BennuFC, a Distributed System for Document Management

Pedro Miguel Afonso Completo Bento
Instituto Superior Técnico
Lisbon, Portugal

ABSTRACT

Document management systems are crucial in any organization. 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.

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 will be developed a delta encoder in order to ease the bottleneck constraints often imposed by the network.

1 INTRODUCTION

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.

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 access 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. An alternative to web interface for sharing data files 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. This can be accomplished with the development of a client system that keeps a local copy 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 in the workgroup will receive the update. Success uses of this approach are Dropbox and Google Drive, for example.

This thesis creates a solution to implement a client which can synchronize files among various users in a transparent way and tries to go a little further studying and implementing a way to shorten file transferences.

2 RELATED WORK

2.1 IST document management repository

This client is designed to work along with the IST document management repository. This repository is integrated with the IST Bennu Framework [1] and it serves as an interface to access the repository file system. So far the only graphical user interface it provides is a web interface where after the user is authenticated, he can manage his repository data. 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 core features of this framework reside in standalone independent modules that are joined together with maven [2]. Among the various modules in Bennu's core features, there are two which are the most import for this work: the Authentication, which provides an API to communicate with the IST Central Authentication Service and the File Storage module, who keeps the server file system. This framework stacks over the Fenix Framework [3] which allows the

development of Java-based applications that need a transactional and persistent domain model.

The File Storage module is responsible for maintain an abstract File System over any file system. To keep the persistence the repository's structure is modeled in the Fenix Framework as a SQL schema. Therefore the file system internal structure can be viewed as a SQL schema:

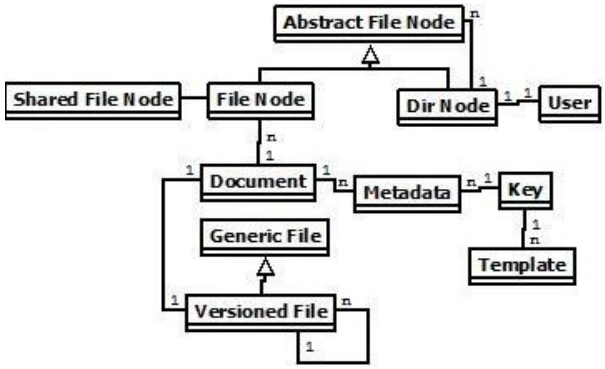

Fig. 1. File Storage Internal Structure

2.2 Distributed File systems

The idea of creating a client which synchronized files remotely is not new. This problem has been, and it is still being addressed in several systems distributed file systems, according to the hardware challenges existing in the time that they were developed.

The **Network File System** (NFS) [4] [5] uses a client-server paradigm to allow multiple distributed clients to shared access to files that either a server or a client can export from his file systems or subdirectories. 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. But as the servers were stateless, the client never knew when the file was updated, which lead to inconsistencies.

Later the **Andrew File System** (AFS) [6] tried to address by creating stateful servers and solve the concurrency problem using callbacks. Its objective is to record what a client has cached and while a 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. The only problem was when two clients closed the same file, at same time, because in this case, none of them will notify the other, resulting in unpredictable results.

While the two previous distributed file system clients offered interfaces to communicate and synchronize files with their repositories, none of them could keep the availability of the system when disconnected from the network. The **Coda File System** [6] [7] implements AFS-2 focusing on its best features: scalability, performance and security, and additionally high availability. This is accomplished by two of the main features in Coda: server replication and support for disconnected operation. This system uses hoarding as a way to mitigate the problems during the disconnection. While disconnected, it uses the files cached to serve the file requests, so a user is limited to access only the previously cached files. Lastly, on reconnection, it has to propagate the updates to the Coda servers.

Although of its great ideas, this file system was plagued by bugs, bad performance and it was never applied in real world applications.

