



INSTITUTO SUPERIOR TÉCNICO
Universidade Técnica de Lisboa

VFCbox: A Multi-user System For Consistent File Sharing

Jean-Pierre Ramos

jean-pierre.ramos@ist.utl.pt

Dissertação para obtenção do Grau de Mestre em
Engenharia Informática e de Computadores

Júri

Presidente: Prof. Dr. Mário Rui Fonseca Gomes

Orientador: Prof. Dr. Paulo Jorge Pires Ferreira

Co-orientador: Prof. Dr. Luís Antunes Veiga

Vogal: Prof. Dr. José Luís Brinquete Borbinha

2011

Acknowledgements

I would like to thank Prof. Dr. Paulo Ferreira and Prof. Dr. Luís Veiga, my thesis advisors, for the total support, motivation and orientation throughout the elaboration of this work. Their orientation and advice contributed greatly for the quality of this thesis.

To my MSc colleagues, Edgar Rocha, João Ribeiro and João Lemos I want to express my gratitude for all the support, motivation and help provided through my BSc and MSc.

I would like to thank my parents for all the support and help. I owe them my success.

I would like to thank my girlfriend, Susana Correia, for all the motivation and companionship during my BSc and MSc. I will never forget all the help and support.

Resumo

O surgimento de sistemas de partilha de ficheiros na nuvem tem sido motivado pelas necessidades reais dos utilizadores para partilhar dados. Existem muitas soluções de apoio à partilha de dados, tendo todos o objetivo comum de ser amplamente escalável, enquanto garantindo aos utilizadores uma partilha coerente dos dados partilhados. No entanto, a oferta de coerência de dados está em desacordo com a escalabilidade, uma vez que requer muitas mensagens e largura de banda disponível para transferência de ficheiros.

A largura de banda de rede pode ser minimizada através de várias técnicas, tais como a deduplicação, a compressão, o delta-encoding, etc. No entanto, estas abordagens não têm em conta que nem todos os ficheiros necessitam uma coerência total em todos os momentos para todos os utilizadores.

Neste trabalho, melhoramos a escalabilidade de um sistema de partilha de ficheiros na nuvem, chamado VFCbox, tendo em conta a noção de interesses dos utilizadores. Por outras palavras, o VFCbox gere a coerência dos dados dos utilizadores em relação às necessidades comuns sobre os ficheiros, de forma a evitar o envio de dados inúteis pela rede. De facto, alguns ficheiros não necessitam de ser constantemente propagados para todos os utilizadores, já que alguns deles não exigem tal imediatismo dada a semântica particular dos dados partilhados.

O VFCbox utiliza técnicas não só de deduplicação para minimizar a utilização da rede, mas também um modelo de coerência de dados que tem em conta os interesses dos utilizadores. O resultado é um sistema de partilha de ficheiros na nuvem escalável e eficiente que providencia aos utilizadores uma partilha coerente de dados de acordo com as suas necessidades.

Abstract

The emerging of cloud file sharing systems has been motivated by real user needs to share data. There are many solutions providing such sharing support all having the common goal of being widely scalable while providing users with consistent shared data. However, offering consistent data is at odds with scalability as it requires many messages and available network bandwidth for file transfer.

Network bandwidth can be minimized using several techniques such as compression, deduplication, delta encoding, etc. However, these approaches do not take into account that not all files must be fully consistent at all times for all users.

In this paper we further increase the scalability of a cloud file sharing system, called VFCbox, by taking into account the notion of users interest. In other words, VFCbox considers users consistency needs regarding shared files, to avoid sending useless data through the network. As a matter of fact, some files do not need to be constantly propagated to all users, because some of them do not require such immediacy given the particular semantics of the shared data.

VFCbox uses not only deduplication techniques to minimize network usage but also a consistency model that takes into account the interests of users. The result is a scalable and efficient cloud file sharing system that fulfills users needs regarding data sharing.

Palavras Chave

Keywords

Palavras Chave

Partilha de Ficheiros

Armazenamento Eficiente

Sincronização Eficiente

Deduplicação

Replicação Optimista

Gestão de Interesses

Computação na Nuvem

Keywords

File Sharing

Efficient Storage

Efficient Synchronization

Data Deduplication

Optimistic Replication

Interest Management (locality-awareness)

Cloud Computing

Contents

1	Introduction	19
2	Related work	23
2.1	Data Redundancy	23
2.1.1	Delta-Encoding	24
2.1.2	Compare-by-hash	24
2.1.3	Version-Based Deduplication	27
2.1.4	Deduplication Techniques Summary	28
2.1.5	Deduplication Timing	28
2.1.6	Deduplication Placement	29
2.2	Consistency in Distributed Systems	30
2.2.1	Limitations of Pessimistic Replication	30
2.2.2	Introduction to Optimistic Replication	30
2.3	Adaptive Data Consistency	33
2.3.1	TACT - A Consistency Model for Replicated Services	33
2.3.2	Locality-awareness in Large-scale Systems	33
2.3.3	Vector-Field Consistency (VFC)	34
2.4	Cloud Computing	35
2.4.1	Cloud Storage	35
2.5	Relevant Systems	36
2.5.1	VFC for Cooperative Work	36
2.5.2	Amazon S3 (Simple Storage Service)	36
2.5.3	Dropbox	37
2.5.4	Microsoft Live SkyDrive	37
2.5.5	Haddock-FS	37
2.5.6	LBFS	38
2.5.7	redFS	39
2.5.8	Semantic-chunks	39
2.5.9	ShiftBack	39
2.5.10	Subversion (SVN)	41
2.6	Summary	41
3	Architecture	43
3.1	Baseline Architecture	43
3.2	Architectural Decisions	45
3.2.1	Data Deduplication	45
3.2.2	Replication Model	46
3.2.3	VFCbox Main Components	47
3.2.4	Upload Pipeline	49
3.2.5	Download Pipeline	51

3.3	Data File Representation (Intermediate Format)	53
3.4	Deduplication Architecture	54
3.4.1	Deduplication Process	54
3.4.2	Deduplication Modules	55
3.4.3	Data Transfer Protocol	56
3.4.4	Chunks Repository	57
3.4.5	References Table	57
3.5	Data Compression	58
3.6	Consistency Model Architecture	59
3.7	VFC Enforcement	63
3.7.1	Consistency Level Assignment	63
3.7.2	VFC Limits	63
3.7.3	Checking VFC Limits	64
4	Implementation	65
4.1	Client	65
4.1.1	Content-based (variable-size) Chunking	65
4.1.2	Interception of Updates	66
4.1.3	Shell Extension Menu Handler	66
4.1.4	Interests Uploading Interfaces	67
4.2	Server	68
5	Evaluation	71
5.1	Qualitative Evaluation	71
5.1.1	Evaluation of File Conflicts Occurrence/Resolution	71
5.1.2	Data Latency Control	73
5.1.3	Summary	73
5.2	Quantitative Evaluation	74
5.2.1	Experimental Setting	74
5.2.2	Compared Solutions	74
5.2.3	Workload Description	75
5.2.4	VFC Limits	75
5.2.5	Bandwidth Usage Analysis: File Download Stress Test	75
5.2.6	Bandwidth Usage Analysis: File Update Stress Test	76
5.2.7	Bandwidth Usage Analysis: Deduplication Stress Test	78
5.2.8	Bandwidth Usage Analysis: Consistency Model Stress Test	79
5.2.9	Bandwidth Usage Analysis: Global System Stress Test	81
5.2.10	Bandwidth Usage Analysis: Real Workload Stress Test	83
5.2.11	Summary	85
6	Conclusion	87
6.1	Future Work	88
7	Appendix	93

List of Figures

2.1	Example of the chunk transference protocol.	25
2.2	Example of application of FBH. The gray color identifies the chunks that were considered as new chunks.	26
2.3	Example of application of VBH. The gray color identifies the chunks that were considered as new chunks.	26
2.4	Consistency zones centered on a pivot.	34
2.5	Example of deduplication process.	40
3.1	VFCbox's main overview.	43
3.2	A file divided in zones defining multiple consistency zones. This example illustrates an interface in which users may specify their interests overs parts of a file.	44
3.3	VFCbox interface in which users may specify special interests over files of a shared folder. . .	45
3.4	A file divided in zones defining multiple consistency zones (semantic zones). This example illustrates how VFC was adapted to documents.	47
3.5	Client Architecture.	48
3.6	Server Architecture.	48
3.7	VFCbox's data upload pipeline: modules of the baseline architecture that are involved in the data upload.	49
3.8	VFCbox's data download pipeline. The figure illustrates the modules of the baseline architecture that are involved in the data download.	51
3.9	Example of a Latex document translated to the intermediate format.	53
3.10	Example of deduplication process.	54
3.11	Deduplication Client Architecture.	55
3.12	Deduplication Server Architecture.	55
3.13	Example of the compact format of a file after deduplication.	56
3.14	Example of the transfer of a file. This example illustrates the transfer of the compact format of the file exemplified in Figure 3.13. The black boxes represent literal data. The gray arrows represent references to the actual chunks of data.	56
3.15	Example of the contents of multiple files. The internal representation of each file is constituted by a linked-list of references to the chunk's hashes.	57
3.16	Example of the references table on the server side. The example represents that the Client 1 has references to the hash values H1, H2 and H3, which means that the data chunks with those hash values have already been sent to the concerning client.	58
3.17	VFCbox interface, in which users may specify special interests over certain files of a shared folder.	59
3.18	A Latex file divided in chapters and sections defining multiple consistency zones. This example illustrates an interface in which users specify their interests overs parts of a Latex file. . . .	60
3.19	Example of a file format to VFCbox. A file is composed by a list of zones and each zone is composed by a list of references to chunks.	61

3.20	Example of two clients trying to update to the server two different zones of the same file. The underlined version stamps represent the new updates. The version vector at the top of each file represents the version of the document structure.	62
3.21	Example of two clients after updating to the server two different zones of the same file. This exemplifies the result obtained by example represented in Figure 3.20. The underlined version stamps represent the new updates (already synchronized to the server). The version vector at the top of each file represents the version of the document structure.	62
4.1	Right click menu on windows explorer exposing the VFCbox menu. In this case users may decide to share the current folder with a certain user or to upload some interests over certain files of the current folder.	66
4.2	VFCbox interface, in which users may specify special interests over certain files of a shared folder.	67
4.3	Example of the assigned consistency levels when a section of a Latex document is chosen as pivot zone.	68
4.4	Example of the assigned consistency levels when a section of a Latex document is chosen as pivot zone.	68
5.1	Two users modify the same document, but since they are modifying different sections of the document, it is not considered a conflict and the modifications are naturally merged.	72
5.2	Resolve conflict - step 1: select the file to resolve the conflict.	72
5.3	Resolve conflict - step 2: chose to open WinMerge to resolve the conflict or mark the current version as the resolved version.	73
5.4	Resolve conflict - step 3: use WinMerge to resolve the conflict.	73
5.5	Bandwidth usage (in KB) of downloading 5 different file samples.	75
5.6	Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the beginning of each file.	76
5.7	Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the middle of each file.	77
5.8	Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the end of each file.	77
5.9	Dropbox's bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy. . . .	78
5.10	VFCbox's bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.	78
5.11	SVN's bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.	78
5.12	LBFS's bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.	78
5.13	Average gain percentage of bandwidth usage to download sets of samples with different redundancy percentages.	79
5.14	Dropbox's bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.	80
5.15	VFCbox's bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.	80
5.16	SVN's bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.	80
5.17	LBFS's bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.	80
5.18	Total used bandwidth (in MB) to download 10 file updates of 5 different file samples. It represents the sum of the used bandwidth of the stress test illustrated in Figures 5.14, 5.15, 5.16 and 5.17.	81

5.19	Bandwidth usage (in MB) of downloading 10 file updates of a set of 20 different file samples (10MB each file). The file updates were composed by a total replacement of the content of each file. These set of files have in common the same introduction, which produces a total of 33% of redundancy.	82
5.20	Bandwidth usage (in MB) sum of the stress test illustrated in Figure 5.19.	83
5.21	Bandwidth usage (in KB) of downloading 10 file updates of a set of 10 different thesis files. The file updates were composed by small changes on each chapter of each file.	84
5.22	Bandwidth usage (in KB) sum of the stress test illustrated in Figure 5.21.	85
7.1	Table results of the evaluation test illustrated in Figure 5.5.	93
7.2	Table results of the evaluation test illustrated in Figure 5.6.	93
7.3	Table results of the evaluation test illustrated in Figure 5.7.	94
7.4	Table results of the evaluation test illustrated in Figure 5.8.	94
7.5	Table results of the evaluation test illustrated in Figure 5.9.	95
7.6	Table results of the evaluation test illustrated in Figure 5.10.	95
7.7	Table results of the evaluation test illustrated in Figure 5.11.	96
7.8	Table results of the evaluation test illustrated in Figure 5.12.	96
7.9	Table results of the evaluation test illustrated in Figure 5.14.	97
7.10	Table results of the evaluation test illustrated in Figure 5.15.	97
7.11	Table results of the evaluation test illustrated in Figure 5.16.	98
7.12	Table results of the evaluation test illustrated in Figure 5.17.	98
7.13	Table results of the evaluation test illustrated in Figure 5.19.	99
7.14	Table results of the evaluation test illustrated in Figure 5.20.	99
7.15	Table results of the evaluation test illustrated in Figure 5.21.	100
7.16	Table results of the evaluation test illustrated in Figure 5.22.	100

List of Tables

2.1	Comparison between deduplication techniques.	28
2.2	Comparison between the studied systems.	41
3.1	VFC model: semantic zone limits.	64
5.1	VFC model: zone limits settings.	75

Chapter 1

Introduction

The increasing development of the ubiquitous and pervasive computing areas[44] have been creating a requirement for new and more efficient ways of sharing data. Ally to this, there are file sharing systems that foster this sharing of data and its manipulation amongst users. These types of tools are being spread for many environments, in which there are many limitations such as reduced memory and low bandwidth networks. Due to this, efficient information sharing is considered a fundamental aspect to the consistent sharing of files.

File sharing systems are becoming an emergent solution to the problem of sharing and updating data between multiple users. Therefore, and due to this emergent utilization, these systems are requiring new methods and new techniques to synchronize data in a more efficient way, specially w.r.t. bandwidth usage that is still considered a scarce resource and a bottleneck to this type of systems.

Consider for instance a team of co-workers that want to share and change a set of files. To do this, they can use a file sharing system that shares data consistently. Furthermore, users can be working with either a desktop that has a high speed connection; a laptop linked to a low bandwidth network, or even with a smartphone with an intermittent connection to the network. Additionally, we may find that many times most of the elements of the team are only interested in some files or even parts of a file. For instance, a team of co-workers writing a document may have elements that are only working in a specific part of the document, not being really interested on the rest of the document. Taking this into account. we may say that it might not be a concern to the elements of a working team, if parts of the sharing data in which they are not interested are not consistent at all times. Nevertheless, if two users are working in the same chapter of a document, they may want to ensure the consistency of that data through time.

The goal of this work was to design and build a system called **VFCbox** that efficiently manages the consistent sharing and storage of data across a multi-user network. The system is required to be efficient, regarding not only the extra overhead required for synchronization, but also the memory usage required for storage and network bandwidth. Another requirement is that the system has to be scalable to large networks and to manage large amounts of data. Ideally, the system should also be able to reduce the latency of data which is considered as interest to users and improve concurrency amongst users. To deal with large networks and avoid the problem of having a bottleneck in the network communication, VFCbox has to consume the minimum of network bandwidth, reducing communication channels where they are not fundamental as well as data to be transfered. To deal with large amounts of data and avoid an overflow of the storage site's disk capacity, this system has to reduce the amount of data stored.

