 2012. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa372948(v=vs.85).aspx. [Last access 7 May 2013].

[55] Microsoft, “File and Storage Services Overview,” 29 February 2012. [Online]. Available: http://technet.microsoft.com/en-us/library/hh831487.aspx. [Last access 7 Mat 2013].

[56] Oracle, “Interface WatchService,” 2013. [Online]. Available: http://docs.oracle.com/javase/7/docs/api/java/nio/file/WatchService.html. [Last access 7 May 2013].

[57] Microsoft, “Change Journals,” 16 April 2013. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/aa363798(v=vs.85).aspx. [Last access 7 May 2013].

[58] Microsoft, “How NTFS Works,” 28 March 2003. [Online]. Available: http://technet.microsoft.com/en-us/library/cc781134(WS.10).aspx. [Last access 7 May 2013].

[59] Linuxtopia, “What is the ext3 filesystem?,” 2013. [Online]. Available: http://www.linuxtopia.org/HowToGuides/ext3JournalingFilesystem.html. [Last access 7 May 2013].

[60] Apple, “Mac OS X: About file system journaling,” 2013. [Online]. Available: http://support.apple.com/kb/HT2355. [Last access 7 May 2013].

[61] Jersey, “Jersey,” 2013. [Online]. Available: http://jersey.java.net/. [Last access 7 May 2013].

[62] L. Miňo e R. Bartuš, “SHA-1 collision found,” [Online]. Available: http://bezadis.ics.upjs.sk/old/CryptoSymposium/files/paper10.pdf. [Last access 7 May 2013].

[63] L. E. P. Malère, “Authentication using LDAP,” [Online]. Available: http://tldp.org/HOWTO/LDAP-HOWTO/authentication.html. [Last access 8 January 2012].

[64] M. Satyanarayanan, “Scalable, Secure, and Highly Available Distributed File Access,” 5 1990. [Online]. Available: http://www.cs.cmu.edu/~coda/docdir/scalable90.pdf. [Last access 18 December 2011].

[65] M. Mathieu e E. Capozzoli, “THE PAPERLESS OFFICE: ACCEPTING DIGITIZED DATA,” 2002. [Online]. Available: http://www.cabinetpaperless.com/wp-content/uploads/2012/09/Paperless-Office-Troy-State.pdf. [Last access 2 June 2012].

[66] Knowledgeone Corporation, “IMPLEMENTING ELECTRONIC DOCUMENT MANAGEMENT WITH KNOWLEDGEONE CORPORATION,” 17 October 2005. [Online]. Available: http://www.knowledgeonecorp.com/news/pdfs/Implementing%20Electronic%20Document%20Management.pdf. [Last access 1 June 2012].

[67] S. Deuby e T. Daniels, “Dfs: A Logical View of Physical Resources,” 1 December 1996. [Online]. Available: http://www.windowsitpro.com/article/windows-client/dfs-a-logical-view-of-physical-resources. [Last access 7 May 2013].

[68] Microsoft, “Data Access and Storage,” 19 November 2012. [Online]. Available:

http://msdn.microsoft.com/en-us/library/windows/desktop/ee663264(v=vs.85).aspx. [Last access 7 May 2013].