Today cloud services have become more and more popular. Great examples of such systems are **Dropbox** [8] and **Google Drive** [9]. What makes Dropbox so popular is the client application which combines a set of unusual features: automatic synchronization, versioning of files, delta encoding and web-interface. 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 to manage files. 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. Dropbox also allows the data to be modified offline and resynchronized later. This system also optimized both the data transferences and data storage by executing a "binary-diff" [10] which will mark the portions of the data that have been modified and only those changes are transferred to the server-side letting this way to keep the low bandwidth usage, especially in large files. The data storage is also improved by using a deduplication algorithm [11], which when uploading a file, 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.

Google Drive [9] 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. 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 users.

2.3 Efficient data transfer

A crucial factor which can quite influence the user experience, especially in low bandwidth networks is the length in file transferences. This however can be lessened by using a set of techniques.

Compression can reduce the size of the transference by eliminating the redundant or duplicated data. This can be achieved by using a dictionary approach LZW [12] or a statistical based approach, Huffman coding [13].

Chunks and Hashing is an alternative to compression by 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. So exploring these similarities we are able 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. In a file transfer **Rsync** [14] analyses files by splitting it in a fixed size blocks and for each it calculates two checksums: a weak "rolling" 32-bit checksum and a stronger MD5 checksum. They are sent to the receiver and by comparing the hashes with the file, the receiver decides which blocks are needed and only those are transferred.

LBFS [15] and **Microsoft DFS** [16] [17] compute the differences in files by dividing it 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. 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.

Microsoft also improved the storage services by using data deduplication [18]. This method 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.

3 ARCHITECTURE

The objective of this work is to develop and implement a client which can synchronizes autonomously 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. 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, even if the change was small, in a normal file transfer it implies sending the entire content of the file to the repository. This work also aims to develop a method to synchronize files by exploiting the similarities between different versions of the file. To conceive this solution will be used an approach based on delta encoding using a static block size analysis.

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 next figure will present the architecture of the solution.

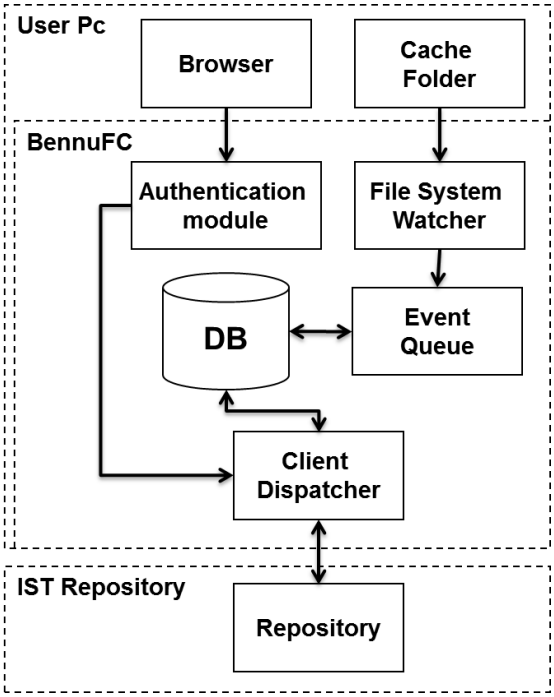

Fig. 2. System architecture

Authentication module. One of the requirements of this project was to use the same authentication services that IST provides. 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 Central Authentication Service (CAS) [19] was the chosen authentication service. Still, so far this protocol has only been used in the authentication between the user's web browser and IST web applications.

The solution was developed 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 browser will search on the Windows Registry for a previously registered application associated with that protocol. If it founds any, that application is executed using the browser link as the argument.

The solution is as simple as this, as long as there is no client running at the moment of the login. Otherwise, it has the main problem that every time it 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. The solution was accomplished using a common Inter-Process Communication method: TCP Sockets. 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. In this case, the newly created client will deliver the CAS ticket to the older client using that already existent TCP socket and then exits.

Cache. The cache concept used in this project is very similar to same concept used in remote communications, if the requested data is available locally, then it can be retrieved immediately without further communication with the repository. Otherwise it would increase its access time.