To fulfill the above mentioned requirements, VFCbox has to deal with the following challenges: i) reduce the space used to store data through the use of compact forms of representing data; ii) reduce the amount of data to be transfered through compact forms of representing data to be transfered; iii) allow concurrent access to files, while preserving replica consistency; iv) ensure the correct data synchronization, while re-

ducing the use of network resources; v) deal with conflicts, supporting ways of detecting and resolving them through the merge of concurrent data updates; vi) support disconnected work.

Many of the current solutions either to synchronize or store data in distributed systems such as Semantic-chunks[40], BackupChunk [24], Venti[28] and LBFS[23], are based in similarity between data. Data similarity or data redundancy is the concept related with the fact that most of the data that we share and store is similar to other pieces. This redundancy can be detected using **data deduplication**[21] techniques that compare the similarity between portions of data. With this data similarity exploitation, we may achieve higher performance in the storage of data, eliminating redundant data, and substituting it by references to a single instance storage. Efficient synchronization may also be achieved by not sending data that is found as redundant between two sites. Nevertheless, these solutions have the problem of not reducing/postponing communications between sites, due to the inability to adapt consistency guarantees according to the needs.

Other solutions are based in **Optimistic Replication**[32], which enables the bounded divergence of data consistency. These type of solutions try making a trade-off between weaker consistency guarantees and the possibility of a higher performance on availability and access to the replica objects. Nevertheless, most of these solutions do not contemplate an efficient management of updates. Most solutions decide to take an approach to either enforce strong consistency guarantees to all updates, or non at all. Additionally, these solutions also do not cope with the requirements of low memory usage, not being able to sometimes avoid overflowing the disk's storage site.

Some solutions such as Semantic-chunks[40] and VFC for Cooperative Work [11] also take into account the **interest management**[16](or locality-awareness) of a user to filter massive volumes of data in large-scale distributed systems. These tools try to reason about the importance of each update, performing an intelligent management of updates and performing a selective scheduling based on this importance. Many times, users are only concerned with a small subset of the total sharing set of files. Nevertheless, systems waste resources by updating users with updates which are of no interest to them. Thus, the notion of interest management brings benefits by filtering data updates of low interest to users. Systems can enforce higher consistency guarantees over data subsets in which users are most concerned, and enforce low consistency guarantees over the others. This can have great impacts reducing the amount of updates and bandwidth usage.

VFCbox combines deduplication techniques with an optimistic replication schema that is capable of adapting consistency guarantees according to user's interests. Through information provided by the user, the system is able to identify the user's interest over each data set. The benefits of this interest management are dual. Firstly, data with higher interest is forwarded to users in advance, reducing its latency and helping users to receive it ahead of data with less interest. Secondly, it reduces the number of updating messages regarding data of low interest. Additionally, deduplication techniques allow the system to reduce the redundant data stored and transfered between sites, reducing space and bandwidth usage. Thus, the system aims to an efficient way of sharing data, reducing its redundancy and the amount of communications taking into account the subsets of data that are more relevant to users.

In this work we present the implemented prototype of the VFCbox solution. This prototype consists in a multi-user file sharing system capable of performing an efficient data sharing, w.r.t. bandwidth usage. To accomplish this, the system makes use of deduplication techniques to exploit the redundant data. Additionally, the system implements a consistency model that takes into account the interest of users over parts of the shared files in order to enforce multiple consistency levels. To extract the user's interests, the application makes use of an interface where users may specify their interest level over certain files or even over certain parts of a file (for instance a chapter or a section of a document).

The remainder of the document is organized as follows. Chapter 2 describes the related work and briefly presents some relevant systems. In Chapter 3 we present the architecture of our solution. Chapter 4 presents

the implementation details of the VFCbox prototype. Chapter 5 presents the methodology used to evaluate the system and an analysis over the obtained results. Finally, Chapter 6 presents the conclusions of this work and discusses some of the possible improvements that may be performed in future.

Chapter 2

Related work

This chapter presents the state-of-the-art of work suitable for the file sharing system proposed in this document. Relevant design choices are studied and existing systems are presented and discussed.

The remainder of the chapter is organized as follows.

Section 2.1 describes the related work on data redundancy, in which several data deduplication techniques used to minimize data storage and data transfer are presented.

Section 2.2 addresses the topic of optimistic replication, in which several topics about the consistency of data are presented.

Section 2.3 address the topic of adaptive consistency and presents the TACT and the VFC model, two replication models that adapt consistency guarantees according to the required.

Section 2.4 describes the inherent problems of storage systems over large-scale networks (e.g. Internet) and addresses the topic of cloud computing, a highly scalable, reliable, secure, fast and inexpensive way for storing data.

2.1 Data Redundancy

Nowadays, we may find that most computational systems have a large percentage of redundancy in stored data[21]. This redundancy is related to the fact that there is a high duplication of data parts, or a high similarity between distinct files.

For instance, a system that keeps an history of multiple file versions has a substantial redundancy between multiple versions. This is because most of file versions only append a portion of data w.r.t. previous version, or if not, only modify a confined part of it. Apart from this, there are also other situations where we may find a high similarity between data, such as a file heading that several files use or even a piece of code that is auto-generated, and so duplicated across several files.

There are two distinct types of redundancy. One is related with the redundancy between versions of the same file and is called *cross-version redundancy*. The other one is related to parts of information within a file that are similar to other parts within another file, and is called *cross-file redundancy*.

Data redundancy can be detected using **deduplication** techniques that compare the similarity between portions of data - called **chunks**. With this data similarity exploitation, higher performance in the storage of data may be achieved, by eliminating redundant data, and replacing it by references to a single instance storage of a chunk. Efficiently synchronization may also be achieved by not sending data that is found as redundant between two sites.

These techniques differ in 3 fundamental dimensions[21]: *algorithm*, *timing*, and *placement*. The choice of the algorithm depends on its efficiency in storage reduction, reconstruction time, network bandwidth usage and impact in system's resource consumption.

Current *algorithms* are divided in three main families: Delta-Encoding, Compare-by-hash and Version-Based deduplication. W.r.t. *timing*, deduplication can be performed as *Synchronous/In-Band*, *Asynchronous/Out-of-Band*, or as *Semi-Synchronous* operation. Finally, regarding the *placement* of deduplication, it can be placed at the client side or at the server side. This placement choice may affect resource utilization, for instance, client-based placement might improve the bandwidth utilization.

2.1.1 Delta-Encoding

Delta-Encoding[18, 13] is a deduplication technique that consists in the encoding of a file relatively to another. It is often used between versions of the same file, where the new version may contain a large part of the content of the previous version.

This technique compares each byte of the file (binary diff) to be compressed against another one called reference file, and calculates a delta between them. This delta contains the modifications that were made to the reference file. The goal is to transfer and store only the delta files, keeping references to the original data regions, improving the storage space and bandwidth usage (by transferring only the deltas).

To take advantage of this technique it is fulcral to detect pairs of files in which there is a high probability of existing data similarity. Applying this technique to two completely different files would end up in the storage of both files, without any storage gains. Thus, it is important to have an heuristic to help in the detection of reference files. Version control systems like CVS[10] and SVN[15] use Delta-Encoding with the naming of files as the heuristic. This heuristic explores most of the *cross-version redundancy*, since a file name is normally maintained through multiple versions.

As this technique needs to compare two files to exploit redundancy, it is only able to explore redundancy locally. This type of redundancy is called *locally trackable redundancy*[7] where data redundancy occurs only locally, and is the contrast to the *locally untrackable redundancy*[7], which exploits redundancy between two sites without the need to have both files in the same site. This is a limitation to systems that may want to explore similarities between data that came from multiple sites (*locally untrackable redundancy*).

Another limitation of this technique is the inability to detect redundancy within a set of versions. As the algorithm makes use of the comparison between two files, it is only possible to detect similarities between them, and not within a set of files.

2.1.2 Compare-by-hash

Compare-by-hash[23, 38, 12, 5] is a deduplication technique based on the comparison of hash values of each chunk of data. These hash values have to be collision resistant so that we can assume that a content of a chunk is redundant relatively to another one, only by comparing the hash values[14, 31]. If one hash is equal to another one, we may say that the content of both chunks is identical.

This algorithm is thus capable of identifying either *cross-version* or *cross-file redundancy*, only by comparing hash values. It improves data storage by detecting chunks with the same hash value (same content), eliminating duplicates and substituting them by references to a single instance storage.

Furthermore, it is also capable of exploring *locally untrackable redundancy*, and improve the bandwidth usage. This technique is able to achieve this by transferring hash values between two sites.

For instance, a site S may have a file to send to site R. Instead of sending the whole file, S could only send the hash values of the data chunks. Then, R would only have to search locally for chunks with the same hash value, and return to S the information about the data chunks that are missing - called literal data. By the end, S would only have to send to R the literal data, avoiding to send chunks that R already has (redundant data). This process is depicted in Figure 2.1.

Efficient synchronization may then be achieved by not sending data that is found as redundant between two sites. However, in terms of data transmission, this technique introduces a new round-trip to the transfer protocol and an increase in the volume of meta-data exchange.

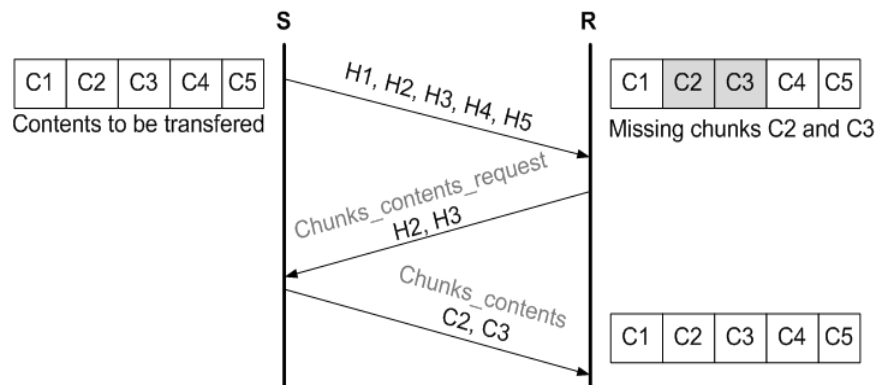


Figure 2.1: Example of the chunk transference protocol.

In terms of granularity, this technique can be separated in *Whole File Hashing*, *Fixed Block Hashing* or *Variable Block Hashing*.

Whole File Hashing (WFH):

The Whole File Hashing technique used by systems such as Single Instance Storage[8] consists on the hashing of a whole file and its comparison against others. It is the simplest technique of the compare-by-hash algorithms and it is a simple method to detect duplicated files. As it takes into account a whole file, it has not to calculate chunk boundaries, making it easier to implement and more efficient in terms of time processing.

A SHA-1[14] or MD5[31] hash of the file can then be computed and compared to the pre-existing hashes in the system, in order to identify duplicates. Nevertheless, it needs an heuristic to detect similar data blocks, for e.g. the name of files (a simple file renaming breaks this heuristic). It is also unable to detect other forms of redundancy, not detecting redundancy between file versions, or between parts of files.

Fixed Block Hashing (FBH):

The Fixed Block Hashing technique consists in detecting data redundancy through the hashing of chunks of the same size. Systems such as Venti[28] and Rsync[38] use this technique of partitioning files into chunks. When compressing a file, it has to split a file into chunks, calculating its boundaries according to a constant size (chosen *a priori*).

Afterwards, it has to calculate their respective hash values (using a SHA-1 or MD5 hash over each block), and then to search for chunks with an equivalent hash. Thus, it is able to detect redundancy in a more fine-grained way, according to the specified size block.

This is an improvement when compared to Whole File Hashing, achieving better compression rates.

However, this approach is very sensitive to modifications performed on consecutive versions of a file. Since chunk's boundaries are calculated with a fixed size, a simple insertion of data may shift all the boundaries of the blocks from that point until the end of the file. Thus, these blocks will all be considered as new data blocks. Figure 2.2 illustrates this problem of overlapping chunks. This technique may be appropriate to situations where modifications to files are only appending data. Nevertheless, it is unable to detect a high percentage of redundancy when there is a shifting of chunk's boundaries.

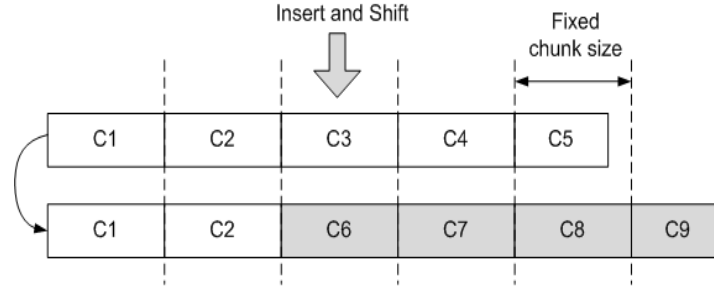


Figure 2.2: Example of application of FBH. The gray color identifies the chunks that were considered as new chunks.

Other problem inherent to this solution is to find the optimal block size, which is a non-trivial task.

Further, it depends on the type of the data. For instance, bigger blocks work better with highly redundant data. However, a smaller block size is able to find more redundancy, especially in less similar data. But then, using smaller blocks requires more meta-data and, at some point, the size of the meta-data generated does not pay the savings in space by using smaller blocks.

Variable Block Hashing (VBH):

The Variable Block Hashing technique consists in splitting data into chunks according to its content. This content-defined chunking is used in many systems such as LBFS[23], Pastiche[12], BackupChunk[24], Haddock-FS[6] and ShiftBack[39].

Instead of calculating boundaries with a fixed sized, it calculates it having the content of a file into account. Thus, it avoids the problem of shifting of boundaries (Figure 2.3), improving efficiency in data compression.

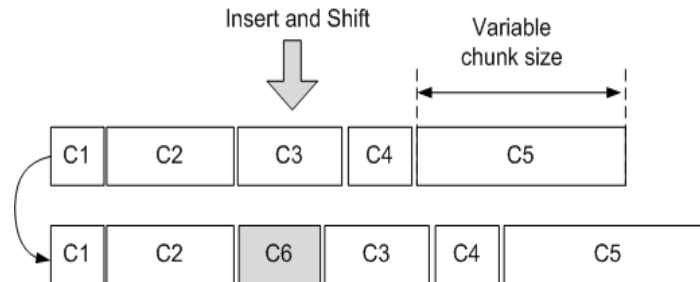


Figure 2.3: Example of application of VBH. The gray color identifies the chunks that were considered as new chunks.

To divide a file into chunks, the algorithm examines every 48-byte regions of the file and with a probability of 2^{-13} over each region's contents considers it to be the end of a data chunk. It can also be limited with a minimum and a maximum constant size (chosen *a priori*), to restrict very small or very big chunks of data. After calculating chunk's boundaries, this technique only has to calculate a SHA-1 or a MD5 hash over each chunk of data, and to compare them to the pre-existing hashes in the system, in order to identify duplicates.

Current solutions calculate the content-based boundaries using *Rabin Fingerprints*[29]. A fingerprint consists in a polynomial representation of the data modulo. When the low-order 13 bits of a region's fingerprint is equal to a chosen value, that region constitutes a boundary. *Rabin Fingerprints* is normally used because it is efficient to compute it on a file sliding window.

Comparing VBH with FBH, we may say that VBH is a more efficient algorithm to find redundant chunks of data. Nevertheless, it is a more complex and computational expensive algorithm than FBH. When choosing between VBH and FBH, one has to take into account whether the system requires an algorithm more efficient in terms of redundancy detection or in terms of computational consumption. Furthermore, in systems where the problem of sliding boundaries does not often occur, the FBH is able to find redundancy close to levels obtained by VBH.

2.1.3 Version-Based Deduplication

As described before, compare-by-hash algorithms are very powerful to detect either *cross-file* or *cross-version redundancy*. Nevertheless, and concerning data transmission, it adds a significant overhead to the data transfer protocol. It adds to it more round-trips and a substantial volume of exchanged meta-data. When in presence of a low redundancy situation that overhead may not compensate the gains.