Physically, 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 can be read or written by the user's applications or by the repository, through the client application. This way is possible to take advantage of OS functionalities and features, such as search and keep the OS interfaces to access the cache, which are more familiar to the user. The success of this component is, however, highly related to the storage space available. The larger the information we hold in cache, the faster it is its access, but it comes at cost of storage space. Also by having all user data in the cache will insure that if the client cannot access the online repository, the user can still access to all files he had stored on his remote space.

File System Watcher. 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 was created the File System Watcher (FSW). This component is responsible for the connection between the local cache and the client application. 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. The solution developed was to monitor the local cache using a hybrid algorithm of 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. But since the client cannot receive notifications that occurred before it was started, it will run the extensive file search, where it will check the previous last modified date that the client knows of with the actual "last modified date" of the file. If they are different, the file was modified.

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 use a local database. Its goal is to mirror the local file system, but with additional metadata information about synchronization state of each file and directory. In order to keep the database, the Event Queue receives events generated by the FSW, and they are processed and stored in the database. 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. These events will later be used to execute the modifications at the remote repository.

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. 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 execute it.

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 the remote repository or to the repository.

The final stage is about executing the operations that were fetched in the first stage and transfer the files needed from repository and to the repository. The transference 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. So to take advantage of this similarity, the encoder analyzes both files and transfers only the differences between them.

Delta Encoder. 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 transferences. One way to achieve this reduction is by exploiting the similarities between the files. If a file has already been transferred previously to the remote repository, then it is possible to exploit the file similarities by only transferring the changes.

To solve this problem, we implemented a file transfer protocol based on Rsync algorithm [14]. The aim of this algorithm is to allow the transformation of Data1 in Data2 by sending only the minimum information possible over the network.

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)

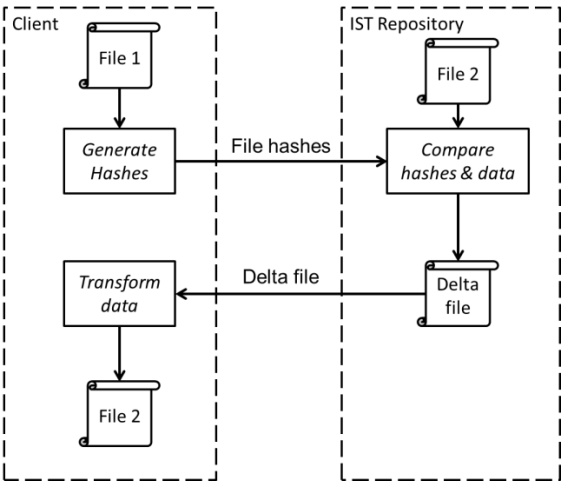

Fig. 3. Delta encoding process

Step1: 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 (block file hashes) to the repository, over the network.

Step2: 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, the binary file, while the other references the structure of the resulting file, i.e. instructions to transform the File 1 into File 2. This file is named instructions file. For example, it can contain information about where to put the new contents or leave the existing ones. These two files are merged in a unique file resulting in a delta file.

Step3: 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 local File 1 with the delta file that came from the repository, the client reconstructs the File 2.

4 IMPLEMENTATION

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. 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 WebServices 1.7 [20]. The client was implemented and tested in Windows 7.

The delta encoder uses Rabin Fingerprints, whose purpose is to do a fast comparison between blocks. What makes it faster than others is his rolling hash property. This hashing algorithm is backed up by SHA1 which offers a considerable resistance to collisions, relatively fast computing and hashes of 160 bits.

5 EVALUATION

To test the application, we created various test scenarios to determine if the overall behavior of the client was according to the defined functional requirements of the project. These are defined and explained in the main dissertation document.

Secondly, we tested the performance of delta encoding algorithm. The tests made will cover various scenarios and will check the gains in data transfer using the delta encoder algorithm versus a normal file transfer. Also it will present the results of delta encoder algorithm using different block sizes.

The overall tested 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.

5.1 Results

The delta encoder algorithm test was performed by creating 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 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 the lengthier will be the comparison between the hashes and the second file. To evaluate how these variables affect the algorithm's performance, special conditions were simulated to emulate the best, normal and worst case scenarios.

5.1.1 Test setup

The file sizes used in tests were: 16 KB, 1 MB, 5 MB, 20 MB, 52 MB, 100 MB and 500 MB.

The block sizes tested were: 2 KB, 4 KB, 8 KB, 16 KB, 524 KB and 1 MB.

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 (delta file + block hashes), 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 block hashes file plus the delta file transferred.

5.1.2 Best case scenario

Objective

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

Results

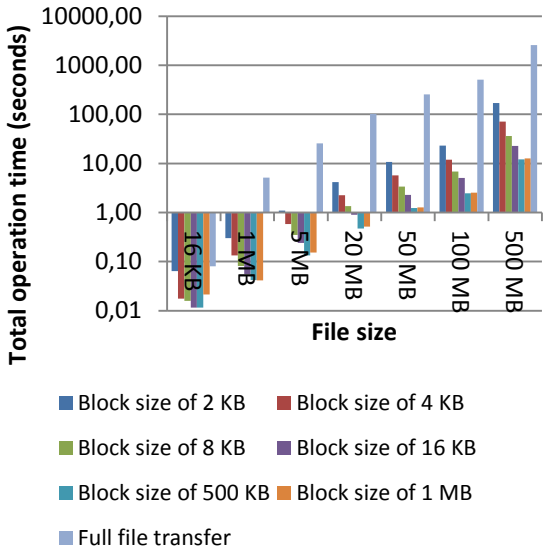

Fig. 4. Performance tests when transferring each file through delta encoding, using different block sizes

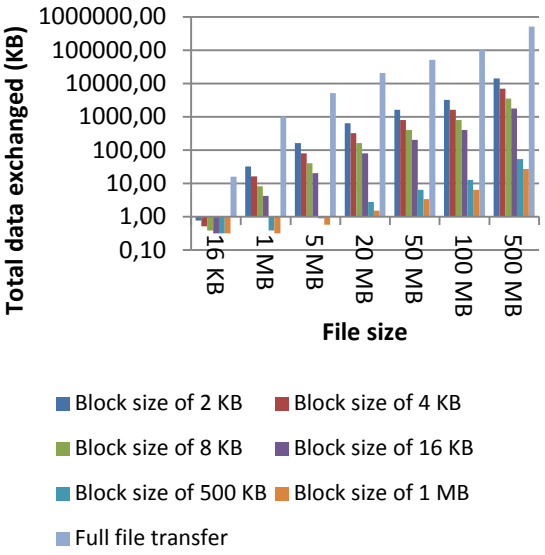

Fig. 5. Total data exchanged when transferring each file through delta encoding, using different block sizes

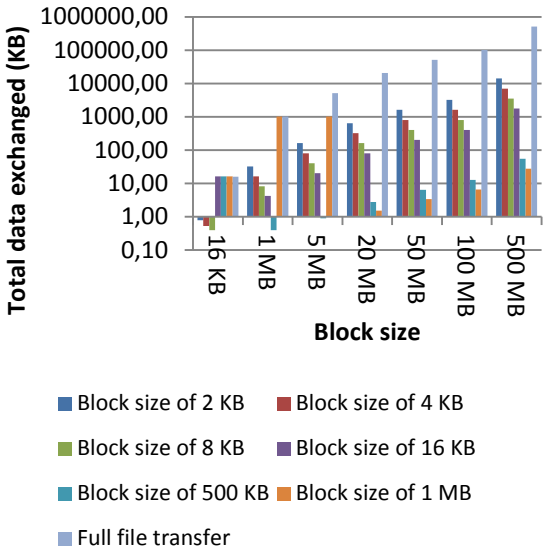

Fig. 7. Total data exchanged when transferring each file through delta encoding, using different block sizes

Remarks

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. 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.