The Version-Based Deduplication[4, 7, 39, 24] appears to reduce this overhead introduced by compare-by-hash algorithms. Version-Based Deduplication is a technique used by systems such as dedupFS[4], ShiftBack[39] and BackupChunk[24], that combines versioning information with local similarity detection algorithms. It consists in the knowledge that each site has about the data that is stored in another site. Having that knowledge into account, one can avoid to send redundant data across the network. That knowledge is based in a space-efficient representation, such as version vectors[22]. The algorithm works in 4 steps:

- A receiver site, R, informs the sender, S, of the version set that R currently stores.
- The sender site compares R's version set with its own, determining the intersection of versions that both contain in common.
- Using some local similarity detection algorithm (e.g. compare-by-hash), S determines which chunks to send to R are redundant with relation to the previous intersection.
- The sender S transfers the contents of the remaining literal chunks to R.

In order to achieve this, Version-Based deduplication imposes the unique identification of each write through the use of monotonically increasing local counters at each site. Thus, each write constitutes a version that is identified by the site's unique identifier and the the local write counter.

Each site maintains two version vectors to identify the knowledge that they have about the state of the other sites. Any site S holds a Knowledge Vector, denoted KV_S , and a Pruned Knowledge Vector, denoted PrV_S . The KV_S is a vector with an entry per known site and contains the last known state of each site, represented by the unique identification of the most recent version written by the other site and obtained by S. The PrV_S is a vector with an entry per known site and contains the last known versions that were pruned and no longer available at the site S. When S receives new data, it updates the KV_S to the value of the

most recent version obtained by each site. Thus, S is able to represent the most recent version that it has ever seen from each site. When S removes some files, it updates the PrV_S , updating it to the most recent version that has been deleted. The interval between the KV_S and the PrV_S , $[PrV_S, KV_S]$, represents the set of data that is found at the site S . Nevertheless, S has to guarantee that for every version v for each site i it can respect the clause $PrV_S[i] < v \leq KV_S[i]$. In other words, S must guarantee that the set between the PrV_S and the KV_S has no gaps. To accomplish this, S must receive only consecutive updates and discard non-consecutive updates.

This technique achieves two fundamental properties: i) as the process of similarity detection is performed locally, it can employ more data-intensive techniques; ii) on contrary to compare-by-hash it does not introduce a substantial overhead w.r.t data transmission.

Yet, as it only takes into account the knowledge of shared data, it does not detect *locally untrackable redundancy*. Depending on the situation, that issue may or may not be important. In most cases the redundancy comes mainly from *cross-version redundancy*. Thus, the gains may overcome this limitation.

2.1.4 Deduplication Techniques Summary

The following table (Table 2.1.4) presents a summary of the above mentioned deduplication techniques.

Deduplication Technique	Description	Detects redundancy within a set of files	Detects locally untrackable redundancy	Notes
Delta-Encoding[18, 13]	Compares 2 files byte by byte and creates a delta between them	No	No	Difficult to have an heuristic to detect good reference files
Compare-by-hash[23, 38, 12, 5]	Compares the hashes of file chunks	Yes	Yes	High exchange of meta-data may not compensate the gains
Version-Based Deduplication[4, 7, 39, 24]	Combines versioning info. with local redundancy detection techniques	Depends of the local redundancy detection technique	No	Reduced exchange of meta-data improves the usage of network resources

Table 2.1: Comparison between deduplication techniques.

2.1.5 Deduplication Timing

Data Deduplication Timing varies according to the time in which it is employed. It can be performed as *Synchronous/In-Band*, *Asynchronous/Out-of-Band*, or as *Semi-Synchronous* operation.

Synchronous/In-Band:

Synchronous deduplication[21] consists in performing deduplication operations when data is being consumed by the system. Furthermore, this means that for each write operation there is a deduplication operation associated, before the effective write occurrence.

This type of deduplication timing allows a search for redundancy before the actually occurrence of each write. As it never writes data into the system before compacting it, it allows a better space usage reduction.

Nevertheless, this type of deduplication timing needs to process data as it is being written. Thus, a continuous overhead in the usage of computational resources is introduced. This causes some latency on writing operations, reducing by this the throughput of the system.

Asynchronous/Out-of-Band:

On contrary to synchronous deduplication operation, asynchronous deduplication [21] consists in performing deduplication operations only periodically. From time to time, it executes a deduplication operation over the new written data, searching for redundancy and compacting the necessary data. By this, the system can perform its activities without being constraint by deduplication operations. Thus, increasing the consumption of system's data and its throughput. However, as data is written immediately, there is no processing before, allowing the writing of redundant data into the system. This requires a higher storage capacity, since it needs to stage data while uncompressed.

Normally, this type of deduplication is not a good solution when perform together with a client-based placement approach. This is firstly explained by the lack of up-to-date deduplicated meta-data at runtime, unabling to do queries immediately. Secondly, it causes a loss in the network bandwidth reduction achieved by the client-based placement.

Semi-Synchronous:

Semi-Synchronous deduplication[21] consists in a combination between synchronous and asynchronous deduplication, performing each technique when more adequate. It chooses the type of deduplication dynamically according to resource availability.

2.1.6 Deduplication Placement

Data Deduplication may be performed on the client side or on the server side according to the intended.

On the client side:

When data deduplication is performed on the client side, the client has the duty of performing deduplication operations. Thus, data is only transfered between sites after redundancy removal. As data is already sent in a compact form, this can achieve higher performances w.r.t. data transmission. Typically, this approach involves a deduplication client that communicates with a server. Thus, the client processes data before synchronizing it with the server, sending to the server only the respective meta-data. With the received meta-data the server can search for identical chunks of data. Then, it informs back the client about the missing chunks. Therefore, the client only sends to the server the literal data, decreasing costs in the usage of network bandwidth.

Despite the gains achieved in data transmission, this type of deduplication implies a higher resource usage on the client side, regarding CPU and IO operations. Taking this into account, the client could be affected in terms of performance of other applications.

On the server side:

Data deduplication performed on the server side consists in having a server appliance that executes deduplication operations on its received data. Venti[28] and Quantum¹ are examples of solutions these appliances. Normally, it is chosen having into account the benefits in the server's storage. On contrary to client-based deduplication, it does not overhead the client with the responsibility of processing data. Nevertheless, as redundant data is sent to the server, it is not so efficient in terms of data transmission.

¹Quantum: Data de-duplication overview.
<http://www.quantum.com/Solutions/datadeduplication/Index.aspx>.

2.2 Consistency in Distributed Systems

Replication is a fundamental technique in file sharing systems to improve availability, scalability, performance and to support disconnected operations. However, these systems have a difficult task ensuring replica consistency across several users. There is a lack of adaptability and efficiency on these systems regarding replica consistency. Most of them are not capable of scale to large networks, due to the huge bottleneck in data transmission. This section presents some replication models, comparing its advantages and disadvantages.

2.2.1 Limitations of Pessimistic Replication

Pessimistic Replication[32] is a model to ensure strong data consistency guarantees. It tries to guarantee that all the replicas are identical to a single copy. Further, for any sequence of read and write operations on a replicated object, it guarantees that the sequence of values associated are the same for any other replica. Thus, a sequence of reads and writes on a replicated object will produce the same effect as if the object were not replicated. To ensure data consistency at the level of a non replicated schema, this model has to block access to data whenever a replica is not up-to-date or disconnected from network. Before performing any operation request over a replica, the pessimistic replication runs a synchronous coordination protocol to ensure that the requested operation will not violate any consistency guarantee. So, it preserves consistency of data preventing conflicts, even at the cost of denying access to replicas.

This model of replication specially fits to systems where stale data cannot be read or data conflicts cannot occur. However, it comes with the cost of reducing data availability, which is a big constraint to file sharing systems. Also, it is difficult to scale systems that use pessimistic replication to larger networks. Its natural frequency of updates causes system's throughput and availability to suffer as the number of sites increases.

2.2.2 Introduction to Optimistic Replication

Optimistic replication[32] is a replication model for sharing data efficiently in wide-area or mobile environments. In opposition to pessimistically replicated systems, its approach is based on the improvement of concurrency.

It consists on the guarantee that object replicas will converge to the same value, within a certain period of time. This convergence is called eventual consistency[32, 42], and assumes that for a long period of time, all updates will eventually propagate through all the replicas. On contrary to Pessimistic Replication, this model does not block access to data even at the cost of suffering some divergence between replicas. Optimistic Replication does not have to run any type of coordination protocol before accepting an operation request. Thus, it overcomes Pessimistic Replication regarding access performance, as the replicated system no longer waits for a synchronization before accepting a request. Concerning scalability, Optimistic solutions are also preferred due to less coordination requirements and due to the possibility of running synchronization protocols in background. Thereby, it trades data consistency for availability and scalability.

With this temporarily relaxed consistency, stale reads and conflicting writes are inherent risks. A classic approach to this problem is to ignore the stale reads and to detect and resolve conflicting writes. Unison[27](file synchronizer) describes 4 types of existing conflicts:

- change the name of the same file to different names on different replicas;
- delete a file on one replica and change its name on another;
- create a file with the same name and different contents on different replicas;
- make different modifications to the same content of a file on different replicas.

The option that Unison makes to resolving conflicts is to detect them and then request users to solve the conflicts. Most of file synchronizers, such as Dropbox² and SugarSync³ take this option to solve conflicts. Others, such as CVS[10] and SVN[15] try to merge different modifications to the content of a file, asking the help of the user only when this attempt fails.

Optimistic replication normally fits to systems that can tolerate some divergence between replicas and where conflicts are very rare. To these type of systems, optimistic replication can bring several advantages such as availability, scalability and the support of disconnected operations. Many, also indicated this model as the appropriate to some human activities, as it is better to allow collaborators (system users) to update data independently and repair occasional conflicts than to lock data while someone is editing it [10].

Distributed file systems are an example of systems that usually fit to an optimistic replication model, where usually conflicts are very unlikely to happen cite[43].

The goal of any optimistic replication system is to provide consistency while improving availability and scalability. However, there are some design choices according to the requirements of each system. On the following we describe some of those design choices, namely *update submission*, *update propagation*, *update transfer*, *Operations Scheduling* and *Conflict Resolution*.

Update Submission: Single-Master vs. Multi-Master [32]

Update submission regards where an update can be submitted to and how it is propagated. This can be divided in two main submission forms, Single-Master and Multi-Master. Single-Master consists on the submission of updates exclusively to one replica (master), and its propagation from that replica to the others (slaves). Since updates are all submitted to one replica, this type of systems can detect and solve conflicts in a centralized way. Besides its simplicity, they may have some limited availability caused by the bottleneck in the master replica when experiencing frequent updates.

Multi-Master consists on submitting updates to multiple replicas independently and its propagation in the background. Updates can be submitted to any replica, existing by this a decentralized way of detecting and solving conflicts. In comparison to single-master systems, these improve availability with the cost of a significantly more complex system. Nevertheless, and w.r.t. scalability it can be a problem due to the increased conflict rate.

Update Propagation: Push vs. Pull Model [32]

Update propagation regards the model that is used when there is an update to be propagated. There are two main models, namely push and pull model. On the push model used by systems such as Bayou[26] and Roam[30], a replica holding an update is responsible for pushing it to other replicas. On the pull model there is the concept of polling replicas in order to request the new updates. This polling process can be manually triggered or automatically using a periodical signal.

This design choice can have an important impact on systems regarding scalability, due to the overhead associated to periodic polling (pull model) or due to the high frequency of update propagation. Systems like Coda[34] use hybrid solutions to take both advantages of each model.

Update Transfer: State vs. Operation Transfer [32]

State and Operation Transfer are variants of update transfer and refer to what is transferred between replicas when there is an update to be propagated. On state-transfer systems, replicas are required to read or to overwrite a entire object. When reading/writing an update, a replica has to read/write the whole object

²Dropbox: Secure backup, sync and sharing made easy. <https://www.dropbox.com>.

³Sugarsync: Backup and file synchronizer. <https://www.sugarsync.com/>.

to which the update concern. It is a simple model of propagating updates that can easily and transparently be adapted to any solution, since maintaining consistency only involves sending the newest replica contents to other replicas.

On operation-transfer systems, replicas are required to propagate only the operations/modifications related to an update. In comparison to state-transfer systems, they can be more efficient since they do not need to send entire objects. Additionally, this model also improves concurrency and a lower conflict rate, since operations may be commutative. Yet, these systems are more complex as they need to reconstruct an history of operations.

Some systems like LBFS[23], Semantic-chunks[40] and Haddock-FS[6] make use of an efficient representation of updates to achieve a mixing of the two solutions (State and Operation Transfer). They use chunks as a representation for updates. As already mentioned on this document, chunks are portions of data within a file. With this notion, both advantages of the state transfer and operation transfer may be achieved. When a chunk is modified, its update involves the transfer of the whole chunk, making the transfer process easier as achieved by the state-transfer model. Yet, when a file is modified, the whole file does not have to be transferred, since only the affected chunks have to be transferred. Thus, it improves concurrency and decreases the conflict rate as achieved by the operation-transfer model.

Operations Scheduling: Syntactic vs. Semantic [32]

Operations Scheduling regards to the ordering of operations in a way that produces equivalent and expected states across users. There are two policies to produce the ordering of operations, namely Syntactic and Semantic scheduling. Syntactic scheduling is based on the time in which operations happened, preserving an operation ordering according to the relationship *happens-before* defined by Lamport[19]. This scheduling method is simpler than the semantic scheduling due to unnecessary knowledge about the semantic of operations. Nevertheless, as it does not examine the semantics of operations, it is not able to order operations in a way that can cause less conflicts. Semantic scheduling is based on the ordering of operations according to the operation's semantics. Thus, this method is able to reduce the conflict occurrence and increase the merging process of different operations. This policy is more complex than syntactic and is only applicable to operation-transfer systems, since state-transfer systems do not take into account the semantics of operations.

Conflict Resolution: Manual vs. Automatic [32]

A conflict occurs when a precondition of the system's scheduling is violated. The detection of conflicts may be based on a syntactic approach, when the *happens-before*[19] relationship is violated (e.g. when two or more operations are concurrently applied). On the other hand, conflict detection may be based on a semantic approach, which identifies conflicts according to the violation of precondition related with the semantics of the application.

W.r.t. conflict resolution/reconciliation[9], it may be performed manually or automatically. When performed manually, the conflict is detected and then delegated to the user to resolve. When performed automatically such as in Bayou[26], the conflict is resolved according to a set of rules defined by the application. These techniques try to reconcile and merge updates that respect to the same object. Systems like rcsmerge[37] try to merge updates through techniques based on plain text files. Systems like Semantic diff[17] try to merge updates based on the particular context of the application. In certain situations, this attempt of reconciliation may failure for example due to non-commutative actions and on this case the reconciliation has to be delegated to the user.

2.3 Adaptive Data Consistency

Many times, designers of replicated services are forced to choose either to use strong consistency guarantees or none at all, in order to cope with system's requirements. In this section we introduce the topic of adaptive consistency. In systems such as Haddock-FS[6] and IDEA[20], instead of having a static model of ensuring consistency guarantees, they use a model where the system adapts to the environment ensuring strong and weaken consistency guarantees as it is appropriate. With this model, systems may preserve strong consistency guarantees while improving scalability and efficiency.

On the following we present TACT, a middleware layer that enforces arbitrary consistency bounds amongst replicas. Further we introduce *locality-awareness*, a notion that has been widely researched and that can be used to achieve adaptive consistency systems. Finally, we present Vector-Field Consistency, an optimistic replication model that adapts consistency through locality-awareness techniques.