5.1.3 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.

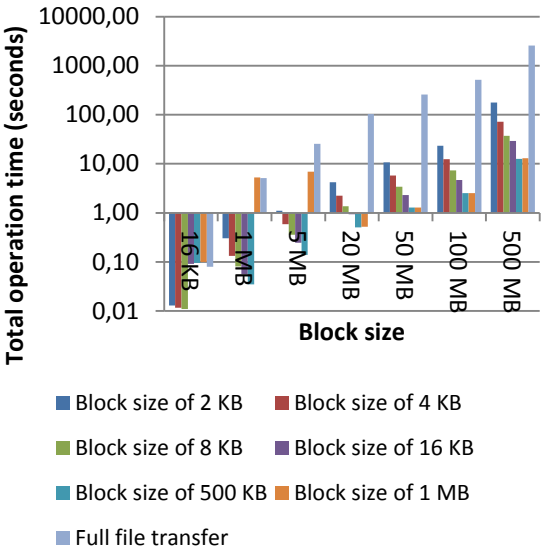

Fig. 6. Performance tests when transferring each file through delta encoding, using different block sizes

Remarks

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. 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.

5.1.4 Normal case: file with 2 modifications

Objective

This case corresponds to the case where the user made two modifications in the original file.

The main difference between this scenario and the previous is that, previously the blocks were shifted by two bytes, but were intact. So, as the hash moves along the contents of the file, they will be found eventually. Here the contents of the block were actually changed. This way the algorithm will never find the differences, because none of the hashes will match.

The same would happen, if instead a modification, there was a deletion of in two points of the file. The algorithm would not recognize any of the changed blocks and mark them as new.

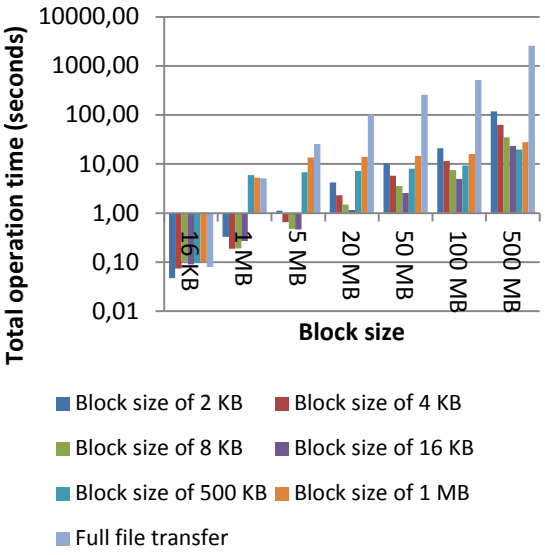

Fig. 8. Performance tests when transferring each file through delta encoding, using different block sizes

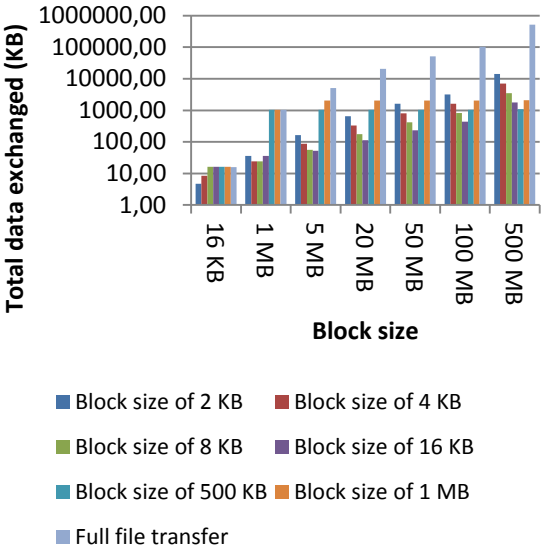

Fig. 9. Total data exchanged when transferring each file through delta encoding, using different block sizes

Remarks:

In this test, just as in previous, the algorithm will start by computing the Rabin fingerprint between the 1st byte and 2048th byte. While in the previous case the contents of the blocks were intact, which generate known block hashes, in this test, the actual contents of the blocks were modified. For this reason, those blocks will not generate known block hashes. This way the modified blocks will be merged in the binary file, while the others are referenced in instruction file.