2.3.1 TACT - A Consistency Model for Replicated Services

TACT[45, 46] is an efficient and adaptable consistency model based on optimistic replication. Instead of having a model where either strong or weak consistency guarantees are enforced, TACT allows a consistency enforcement over multiple levels of consistency guarantees. With this concept of multiple consistency levels, it is possible to adapt consistency guarantees according to requirements (availability, performance, probability of inconsistent access). To achieve this consistency multiple level, TACT allows a bounded divergence between object replicas in accordance to a maximum level of inconsistency. This model proposes three metrics to bound consistency:

- **Numerical Error:** limits the total weight of writes that can be applied across all replicas before being propagated to a given replica.
- **Order Error:** limits the number of tentative writes that can be outstanding at any one replica.
- **Staleness:** limits the time delay of write propagation amongst replicas.

In order to specify consistency levels, applications specify their application-specific consistency semantics using *conits*. A *conit* is a physical or logical unit of consistency where applications quantify consistency continuously along a three-dimensional vector:

$$\text{Consistency} = (\text{Numerical Error}, \text{Order Error}, \text{Staleness})$$

According to consistency semantics, TACT is able to efficiently manage the propagation of updates, delaying updates that do not violate consistency bounds. Thus, TACT reduces the use of network resources and masquerades latency, while adjusting consistency guarantees in accordance to the application semantics.

2.3.2 Locality-awareness in Large-scale Systems

To achieve system's scalability over large networks such as the Internet, there is a need to reduce the amount of exchanged data between multiple entities. The notion of Locality-awareness is associated with Interest Management[16, 35, 36] that is a filtering mechanism that aims to solve the scalability problem through techniques that take into account the users' interest. Systems such as MANET(Mobile Ad-hoc Networks)[36] use techniques to detect the users shared interest in some topics in order to filter messages of low interest to them. Therefore, a higher scalability may be achieved by filtering massive volumes of data and thus reducing the volume of exchanged data that would be found as no interesting.

Locality-awareness is also a form of interest management, detecting users interest based on their locality. For instance, in massively multiplayer games, middlewares such as Matrix(Adaptive Middleware for Distributed Multiplayer Games)[2], VFC for Ad-hoc Gaming[33] and Unifying Divergence Bounding and

Locality Awareness in Replicated Systems with VFC[41] can track players position. According to it, they can strengthen consistency guarantees around the player position and weaken it as the distance to the player position increases. This model can achieve a higher scalability since it can adapt consistency levels according to the need and to the associated interest.

Current solutions such as Semantic-chunks[40] also use the notion of locality-awareness to efficiently share files between multiple users, taking into account information provided by the user, regarding their interests over each data set. This can improve these type of systems not only by reducing the overload of the network but also by reducing the latency, helping users to get important data in advance.

2.3.3 Vector-Field Consistency (VFC)

Vector-Field Consistency[41, 33] is an efficient and adaptable consistency model based on optimistic replication. It allows a bounded divergence between object replicas. For this, it takes into account several forms of consistency enforcement and a multi-dimensional criteria (time, sequence and value) to limit replica divergence. These forms of consistency are determined through techniques based on locality-awareness. Thereby, it uses locality-awareness techniques to identify different zones (consider a zone as a subset of a sharing set of data), in which the VFC dynamically strengthens/weakens replica consistency. Different multi-dimensional criterias are then applied to different zones, creating different divergence bounds to each zone.

To identify different zones, VFC uses the concept of *Pivots*. Pivots are based in locality-awareness techniques and they identify points in which consistency around is required to be strong, and weaker as the distance from the pivot increases. Figure 2.4 illustrates an example with 3 consistency zones, where a concentric circle was chosen as the space delimiter. The object O_3 was chosen as pivot. Therefore, VFC would enforce stronger consistency within Z_1 , following with Z_2 and by last Z_3 . Each object covered in each zone, would then have applied different divergence constraints, and thereby different consistency guarantees.

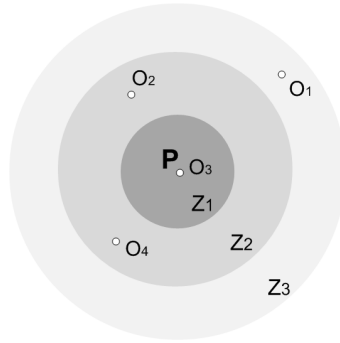


Figure 2.4: Consistency zones centered on a pivot.

To ensure different forms of consistency, VFC provides a 3-dimensional vector, $\kappa = [\theta, \sigma, \nu]$, to specify the consistency degrees. Each dimension of the vector bounds the maximum objects divergence in a particular view.

Each dimension is a numerical scalar defining the maximum divergence of the constraints time (θ), sequence (σ), and value (ν), respectively.

- **Time:** Specifies the maximum time a replica can be without being refreshed with its latest value. Consider that $\theta(o)$ provides the time passed from the last replica update, relatively to object o . The time constraint κ_θ enforces that, at any time, $\theta(o) < \kappa_\theta$. This scalar (not necessarily integer) quantity measures time in seconds.

- **Sequence:** Specifies the maximum number of lost replica updates. Similarly, consider that $\sigma(o)$ indicates the number of lost updates. The sequence constraint κ_σ enforces that, at any time, $\sigma(o) < \kappa_\sigma$. The unit is the number of lost updates.
- **Value:** Specifies the maximum relative difference between replica contents or against a constant. Consider that $\nu(o)$ provides this difference. The value constraint κ_ν enforces that, at any time, $\nu(o) < \kappa_\nu$. The unit of variation is a percentage.

For example, consider a consistency vector $\kappa = [0.1, 6, 20]$, that describes the divergence bounds of each constraint (time, sequence, value). It represents that replicas can only be outdated for 0.1 seconds or 6 lost updates or with a 20% variation in the replica content.

Through a selective form of increasing and decreasing consistency enforcement, VFC is able to ensure critical updates to be immediately sent and less critical to be postponed. Thereby, it makes an efficient resource usage, reducing the network bandwidth usage and masquerading latency.

2.4 Cloud Computing

Cloud computing[1] is a modern concept of using software and hardware infrastructures as a service over the Internet. It pretends to provide high performance computing as a cloud, where users may run their programs or store their data without having to concern about the management of the background system/infrastructure. As such, instead of having to manage large infrastructures and develop systems to provide high availability and scalability, cloud users can use the cloud as a service, paying only for what they use.

On contrary to client-server architectures, cloud computing provides an abstraction over the details of individual servers. Instead of performing requests to a server, cloud users may perform requests as a service, without the need of being concerned about the physical location of servers or cloud computing infrastructure. As such, users do not require any knowledge regarding the control and management of the remote services, as they are handled by cloud providers.

The most attractive features of cloud computing are: **i) Cost Reductions:** fewer IT skills, fewer implementation requirements and no waste of power and computing resources; **ii) Scalability:** mechanisms may auto-scale functionality according to users requirements. Developers do not need to concern about peak loads, as the cloud system scale resources; **iii) Availability:** improved if multiple redundant sites are used. Possibility of using multiple cloud providers; and **iv) Reliability:** improved if multiple redundant sites are used Possibility of using multiple cloud providers.

2.4.1 Cloud Storage

There are several types of services that may be provided by cloud computing. Nevertheless, for this particular document we focus on cloud computing as a system to provide large-scale storage (Cloud Storage). Many systems such as Dropbox⁴, SkyDrive⁵, SpiderOak⁶, Box.net⁷, SOS Online Backup⁸ and SugarSync⁹ use cloud systems in order to provide high available and reliable storage.

⁴Dropbox: Backup and file synchronizer. <https://www.dropbox.com>.

⁵Microsoft Windows Live Skydrive: Backup and file synchronizer. <http://skydrive.live.com/>

⁶SpiderOak: Backup and file synchronizer. <https://spideroak.com/>

⁷Box.net: Backup and file synchronizer. <http://box.net>

⁸SOS Online Backup: Backup solution. <http://www.sosonlinebackup.com>

⁹Sugarsync: Backup and file synchronizer. <https://www.sugarsync.com/>.

Such a service is offered by software running on a collection of servers, with data from client machines stored at the hard disks of multiple server nodes.

Typically, a cloud storage process on a client node transfers (part of) the data available in local storage back and forth to an entry point of the cloud storage service. This entry point makes sure that the data from the client is distributed over other server nodes. The cloud storage process keeps local data synchronized with data stored at the cloud storage service: new data generated locally by the user is uploaded to the cloud, data is retrieved from the cloud when local data was lost.

One advantage of cloud storage backup, contrary to a local backup to a medium like optical disk, is that the data is automatically geographically dispersed, which reduces the risk of losing data.

Another advantage is that cloud storage often offers additional features such as synchronization of the data between multiple computers or the sharing of data with others.

2.5 Relevant Systems

2.5.1 VFC for Cooperative Work

VFC for Cooperative Work[11] is a synchronization tool to efficiently synchronize Latex documents amongst collaborators. It uses the concept of locality-awareness to intelligently manage the transfer of updates. Through techniques that track the editing position of a user within a document, the system is able to reason about the importance of sections of a Latex document. With this information it performs a selective scheduling of update propagation, enforcing strong consistency guarantees to most important updates and weaker consistency guarantees to less important updates. This system divides Latex documents into chunks, which can be considered as sections or paragraphs of a document. Multiple consistency guarantees are then applied to multiple chunks according to clients locality.

To accomplish the dynamically adaptation of consistency guarantees, VFC for Cooperative Work uses the Vector-Field Consistency model. With this model, the system is able to assign multiple consistency guarantees (creating bounds of inconsistency) to multiple sections of a Latex document.

In comparison to systems that ensure total consistency, VFC for Cooperative Work is able to achieve higher performances w.r.t. bandwidth usage. Additionally, this system is able to improve concurrency and reduce the conflict rate, since modifications to different chunks of a document (different sections for example) are not considered as conflicts as they can be merged and constitute a single version. Nevertheless, this system does not perform a compression of transfered data, being unable to reduce even more the use of network resources. It also does not contemplate a compression of stored data.

2.5.2 Amazon S3 (Simple Storage Service)

Amazon S3¹⁰ (Simple Storage Service)[25] is an Amazon's system based on cloud computing to provide storage for the Internet. It provides a simple web services interface that can be used to store and retrieve any amount of data, at any time, from anywhere on the web. Through Amazon S3, developers may achieve highly scalable, reliable, secure, fast and inexpensive infrastructures to store data, without having to be concerned about any internal issues.

Amazon S3 is supported by a large number of data centers in the United States and Europe and is expected to offer low data access latency, infinite data durability and 99.99% of availability[25].

Data stored in Amazon S3 is organized in a two level namespace: *buckets* and *object names*. *Buckets* are similar to folders and allow users to organize their data. *Object names* correspond to objects that are stored

¹⁰Amazon s3. <http://aws.amazon.com/s3/>.

into buckets.

Regarding the data access protocols, Amazon S3 supports 3 main protocols: SOAP¹¹, REST¹² and BitTorrent¹³.

2.5.3 Dropbox

Dropbox¹⁴ is a commercial backup and file synchronizer[3] system that enables users to share data with others across the Internet. It is designed to achieve a high performance regarding the transfer of data between clients and servers. To provide and support storage to large-scale networks, Dropbox uses Amazon S3 to store files.

Each Dropbox client has in his computer a “Dropbox Folder” where he can modify and upload new files/folders. This folder is managed by a background process that is responsible for the correct synchronization of the folder between clients and servers. This application can also be used by several users as a collaboration tool as a user can allow others to access specific folders inside his “Dropbox folder”.

To accomplish the high performance w.r.t. data transfer, Dropbox uses delta-encoding techniques to produce a “binary diff” between new and previous versions of a file. As such, it enables an efficient syncing, only uploading changes made to a file. It also enables a file versioning allowing clients to fetch previous versions. To detect the existence of modified data, Dropbox also uses Compare-by-Hash techniques over folders to find out which folders have been modified. For this, the system exchanges folder hashes in order to find if the contents of a given folder have been modified.

With delta-encoding Dropbox is able to reduce the use of bandwidth, by only uploading the changes performed over a file. However, it does not perform deduplication to reduce the amount of stored data. Also, it does not use any kind of technology to specify multiple consistency levels, which could improve even more the efficiency in data transfer. Moreover, Dropbox does not provide any kind of tools to resolve updating conflicts, delegating to clients this task when any concurrent operations to the same files have been performed.

2.5.4 Microsoft Live SkyDrive

SkyDrive¹⁵ is a File hosting service that allows users to upload files to a cloud storage and then access them from a Web browser. It is based on cloud storage services that allow users to store and share data. It does not provide synchronization functionality between devices, and the primary interface is web browser based. A number of tools exist, however, that make uploading and syncing of local data more convenient. SkyDrive does not support versioning of objects. Individual items can be shared with others through the web.

2.5.5 Haddock-FS

Haddock-FS[6] is a peer-to-peer replicated file system. It is designed for mobile ad-hoc environments where constraints of reduced memory and low bandwidth are usual. Haddock-FS allows collaborative operations, detecting and solving conflicts by comparing multiple versions. To accomplish this, it uses a consistency protocol that relies on *dynamic version vectors*[30]. Haddock-FS is based on an update log system that organizes operations as tentative or as stable, according to the state of updates. Tentative updates are reversible on contrary to the stable updates. Stable updates are selected by a single replica called *primary*

¹¹<http://www.w3.org/TR/soap/>

¹²http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

¹³<http://www.bittorrent.com>

¹⁴Dropbox: Secure backup, sync and sharing made easy. <https://www.dropbox.com>.

¹⁵Microsoft Windows Live Skydrive. <http://skydrive.live.com/>

replica.

W.r.t. data redundancy, Haddock-FS makes use of compare-by-hash techniques in order to improve the bandwidth usage and to reduce the memory used by peers. To deal with the problem of shifting file offsets and overlapping chunks, this system uses Variable-size Block Hashing, basing chunk boundaries on file contents.

Haddock-FS is based on an adaptable optimistic consistency protocol, providing a highly available access to a weakly consistent view of files, while delivering a strongly consistent view to more demanding applications.

Briefly, Haddock-FS is able to reduce resources consumption regarding network bandwidth and memory. It makes use of deduplication techniques to either explore *cross-file* or *cross-version redundancy*. Additionally, it is also able to detect *locally untrackable redundancy*. Regarding the consistency protocol, it makes an efficient use of resources by enforcing either weak or strong consistency guarantees according to the required. Nevertheless, this system is not able to enforce multiple levels of consistency guarantees. By enforcing multiple levels of consistency guarantees, this system could balance resources and requirements, instead of forcing applications to choose between weak or strong consistency guarantees. Further, as Haddock-FS makes use of compare-by-hash techniques, it introduces an overhead regarding the meta-data exchange, which may not compensate the gains over low redundancy situations.

2.5.6 LBFS

LBFS[23] is a network file system designed to perform in low-bandwidth networks. The main goal of this system is to avoid the transmission of data that may already be found at the receiver's site. To accomplish this, this system makes use of compare-by-hash techniques in order to improve the bandwidth usage. To deal with the problem of shifting file offsets and overlapping chunks, this system uses Variable-size Block Hashing, basing chunk boundaries on file contents.

To make the chunk comparison possible both client and server store chunks in a database indexed by chunks hashes. When reading a file from the server, the client makes a request to the server in order to retrieve the hashes of the chunks to be read. Further, the client compares the received hashes with the already detained, labeling the chunks that were not found as missing. After this, the client requests the missing chunks to the server, receiving by this the missing data. As these operations are all pipelined, downloading a file only incurs in two network round-trips plus the cost of downloading the data.

When writing back a modified file, the opposite of the reading process is done. Firstly, the client sends the hashes and only after the missing data. To avoid dealing with the reordering of writes, LBFS implements atomic updates and a Close-To-Open consistency model. Thus, the commitment of updates is only applied when a file is closed. Additionally, when a file is closed by a client and another client reads it, it always receives its last content.

To achieve atomic updates, LBFS makes use of temporary files, in which updates are incrementally written. When the file is closed, the temporary file is then committed, overwriting the previous version of the file. First the client sends a "create temporary file" request to the server, to which the client will write the updates. Then, it sends the hashes of the chunks that compose the new version of the file. The server will return with a "missing chunk response" or with an "Ok response", which indicates that the server already detains that chunk. Missing data is pipelined from the client to the server, and at close time the client requests a commitment of the file.

Summing up, LBFS is a system that efficiently synchronizes data, saving resources regarding the use of network bandwidth. Although it is clear the improvement in the transfer protocol, this system has to

exchange sets of hash values between client and server, which in case of low redundancy may not compensate the gains and introduce a substantial overhead. Moreover, it does not explore data redundancy to efficiently store data. This system could use the same deduplication techniques to explore this last issue, making this a system that could efficiently transfer and store data. Additionally, LBFS also does not have into account multiple consistency guarantees, which could guarantee a multi-level consistency according to the bandwidth constraints and user needs. This could even improve more the efficiency w.r.t. data transmission.

2.5.7 redFS

redFS[7] is a distributed file system that performs *locally trackable deduplication* in order to achieve an efficient synchronization. The synchronization process of redFS is composed by two main steps: I) version tracking - responsible for detecting the set of versions that two synchronizing sites share in common; II) local redundancy detection - responsible for detecting local redundancy.

To perform the local redundancy detection, redFS makes use of Variable-size Hashing in addition with a byte-by-byte comparison technique. redFS uses simple hash functions over data chunks in order to detect similar chunks. After detecting the similarity it uses a byte-by-byte comparison technique to confirm the chunks similarity, preventing hash collisions. With this techniques, redFS is able to detect the common versions (common data chunks) between two sites and reduce redundancy with local deduplication techniques, avoiding the transfer of local redundant data and data that is already found at the receiver's site. Thus, when transferring data between two sites, redundant data is substituted by references to the actual chunks. Thus, only the non-redundant data (literal data) is sent over the network.

Summing up, redFS may be able to achieve better data transfer performance than systems based on Compare-By-Hash and Delta-Encoding, while maintaining the integrity of data during this operation.

2.5.8 Semantic-chunks

Semantic-chunks[40] is a middleware that aims to efficiently enforce data consistency for cooperative work systems. It uses Compare-by-Hash techniques with Variable-size Block Hashing to exploit data redundancy either locally or between multiple sites. Thus, it is able to detect either *locally trackable* or *locally untrackable redundancy*, being able to improve the efficiency of storing and syncing data.

Additionally, Semantic-chunks have presented a model in which documents are partitioned into chunks, and each chunk is annotated with semantic consistency information. These chunks are called semantic-chunks and consist in semantically-annotated document regions with relevance to applications and users, further annotated with consistency information and enforcement. The consistency information is in part provided by users. With this semantical structure a user is capable of know the structure of a given document, without the need of having to download the whole document content. As such, a user can work over parts of a document without the need of having the whole content of it. Thus, Semantic-chunks can ensure consistency over semantic-chunks of important relevance to a user, and relax consistency over less important chunks. With this concept, this system is able to improve concurrency and reduce update conflicts. Moreover, it reduces bandwidth usage, not only by exploiting data redundancy, but also by reducing and postponing the transmission of semantic-chunks with low relevance to a user. As Semantic-chunks uses Compare-by-Hash techniques, sometimes the introduced overhead for hash exchange between multiple sites, may not compensate the gains achieved by deduplication due to low redundancy. To overwhelm this situation, this system could use more efficient ways of representing the chunks knowledge.

2.5.9 ShiftBack

ShiftBack[39] is a backup system based in a client-server architecture, designed to support an efficient storage and network bandwidth use. Additionally, this system also supports a task-oriented backup with a

time-shifting interface which enables the browsing and retrieval of versions from any point in time. To cope with the requirements it uses Version-Based Deduplication techniques to explore *locally trackable redundancy*.

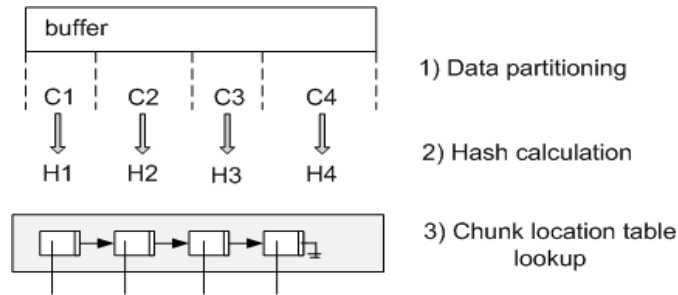


Figure 2.5: Example of deduplication process.

ShiftBack supports 3 main operations: Data Backup, Data Index and Data Recovery. Data Backup is the most frequent operation and consists in backing up data from the client to the server, storing and transferring only one occurrence of each chunk. This operation starts at the client side, that requests to the server its current state (knowledge vector). After this, it increments the knowledge vector and uses a Variable-size Block Hashing technique to find redundant chunks of data. Finally it returns to the server the literal data and its meta-data.

The local redundancy detection, illustrated in Figure 2.5, is accomplished within 3 main steps:

- Data partitioning using a rolling hash (variable-size blocks);
- Hash Calculation over the variable-sized blocks;
- Chunk repository lookup.

The chunk location table lookup is the process that involves finding a chunk with the same hash value (similar content). ShiftBack performs a search operation in which it tries to find a hash value. If the process succeeds, the chunk is substituted by a reference link to the single instance chunk. Otherwise, The new hash value is inserted in the chunk location table for further searches.

Data Index is a server-side only operation that consists in indexing the data stored at the server during backup, creating mappings between each backed up chunk's hash value and its location.

Data Recovery consists in the operation that retrieves the backed up data to the client. To do this, the server makes use of the mappings created by the data index operation, in order to find the chunks to be retrieved.

A special feature of ShiftBack is its high level of pipelining, being able to perform deduplication operations over sets of data while others are already being sent. Moreover, ShiftBack uses a pipe and filter architecture which allows each filter to run on a different thread enabling a better use of the CPU at the client.

ShiftBack provides a high efficient protocol to backup data through low-bandwidth networks, reducing data transfer and achieving better performances than other solutions based in Delta-Encoding and Compare-by-Hash techniques. This is due to the use of lightweight structures (knowledge vectors) to represent the whole state of a site, reducing the amount of exchanged meta-data. However, this system does not provide any form to reduce/postpone the exchange of data having into account the importance of the backed up data. This system could use a model to somehow ensure that important data is immediately backed up, and less important data is postponed to moments of high-bandwidth connection.

2.5.10 Subversion (SVN)

Subversion (SVN)[15] is a revision control system typically used to synchronize and store multiple versions of source code files. SVN is based on a client-server architecture. It supports disconnected operations and provides to clients tools for handling conflicting updates, since normally there are multiple clients making concurrent changes to the same files.

In order to achieve higher performances w.r.t. data transfer protocol, SVN tries to reduce the use of network resources through the use of delta-encoding techniques. Through this technique it compares file versions with their previous versions, detecting *cross-version redundancy*. This redundancy exploitation is not only used to reduce the use of network bandwidth, but also to achieve better performance w.r.t. data storage. As specified, the delta-encoding technique needs one new version and one old version to encode data, which forces the client to use extra space to store old versions. Furthermore, this method also imposes the limitation that each file is encoded only against one other file, which makes SVN unable to exploit *cross-file redundancy*.

In short, SVN is able to perform efficient data transfer, improving concurrency and providing tools to reconcile conflicting updates.

2.6 Summary

The following table 5.2.4 presents a summary of the above mentioned systems w.r.t. deduplication techniques and data synchronization models.

System	Description	Deduplication Algorithms	Adaptive Data Consistency Mechanisms
Dropbox	File sharing system	Delta-Encoding	None
Haddock-FS[6]	Distributed file system	Compare-by-hash (Variable-Size Hashing)	Hybrid consistency: weak or strong consistency guarantees according to the resource constraints
LBFS[23]	Distributed file system	Compare-by-hash (Variable-Size Hashing)	None
redFS[7]	Distributed file system	Version-Based deduplication + Variable-Size Hashing + byte-by-byte comparison	None
Semantic-chunks[40]	Middleware for ubiquitous cooperative work	Compare-by-hash (Variable-Size Hashing)	Several consistency levels according to user provided information
ShiftBack[39]	Backup system	Version-Based deduplication + Variable-Size Hashing	None
SVN[15]	Revision control system	Delta-Encoding	None
VFC for C. W.[11]	Synchronization tool	None	Several consistency levels according to user locality

Table 2.2: Comparison between the studied systems.

Many systems do not have any mechanisms to adapt consistency guarantees according to the needs/resources. Although some systems provide mechanisms to adapt data consistency, they do not provide multiple levels

of consistency. These, either enforce weak or strong consistency guarantees, becoming unable to provide intermediate consistency guarantees.

Further, most systems are not capable of taking into account the interests of users in order to propagate updates of the shared data. Nevertheless, there are some few systems that provide these features. Yet, they lack of efficient mechanisms to explore redundant data in order to either reduce storage space or network bandwidth.

Chapter 3

Architecture

This chapter presents the architecture of the proposed file sharing system, named VFCbox. This system is capable of synchronizing files between multiple users. It makes use of deduplication techniques to reduce data redundancy between synchronizing sites. Additionally, VFCbox uses a relaxed consistency model that takes into account the interests of users over certain files or certain parts of the sharing files, in order to create multiple consistency levels.

3.1 Baseline Architecture

VFCbox is based on a client-server architecture. Figure 3.1 illustrates the process where clients submit their updates and data's interests to the server (for simplicity of the presentation, with no lack of generality, we consider just one server, that could reside inside a data center, or cloud infrastructure). These interests represent files or parts of files (e.g. chapters/sections of a document) in which clients have special interests. Taking these interests into account, the server is then able to enforce multiple consistency guarantees over multiple data subsets.

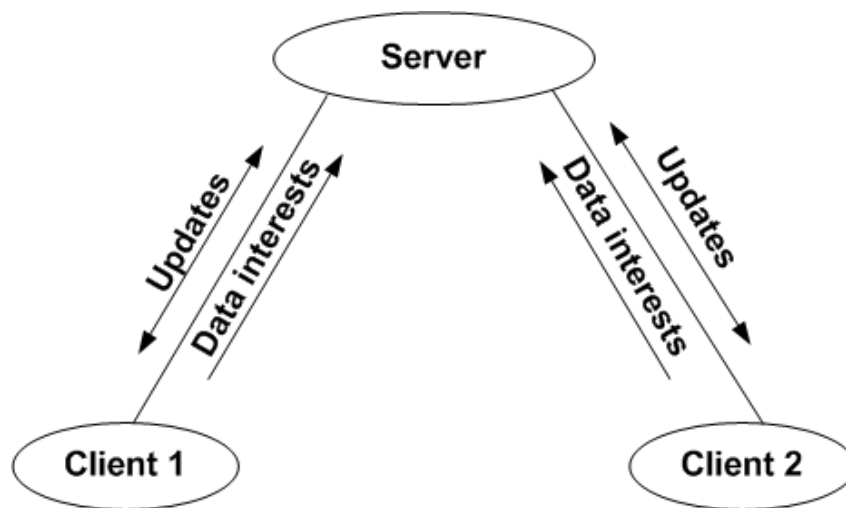


Figure 3.1: VFCbox's main overview.

Thus, the process is based on the exchange of updates between clients and server, and on the submission of client's interests to the server. These interests are then used to make decisions of which updates have to

be immediately propagated and which can be stored for a while and only later be sent.

To reduce the use of network resources, VFCbox employs deduplication techniques to the uploading (from client to server) and downloading (from server to client) process of files and file updates.

Additionally, users may upload interests to certain files or file parts, in order to guarantee a higher consistency level to high relevance data. In particular, it does not propagate useless or unneeded updates (i.e. those regarding data which is not relevant for a user, or that can be subsumed by a later update sent). Thus, files or files parts with low relevance to users have applied lower consistency guarantees, which also reduces the use of network resources. Although we are reducing the consistency between users, this may not really represent an inferior usability feature. Since users have the control of consistency guarantees, the lower consistency may only represent the discard of updates with no relevance to the user.

Users may manifest their interests through an interface. In this interface users may mark which zones of a file, or which files of the total set of files, are of special relevance to them.

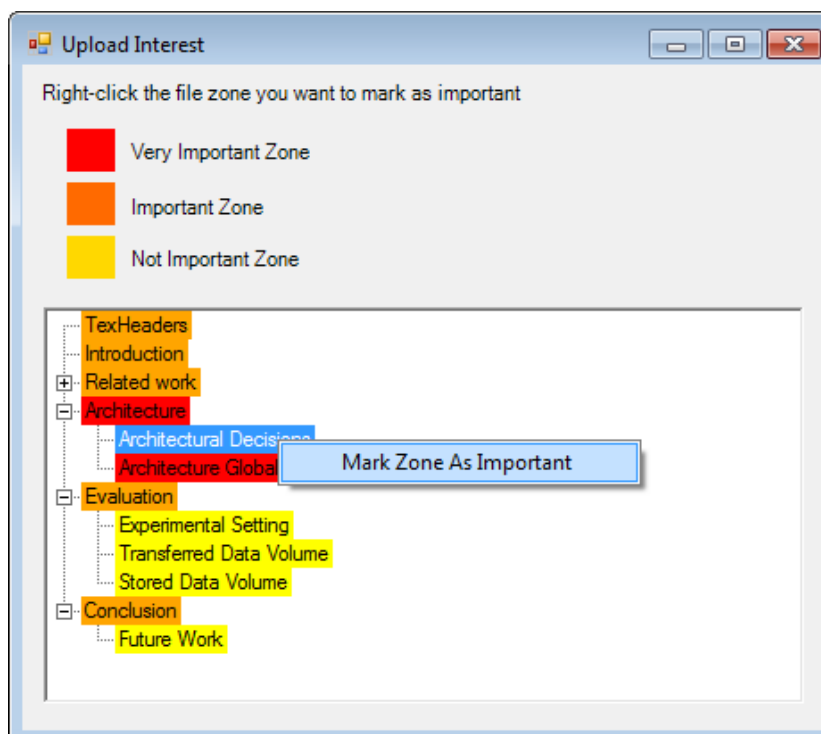


Figure 3.2: A file divided in zones defining multiple consistency zones. This example illustrates an interface in which users may specify their interests over parts of a file.

Figure 3.2 illustrates an example of how users can mark a special interest over a certain zone (which we call **semantic zone**) of a file. For exemplification proposes, we present a Latex document composed by several chapters and sections. We may observe that the user marked a section (Architectural Decisions) as the pivot zone. This means that this client is particularly interested in this section of the document. The colors of the background of each chapter/section show the importance levels that have been assigned to them. Thus, each semantic zone of the current file will have applied a set of consistency guarantees according to the assigned importance level. Additionally, users can mark several pivots over several zones of a file in order to obtain manifest multiple interests.

Additionally, we also provide to users a way of specifying interests over files. In this case, instead of specifying interests over zones of a document, users may specify different interests over specific files.

Figure 3.3 illustrates an example of the above mentioned. In this example the user marked 2 files as very important files, 3 files as important files and 5 files as not important files.