5.1.5 Worst case scenario

Objective:

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

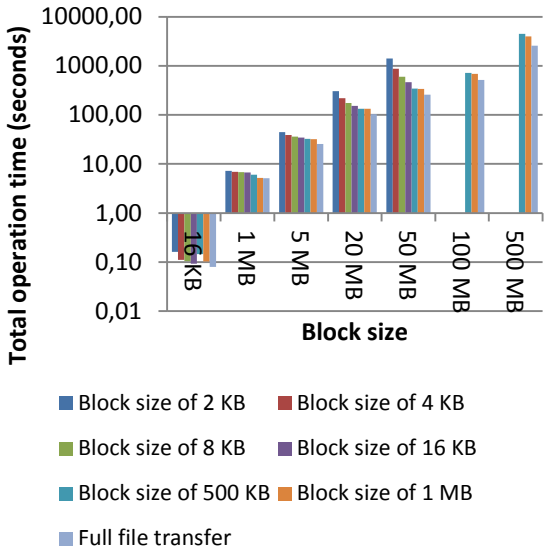

Fig. 10. Performance tests when transferring each file through delta encoding, using different block sizes

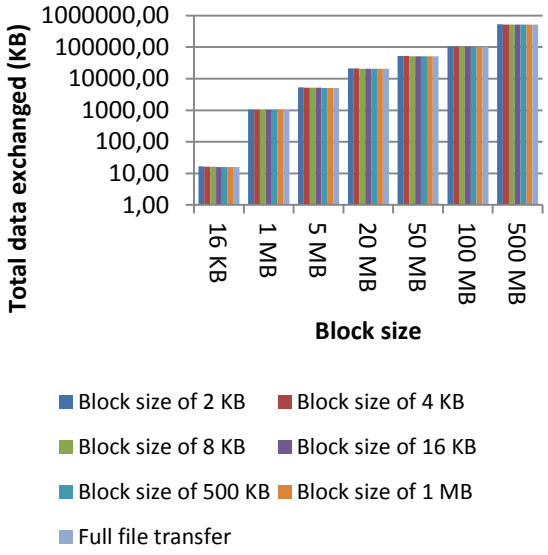

Fig. 11. Total data exchanged when transferring each file through delta encoding, using different block sizes

Remarks:

This case is shows the scenario where there are no matching blocks, because the content of all blocks was changed. As so, it will calculate the hashes from byte to byte until finds a known block hash, but as it will never find one, the algorithm just calculate the hash of every byte.

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, the 50 MB file with block size of 16384 bytes, from the total operation time of 208 seconds (excluding transfer time), 131 seconds were spent on block comparison.

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.

6 CONCLUSION

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.

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, but it does not report who made the modification. This made the monitor even more complex, because the FS can generate more than one event of the same type. 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. On overall the algorithm achieved its purpose and 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. 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. For this reason 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 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] Instituto Superior Técnico, "Bennu Framework," 26 July 2012. [Online]. Available: https://fenix-ashes.ist.utl.pt/fenixWiki/BennuFramework. [Last access 7 May 2013].

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

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

[4] 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].

[5] 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].

[6] 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].

[7] 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].

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

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

[10] 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].

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

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

[13] 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].

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

[15] 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].

[16] 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].

[17] 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].

[18] 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].

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

[20] Jersey, "Jersey," 2013. [Online]. Available: http://jersey.java.net/. [Last access 7 May 2013].

[21] 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].