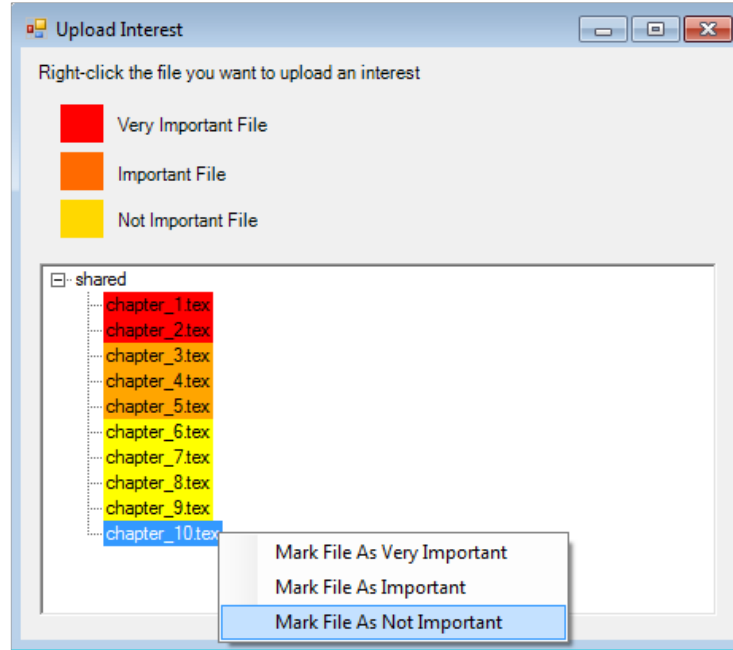


Figure 3.3: VFCbox interface in which users may specify special interests over files of a shared folder.

3.2 Architectural Decisions

In this section we describe the main architectural decisions regarding data deduplication and replication model. These decisions were the drivers to design the VFCbox architecture.

On the following we describe the data deduplication decisions w.r.t. deduplication algorithm, deduplication timing and deduplication placement. Further we describe the replication model decisions w.r.t. the consistency model, update submission, update propagation and conflict resolution.

3.2.1 Data Deduplication

To achieve a high performance in terms of bandwidth reduction, we selected a Variable-size Block Hashing (VBH) technique to perform compare-by-hash techniques locally.

The advantages of VBH are related to the possibility of exploiting **cross-version** and **cross-file redundancy**, and the use of **content-based chunking** which avoids the problem of overlapping chunks. Contrary to the compare-by-hash protocol used by systems like LBFS[23], we opted to make the comparison of hash values only locally.

Thus, we avoid to add an extra round-trip to the transfer protocol to send the hash values and compare them between the synchronizing sites. Instead, we mark each hash value that have already been sent to the synchronizing site. As such, when re-sending an object and as the hash value is marked as synchronized,

only the hash value is transferred.

For example, when a site is synchronizing a file with another site, it partitions the file in several chunks (of variable size), computes their hash values and marks the chunks as synchronized with the concerning site. In further synchronizations, the sender site already knows that the other site contains a copy of those chunks, and thus do not have to re-send them or search for its existence.

With these techniques, bandwidth usage may have several gains. Variable-size Block Hashing allows a fine-grained redundancy exploitation, which can reduce space used to store data and bandwidth usage since redundant data are no more sent over the network.

Regarding deduplication timing, the system is designed to perform deduplication operations on ingestion time (synchronously) at the client side. This avoids the storage of redundant data and avoids the transfer of redundant data to the server.

3.2.2 Replication Model

In order to prevail the features of **availability**, **scalability**, **improved concurrency** and to **support disconnected operations**, we decided to choose an **optimistic replication model**. The occurrence of writing conflicts is then an inherent property.

To deal with the problem of detecting and resolving conflicts, we decided to select a **Single-Master model**, where all updates are submitted, in order to have a centralized way of detecting conflicts. This decision was based on the additional complexity and scalability problems associated to the Multi-Master model, which can increase the conflicting rate.

W.r.t. conflict resolution we decided to delegate the reconciliation to the user. Nevertheless, we decided to contemplate auxiliary tools in order to facilitate the conflict's resolution.

To propagate replica updates we aimed to a **push propagation model** on the client side. This delegates to clients the duty of propagating to the server their existing updates. The opposite way was to have a pulling operation on the server side that periodically could request new updates. Yet, this would introduce an overhead to the server and to the network protocol, which makes the push propagation model a better option.

On the server side we also decided to have a **push propagation model**, which gives the responsibility to the server of propagating new updates. In comparison to the push propagation model, this model avoids the overhead of having peaks of client requests and frequent requests.

W.r.t. file sharing systems, patterns have identified that many times most of the elements of a collaborative team are only interested in parts of the sharing data.

For instance, a team of co-workers writing a document may have elements that are only working in a specific part of the document, not being really interested on the rest of the document. Nevertheless, if two co-workers are working in the same part, they may want to ensure a strong consistency of that data through time.

This led us to the topic of **locality-awareness (or interest-awareness)**, where the knowledge of **user's interests** can be taken into account to specify **multiple consistency guarantees**. As such, we decided to use an optimistic replication model that uses the interest of users over the shared data to enforce different consistency guarantees. The advantages are several:

- **reduced latency** on the transmission of data considered as more relevant to the user;
- **bandwidth usage reduction**, since less updates messages are sent due to the lower consistency guarantees enforcement;

- improved scalability.

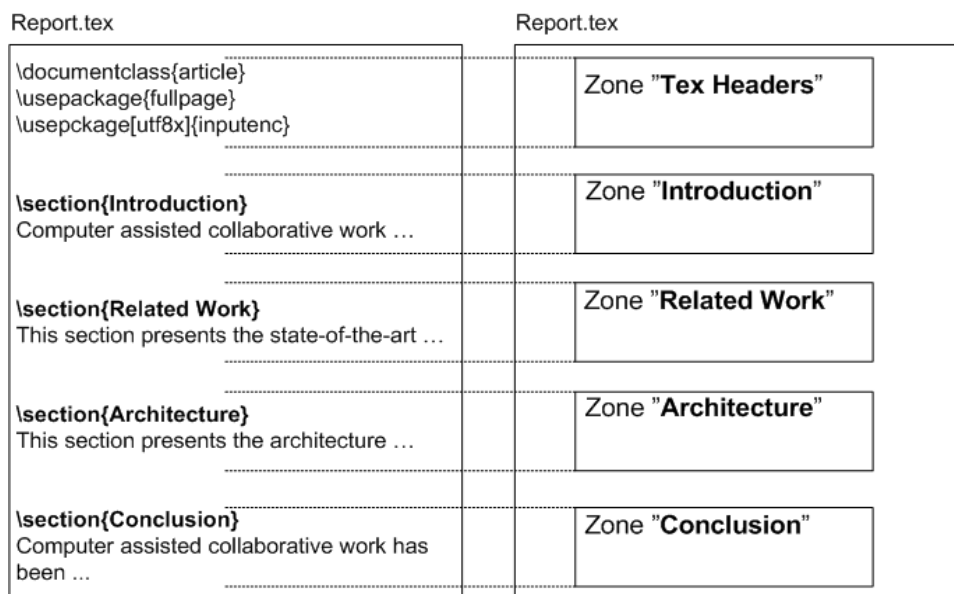


Figure 3.4: A file divided in zones defining multiple consistency zones (semantic zones). This example illustrates how VFC was adapted to documents.

We identified the **VFC model** [41, 33] as a natural fitting model to this environment, where its locality-awareness techniques can be applied from the most to the less important user's data. By this, VFC can impose strong consistency guarantees to parts of data that are of extreme interest to users. Or relax some consistency guarantees to parts of data of less interest to users.

Figure 3.4 illustrates a document file partitioned into several zones, defining multiple consistency zones, which we call **semantic zones**. In this figure and for exemplification proposes we present a Latex file composed by several sections, which naturally defines multiple zones. Each semantic zone may then have applied different consistency guarantees according to user's interests. Users may assign one semantic zone as the more interesting zone (pivot zone). According to this interest, each semantic zone will have assigned a certain consistency level in accordance with the distance to the pivot zone.

3.2.3 VFCbox Main Components

In this section we describe an overview of the VFCbox main architecture, namely the client and server nodes.

Client nodes correspond to a user-level application and environment that enables clients to asynchronously edit files while guaranteeing their efficient synchronization. To accomplish this, the VFCbox client node is continuously monitoring the file system in order to detect file modifications that were performed by other applications. Once detected, the modification is considered as a new update and has to be propagated to the server node. Before transferring the update to the server, the client node performs deduplication and data compression operations in order to reduce the transferred data.

Additionally, the client node has also the duty of informing the server about the current interests that users have over parts of shared data.

The Server node corresponds to the central node of the system. All updates are either received or sent by the server, being this node responsible to ensure data consistency through client nodes. This component is thus responsible for receiving updates and propagate them according to users interests. Additionally, this

component also performs deduplication operations in order to remove redundancy from data to be transferred from the server to clients.

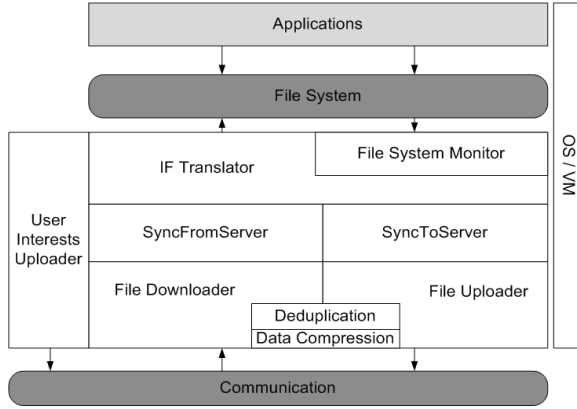


Figure 3.5: Client Architecture.

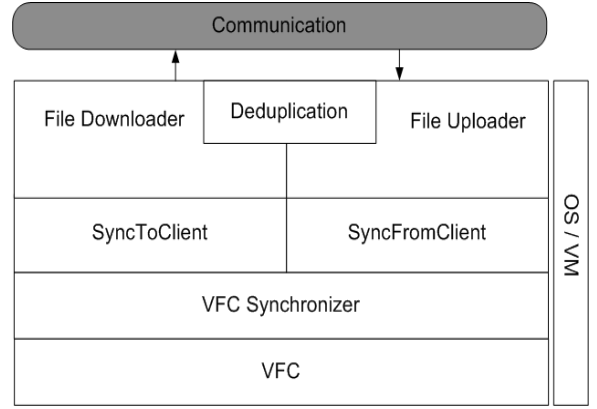


Figure 3.6: Server Architecture.

The client node architecture (Figure 3.5) is composed by 9 main modules: *User Interests Uploader*, *File System Monitor*, *Intermediate Format (IF) Translator*, *SyncFromServer*, *SyncToServer*, *File Downloader*, *File Uploader*, *Deduplication* and *Data Compression*. On the following we describe each module in more detail:

- **User Interests Uploader:** This module is responsible for uploading the user's interests to the server.
- **File System Monitor:** This module is responsible for keeping track of any document update performed by the application.
- **Intermediate Format (IF) Translator:** This module is responsible for translating files to an intermediate format in XML that is used to define the multiple semantic zones.
- **SyncFromServer:** This module is in charge of the synchronization process from the client side to the server side. It is responsible for updating each object that is being updated.
- **SyncToServer:** This module is in charge of the synchronization process from the client side to the server side. It manages the versioning information about each update.
- **File Downloader:** This module is responsible for receiving the data that is being sent from the server to the client side. It is responsible for controlling the data flow through the deduplication and compression process.
- **File Uploader:** This module performs the data transfer from the client to the server side. It is responsible for controlling the data flow through the deduplication and compression process.
- **Deduplication:** This module performs deduplication operations for transfer proposes. It is responsible for detecting redundant data between the client and the server, and for substituting it for references.
- **Data Compression:** This module performs compression operations, either for compressing outgoing data or decompressing incoming data.

The Server node (Figure 3.6) corresponds to the central node of the system. Every updates are either received or sent by the server, being this node responsible to ensure data consistency through client nodes. This component is thus responsible for receiving updates and propagate them according to users interests.

Additionally, this component also performs deduplication operations in order to remove redundancy from data to be transferred.

We propose an architecture where server nodes are composed by 7 main modules: *File Downloader*, *File Uploader*, *Deduplication*, *SyncToClient*, *SyncFromClient*, *VFC Synchronizer*, and *VFC (Consistency Manager)*. On the following we describe each module in more detail:

- **Deduplication:** This module performs deduplication operations for transfer proposes. It is responsible for detecting redundant data between the server and the clients, and for substituting it for references.
- **File Downloader:** This module is responsible for transferring data to clients. It is responsible for controlling the data flow through the deduplication process.
- **File Uploader:** This module is responsible for receiving data from clients. It is responsible for controlling the data flow through the deduplication process.
- **SyncToClient:** This module is in charge of the synchronization process from the server side to the client side. It is responsible for updating clients with new updates and detecting the occurrence of conflicts.
- **SyncFromClient:** This module is in charge of the synchronization process from the client side to the server side. It is responsible for updating the server with the incoming updates and detecting the occurrence of conflicts.
- **VFC Synchronizer:** This module is responsible to synchronize incoming updates with all the clients. It also has the responsibility of forwarding new updates to the VFC (Consistency Manager).
- **VFC (Consistency Manager):** This module is responsible to enforce the VFC model over the synchronization process. Thus, it stores and manages all user's interests in order to perform the propagation of updates in a selectively way.

3.2.4 Upload Pipeline

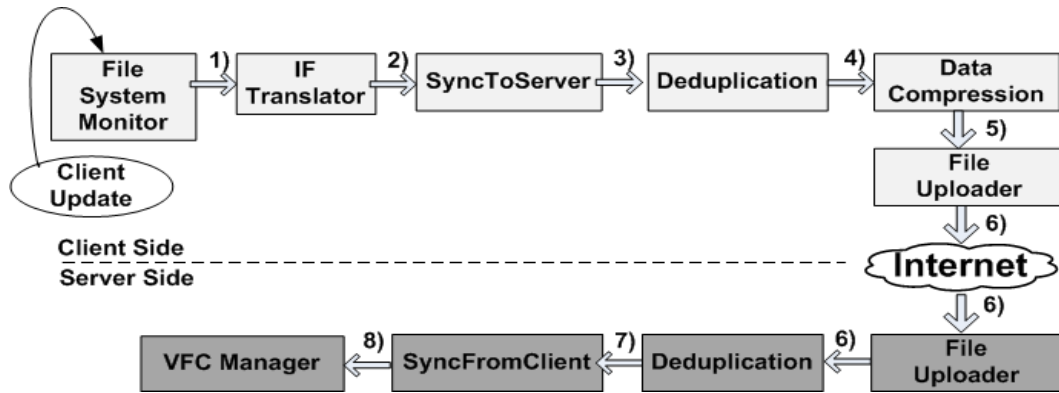


Figure 3.7: VFCbox's data upload pipeline: modules of the baseline architecture that are involved in the data upload.

This section describes how the **upload pipeline** supports the upload of new file updates. Its duty is to monitor and detect new updates (file creation, file's contents update, etc.) and to upload them to the server. Additionally, this pipeline is responsible of performing **deduplication** and **compression operations** in order to reduce the use of bandwidth resources in the synchronization process (from client to server).

Figure 3.7 illustrates the pipeline used to perform the whole process of synchronizing a new update to the

server. The process starts on the client side when a user performs an update to an existing file (or creates a new file). The process then ends when all new updates are synchronized with the server.

The upload process is composed by 7 main steps, namely 1) *File System Monitoring*, 2) *File Translation to the Intermediate Format*, 3) *Client Synchronization*, 4) *Client Deduplication*, 5) *Data Compression*, 6) *Data Transfer*, 7) *Server Deduplication* and 8) *Server Synchronization*. On the following we describe each step and each involved module of the process:

- **1) *File System Monitoring***

The process of uploading a file update starts with an update event, triggered by the file system monitor. It happens when a user performs a file modification, triggering an event on the client node to which information about the update is passed.

Further details about the File System Monitor are explained in Section 4.1.2.

- **2) *File Translation to the Intermediate Format***

After detecting a new update, the update is translated to an Intermediate Format (IF). This allows the usage of the same format and same semantic zone definition for any type of files, since all files are translated to the IF.

Further details about the used Intermediate Format are explained in Section 3.3.

- **3) *Client Synchronization***

In this phase the new update is synchronized with the server. This synchronization step is responsible for managing all the versioning information of updates and for keeping the knowledge of the version that the server is aware of. Multiple semantic zones of a certain file are considered as different and independent objects and therefore are independently synchronized. Thus, when a certain semantic zone of a file is updated, only the concerning semantic zone is updated.

Section 3.6 describes more details about the synchronization process.

- **4) *Client Deduplication***

Before transferring the new update's data to the server, the process of deduplicating redundant data is triggered. Briefly, the deduplication process is composed by 3 main steps, namely data partitioning (chunking), chunk hashing and chunk lookup (actual deduplication). The first step consists in the partitioning of data in chunks, using a contents based method. Then, an hash value is calculated for each created chunk. By last, the hash value is searched in the client's chunk repository and in the client's references table in order to detect if the concerning chunk as already been sent to the server in previous uploads. The chunks repository is responsible for storing each data chunk. The references table is responsible for maintaining information about the data chunks that have already been sent to the server. If the concerning chunk has already been sent to the server, the chunk is replaced by a single reference containing the chunk's hash value. Additionally, the client adds to the references table a reference to each chunk that is sent to the server, in order to find in the future if that chunk has already been synchronized.

Further details about the deduplication process are explained in Section 3.4.1.

- **5) *Data Compression***

After the deduplication, the remaining literal chunks are compressed using the BZip2¹ compression algorithm.

Further details about the compression process are described in Section 3.5.

- **6) *Data Transfer***

The upload process then continues with the data transfer of the new update. As already mentioned, VFCbox uses an Intermediate Format to represent the multiple semantic zones of a document file. Nevertheless, when uploading to the server a file update, only the affected semantic zones are transferred

¹<http://www.bzip.org>

avoiding to send large sets of chunks references. Thus, when uploading a file update, only the updated semantic zones are transferred. Additionally, and for each semantic zone, only the content of the literal chunks is sent plus the references to the non literal chunks.

- **7) Server Deduplication**

On the server side and for each chunk that the server receives, the server adds to the references table a reference to the received chunk, indicating that the uploading client contains a copy of it. This information may be used later, in order to detect redundant data between the server and the concerning client.

Section 3.4.1 describes more details about the deduplication process.

- **8) Server Synchronization**

By last, the server updates the versioning information of the incoming update, determining if there were any conflict occurrence. For each received semantic zone and for each client, a consistency level is assigned in accordance to client's interests. This consistency level is calculated taking into account the distance to the selected pivot zone of the concerning file and user. This phase performed by the VFC model, allows the further synchronization of incoming updates with the rest of the clients. The following section (Section 3.2.5) describes this synchronization: File Download.

Section 3.6 describes the consistency model and the VFC enforcement in more detail.

3.2.5 Download Pipeline

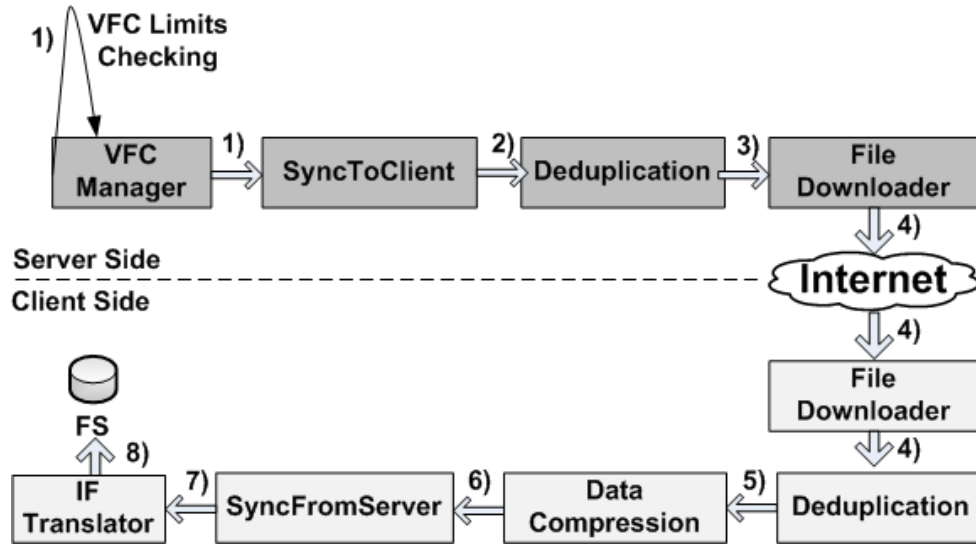


Figure 3.8: VFCbox's data download pipeline. The figure illustrates the modules of the baseline architecture that are involved in the data download.

This section briefly describes how the **download pipeline** supports the download (i.e. from server to client) of new file updates. The download of a file update is initiated on the server side (push propagation model). The duty of this pipeline is to periodically check if there are updates to be sent from the server to clients. This synchronization process is controlled by the VFC model, which determines if an update will be delayed or immediately synchronized with a certain client. The VFC model is able to perform this selective selection and propagation of updates according to each user's interests. Thus, the process of downloading a file update starts on the server side once the divergence bounds of a shared object (file or file semantic zone) are exceeded. Additionally, the server performs deduplication in order to avoid to send data chunks that are already contained by clients.

Figure 3.8 illustrates the pipeline used to perform the whole process of synchronizing a new update with

a client. The process starts on the server side when the divergence limits of a certain object are exceeded. The process ends when the concerning object is synchronized with the client.

The download process is composed by 8 main steps, namely 1) *VFC Limits Checking*, 2) *Server Synchronization*, 3) *Server Deduplication*, 4) *Data Transfer*, 5) *Client Deduplication*, 6) *Data Decompression*, 7) *Client Synchronization* and 8) *File Translation to the File Format*.

We now describe each step and each involved module of the process:

- **1) *VFC Limits Checking***

The download of a file update starts when the VFC Manager of the system triggers a synchronization event to a certain client. As already mentioned, a file is composed by a list of semantic zones, to which different consistency levels are assigned in accordance to each user's interest. Each consistency level has a set of divergence bounds associated, namely sequence and time bounds. The sequence bound limits the number of updates that the concerning client may miss. The time bound limits the time that the concerning client may be without being refreshed with the latest update. Therefore, when a semantic zone exceeds its divergence bounds with a certain client, the semantic zone is elected to be synchronized, and the process of downloading a file update is initiated.

Section 3.7 describes how the VFC model is enforced in a more detailed way.

- **2) *Server Synchronization***

The set of updates that are found to be synchronized with a client are passed to the synchronization module. This module is responsible of updating the versioning information of each synchronizing object.

Section 3.6 describes more details about the synchronization process.

- **3) *Server Deduplication***

Before transferring the contents of each update to the concerning client, the server performs the deduplication process. To accomplish this, the server searches its references table in order to find if the current client already contains a chunk of data. For each chunk to be transmitted, the server performs this action. If a reference of a chunk is found in the references table, the chunk is replaced by a reference containing the chunk's hash value. If not, the actual data chunk is fetch from the chunks repository and the process proceeds to the data transfer step.

- **4) *Data Transfer***

The download process then continues with the data transfer of the current update.

- **5) *Client Deduplication***

When the client receives a literal chunk, it stores the chunk in the chunk repository and adds a reference to it in the references table. When the client receives a non literal chunk, it adds a reference to it in its references table and searches for the chunk's content in the chunk repository.

Section 3.4.1 describes the deduplication process in more detail.

- **6) *Data Decompression***

After the deduplication process, the compressed data chunks are decompressed in order to get the actual content of the update.

Further details about the decompression process are described in Section 3.5.

- **7) *Client Synchronization***

The new update is synchronized with the client. This synchronization step is responsible for updating the versioning information.

Section 3.6 describes more details about the synchronization process.

- **8) *File Translation to the File Format***

The client translates the synchronized update to the actual file format which enables the actual write of the update to the client file. A notification to the client is also shown presenting the affected file and semantic zones.

3.3 Data File Representation (Intermediate Format)

To represent the data files and describe the multiple semantic zones of a document file, VFCbox makes use of an intermediate format (IF) to represent the data and meta-data associated with each file. As such, every file that is consumed by the system is translated to the IF. This allows the addition of any type of files to the system only by adding a plugin to translate the concerning type to the IF. The duty of each plugin is then to determine the multiple semantic zones of a file and represent these zones in the IF. For instance, in the case of Latex documents, we could split documents in chapters and sections, considering each of them as a semantic zone. Another example could be the partitioning of a spreadsheet in multiple sheets or the partitioning of a sliding show in multiple slides.

Latex document	Intermediate format
<pre> \documentclass{article} \usepackage{fullpage} \usepackage[utf8x]{inputenc} \section{Introduction} Computer assisted collaborative work ... \section{Related Work} This section presents the state-of-the-art ... \section{Architecture} This section presents the architecture ... \section{Conclusion} Computer assisted collaborative work has been ... </pre>	<pre> <doc> <docStructure> <zoneId depthLvl=1>Tex Headers</zoneId> <zoneId depthLvl=1>Introduction</zoneId> <zoneId depthLvl=1>Related Work</zoneId> <zoneId depthLvl=1>Architecture</zoneId> <zoneId depthLvl=1>Conclusion</zoneId> </docStructure> <zone id="Tex Headers"> \documentclass{article} \usepackage{fullpage} \usepackage[utf8x]{inputenc} </zone> <zone id="Introduction"> \section{Introduction} Computer assisted collaborative work ... </zone> <zone id="Related Work"> \section{Related Work} This section presents the state-of-the-art ... </zone> <zone id="Architecture"> \section{Architecture} This section presents the architecture ... </zone> <zone id="Conclusion"> \section{Conclusion} Computer assisted collaborative work has been ... </zone> </doc> </pre>

Figure 3.9: Example of a Latex document translated to the intermediate format.

Already existing formats could have been used to express the multiple consistency zones. Nevertheless, we opted to create a simpler format, avoiding to incur in unnecessary overloads.

The VFCbox IF makes use of XML to define the file format. On the following we describe in more detail the structure nodes (composed by lists) of the IF:

- **doc**: parent node of any file. It is constituted by a *docStructure* node and multiple zone nodes.
- **docStructure**: node that describes the list of semantic zones that constitute the file. It is composed by a list of *zoneId* nodes.
- **zoneId**: node that describes the identification and depth level of a semantic zone.
- **zone**: node that contains the content data of a semantic zone.

Figure 3.9 represents an example of a Latex document translated to the IF. In this example, we may see that the Latex document was composed by 5 chapters (Tex Headers, Introduction, Related Work, Architecture and Conclusion), now transformed in 5 semantic zones.

3.4 Deduplication Architecture

In this section we describe the architecture of the deduplication used to reduce the redundant data transferred over the network. The deduplication process is capable to explore the redundant data either from the client to the server side or from the server to the client side. Through comparison of variable-size hashes we are capable to explore both cross-file and cross-version redundancy.

This section describes all the steps of the deduplication process, presents the main modules of the deduplication architecture and describes the data transfer protocol.

3.4.1 Deduplication Process

In short, the deduplication process is composed by three main steps: I) **Data partitioning**; II) **Hash Calculation**; III) **Chunk Lookup in chunk repository and references table**.

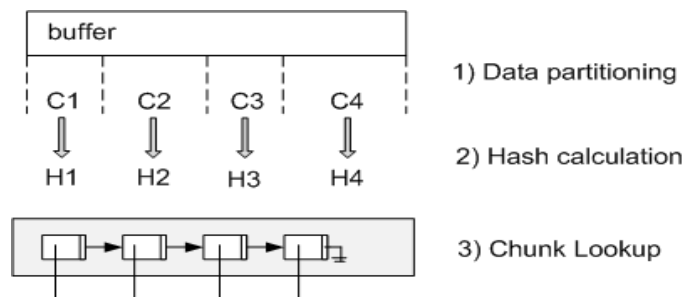


Figure 3.10: Example of deduplication process.

The process of deduplication, illustrated in Figure 3.10, is composed by three main steps. The first step regards the process of partitioning data that is found to be written. This partitioning is accomplished by using Rabin Fingerprints to calculate the chunk boundaries according to data's contents. It is based on the examination of every 48-byte regions of the file and respective calculation of a rolling hash. The rolling hash is then compared with a pre-defined value. The probability of having a match between the rolling hash and the pre-defined value is of 2^{-13} , and when it occurs, the current region is marked as a chunk boundary.

The second step of the process constitutes the hash calculation (using SHA-1) of each chunk provided by step 1.

The third and last step regards the process of writing data in a compact form. The compact form is constituted by a list of references to chunks. For this, it is required a lookup over the chunks repository and over the references table in order to detect redundant chunks. If the chunk already exists, only a reference is added to the chunk contained by the repository. Otherwise, the new chunk has to be added to the repository, creating for it a new entry that is identified by its hash-value.

The duty of the chunk repository is thus to store chunks and provide the possibility of performing lookups, in order to detect already existing chunks. Each entry of the repository contains a unique hash-value of a chunk and a reference to the actual chunk.

The references tables consists on the set of hash values that each site knows to be found at a given site. The duty of the references table is to store references to chunks that have been sent to a certain site. These references are associated with the synchronizing site identification. This provides the possibility of performing lookups, in order to detect if a certain chunk has already been sent to a certain site.

3.4.2 Deduplication Modules

This section describes and illustrates the main modules of the deduplication architecture. The client node of the system has always the duty of performing the whole deduplication process, namely data partitioning, hash calculation and chunk lookup. The server node only performs the last step of the deduplication process, namely chunk lookup.

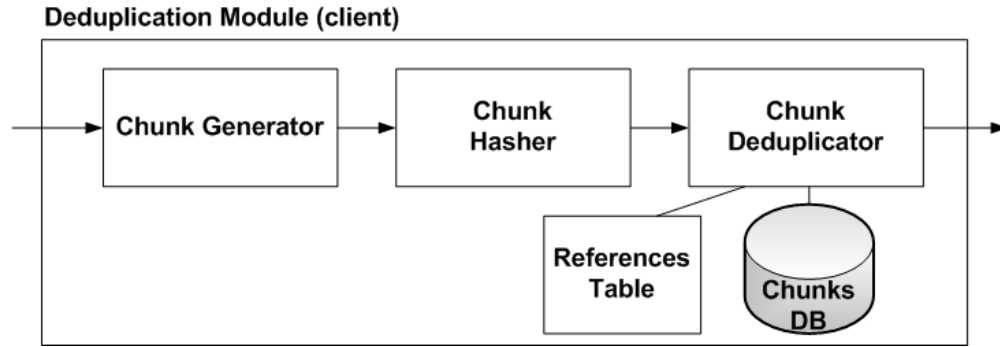


Figure 3.11: Deduplication Client Architecture.

Figure 3.11 represents the deduplication architecture on the client side. It is composed by 3 main modules:

- **Chunk Generator:** This module performs the chunking operations. It is responsible for calculating a rolling hash using Rabin Fingerprints and to identify the bounds of each data chunk.
- **Chunk Hasher:** This module is responsible for calculating the hash value (with SHA-1) of each data chunk.
- **Chunk Deduplicator:** This module performs the actual deduplication. It makes use of a chunk repository to store the literal data that is indexed by their hash values. It also makes use of a references table in order to find if a certain chunk has or has not been already sent to the server.

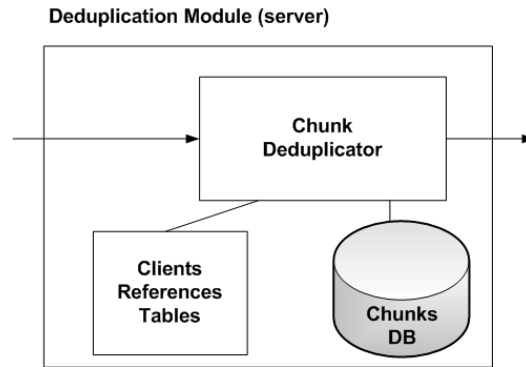


Figure 3.12: Deduplication Server Architecture.

Figure 3.12 represents the deduplication architecture on the server side. It is composed by 1 main module:

- **Chunk Deduplicator:** This module performs the actual deduplication. It makes use of a chunk repository to store the literal data that is indexed by their hash values. It also makes use of a references table in order to find if a certain chunk has or has not been already sent to a certain site/client.

3.4.3 Data Transfer Protocol

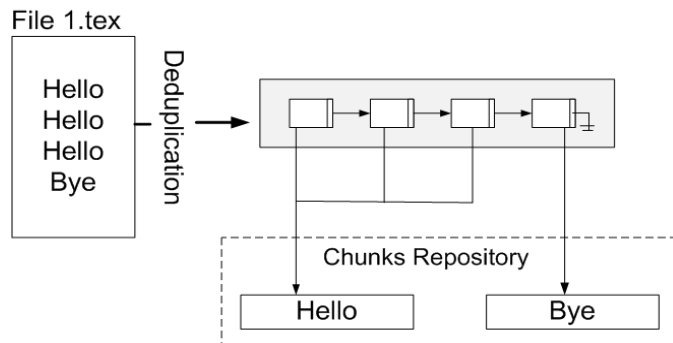


Figure 3.13: Example of the compact format of a file after deduplication.

VFCbox data transfer protocol contemplates the deduplication process on both sides. I.e., if the server receives a certain data chunk from a certain client, the current chunk will not be transferred to the concerning client in future synchronizations. From the other perspective the same happens, i.e., if the client receives a certain data chunk from the server, the current chunk will not be transferred to the server in future synchronizations. This may be accomplished since both server and clients keep references to the sent chunks in the references table.

Figure 3.13 illustrates the compact form used to represent the multiple data chunks. This compact form is constituted by a list of references to the actual chunks stored in the chunk repository. These references are composed by the hash values of the data chunks.

Figure 3.14 represents the data transfer protocol. Before sending any chunk of data, the transferring site searches the references table in order to find if the concerning chunk has or has not been already sent to the receiving site. If the chunk has not been sent, the actual content of the chunk is sent over the network. If not, only a reference (composed by the chunk's hash value) to the chunk is sent over the network. In this example we may observe that the synchronizing file is composed by 3 redundant chunks containing the content string "Hello" and a single chunk containing the content string "Bye" (Figure 3.13). When the first chunk is sent over the network, the sending chunk is not found in the chunk repository and thus the whole content of the chunk has to be transferred (literal chunk). Nevertheless, and when the second chunk is going to be transferred, the current chunk is already found in the chunks repository and in the references table. As such, the chunk is considered redundant and therefore substituted by a single reference. The same action happens with the 3rd data chunk. By last, and since the 4th chunk has not been earlier sent to the synchronizing site, it is considered as a literal chunk.

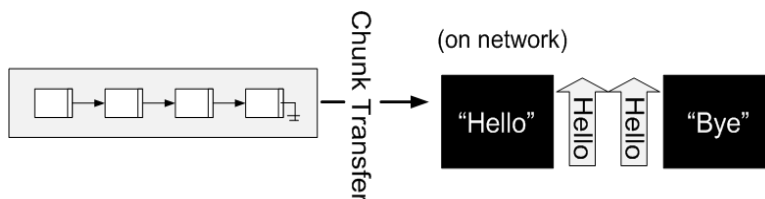


Figure 3.14: Example of the transfer of a file. This example illustrates the transfer of the compact format of the file exemplified in Figure 3.13. The black boxes represent literal data. The gray arrows represent references to the actual chunks of data.

As mentioned above, the transfer protocol relies on a references table. This references table consists on the set of hash values that each site knows to be found at a given site. As such, each time a client sends a

chunk to the server, it adds a reference in the references table. Thus, this client will know that the server also holds an identical chunk of data. The server will guarantee that it will not delete that chunk of data until the client deletes it. The same happens in the server perspective. Each time the server sends a chunk to a certain client, it adds a chunk reference in the references chunk with the concerning client identification. Thus, the server can infer if a certain chunk has been sent to a certain client by searching the references table.

3.4.4 Chunks Repository

In this section we describe in more detail the responsibility and architecture of the chunks repository. The chunks repository is responsible for storing each data chunk both at client and server side. Each chunk that is going to be transferred over the network is searched in the chunk repository. If the chunk is not found in the chunk repository, a new entry is created with the hash value and the content of the chunk. In this case, we know that we are in presence of a literal chunk, and therefore the whole content of the chunk has to be transmitted over the network to the synchronizing site.

If the chunk is found in the chunk repository, we have to search the references table (described in Section 3.4.5) in order to find if the current chunk has already been sent to the synchronizing site.

Figure 3.15 illustrates a set of files constituted by lists of references to the chunk's hash values. These chunks are stored in the chunks repository and are indexed by their correspondent hash values. Thus, the chunk repository is composed by a hash map, which uses chunk's hash values to index chunk's contents.

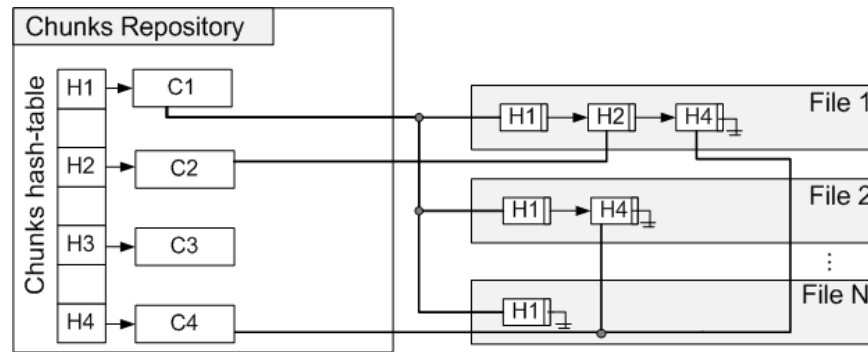


Figure 3.15: Example of the contents of multiple files. The internal representation of each file is constituted by a linked-list of references to the chunk's hashes.

3.4.5 References Table

In this section we describe in more detail the responsibility and architecture of the references table.

The references table is responsible for maintaining information about the data chunks that have already been sent to each known site. As such, when a data chunk is being transferred to a certain site, a reference to it is added in the references table. This allows to perform lookup operations in future in order to detect if a certain data chunk has already been sent to a certain site. Therefore, the references table is the actual component that allows the deduplication between synchronizing sites.

Each client contains a set of references to chunks that have already been sent to the server. This allows the client to know which data chunks have already been transferred to the server, avoiding by this the future transfer of redundant chunks.

The server contains a set of references to chunks that have already been sent to clients. Each reference contains the identification of the synchronizing site. This allows the server to know if a certain chunk has already been synchronized with a certain client.

Both, clients and server have thus to guarantee that they will not delete any data chunk from the chunks repository until no references of that chunk are found in any synchronizing site. Each client does not delete any data chunk from the chunks repository if the server is still containing a reference expressing the existence of that chunk in that client. The same happens with the server. The server does not delete any data chunk without being sure that for each client there are no references to the concerning chunk.

Figure 3.16 illustrates an example of the references table on the server side. This examples illustrates the existence of references to multiple chunks in 3 different clients. This means that the data chunks with the hash values H1, H2 and H3 have already been sent to Client 1. The same happens with Client 2 w.r.t. the hash values H5 and H3.

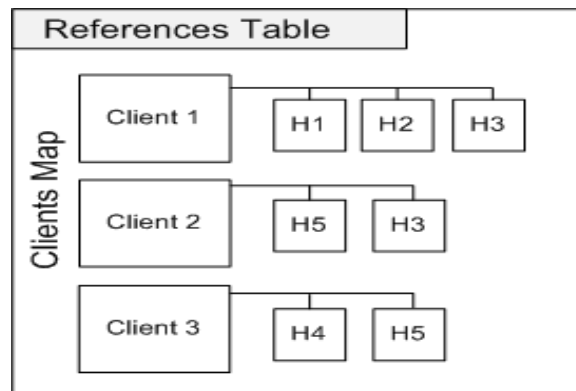


Figure 3.16: Example of the references table on the server side. The example represents that the Client 1 has references to the hash values H1, H2 and H3, which means that the data chunks with those hash values have already been sent to the concerning client.

Therefore, the references table is composed by lists of hash values of the correspondent chunks that have already been transferred to a certain site. Each hash value list is thus assigned with the identification of the correspondent synchronizing site.

3.5 Data Compression

In this section we describe the used method to compress and decompress each transferring data chunk.

The compression process is only performed in the client side. Thus, the server only deals with compressed data chunks. The server only receives chunks that have already been compressed by the client side. Then, the server stores those compressed chunks for further synchronization processes. When the server transfers a chunk to a certain client, the client is responsible for decompressing the literal data. The duty of the data compression module (only present in the client side) is thus to compress the contents of each literal chunk that is being transferred and to decompress each literal chunk that is being received.

To perform the compression we opted to use the BZip2 algorithm² due to the high compression rates that may be achieved. Additionally, and in contrary to other algorithms (e.g. deflate algorithm), this compression

²<http://www.bzip.org>

algorithm is capable of achieving high compression rates even when in presence of random data or binary data.

3.6 Consistency Model Architecture

In this section we describe the details of the implemented consistency model. First we describe how we manage the consistency meta-data and how we identify different versions of the same objects. Further, we describe how the used consistency model improved the file sharing system w.r.t. the reduction of conflicts.

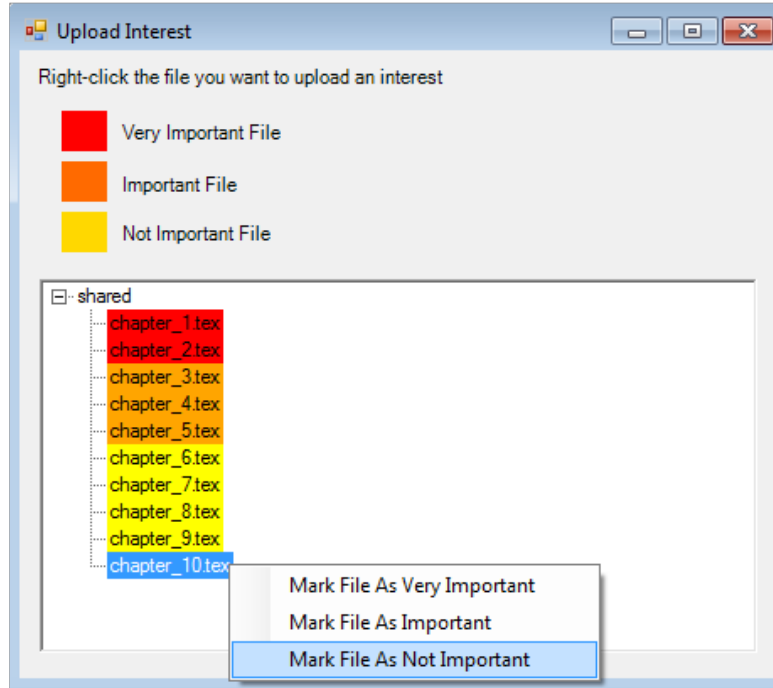


Figure 3.17: VFCbox interface, in which users may specify special interests over certain files of a shared folder.

As already mentioned, to accomplish the multiple consistency requirements we use the VFC model to apply a relaxed consistency over the sharing files. In order to apply the VFC model, the system takes into account 2 levels of users interests:

- i) file interests;
- ii) content file parts interests.

The first type of interests is related to the interests that a user may have over certain files. With this interest specification, a user may specify which are the files he wants to guarantee stronger or weaker consistency guarantees. The consistency guarantees are thus applied over the whole file.

Figure 3.17 illustrates an example where users may indicate these interests. In this example, a book with several files is illustrated. The user identified two files, in which he has a stronger interest (Files chapter_1.tex and chapter_2.tex). Three other files were also marked as important files (Files chapter_3.tex, chapter_4.tex and chapter_5.tex) and five files as not important files. Therefore, and in accordance with this interests specification, this user will have stronger consistency guarantees applied over the two first files

(Files chapter_1.tex and chapter_2.tex). Weaker consistency guarantees will then be applied over the other files according to each consistency level.

The second type of interests is related with the interests that a user may have over parts of a certain file. With this type of interests, a user may indicate a special interest over a part, which we call a semantic zone, of a document file. Consistency guarantees would then be stronger as close to the indicated semantic zone, i.e., as close to the selected pivot zone.

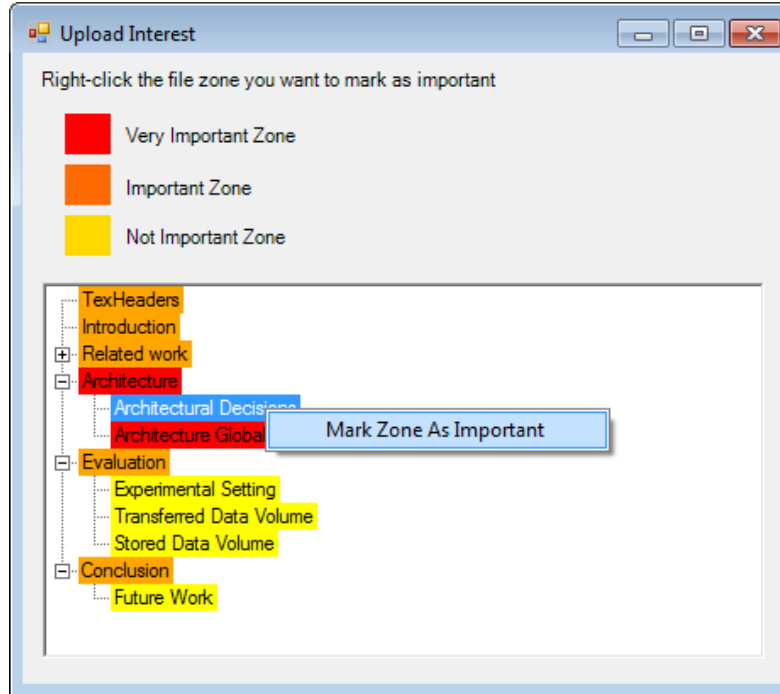


Figure 3.18: A Latex file divided in chapters and sections defining multiple consistency zones. This example illustrates an interface in which users specify their interests over parts of a Latex file.

Figure 3.18 illustrates an example where users indicate these interests. In this example, the user defined a section that is considered as important in the document, in which is required stronger consistency guarantees. The rest of the sections are then considered as less important, and therefore have applied weaker consistency guarantees. In this example, 3 consistency zones are identified. These zones are classified as follows and have applied an according consistency guarantee level:

- i) *very important zones*;
- ii) *important zones*;
- iii) *not important zones*.

As just mentioned, we defined three levels of interest. Less than three levels would force users to decide either to have strong or weak consistency guarantees, which is not our intention. By the other hand, we could have defined more than three levels of importance, nevertheless, it would be much difficult to the user to decide which level to assign. Further studies could be made in order to find out which is the best number of interest levels. However that is not the intent of this specific work.

Similarly to the previous example (Figure 3.17), in this example the user will have stronger consistency guarantees applied over the most important zones and weaker consistency guarantees as the importance of the semantic zones decreases. However, and contrary to the previous example, in this case, the multiple consistency levels are enforced over parts/zones of a file and not to the whole content of the file.

To accomplish the above mentioned, it was required some method to represent document files divided in parts (semantic zones). Figure 3.19 represents a file divided in semantic zones. These semantic zones may be seen as natural parts of a document, for instance, chapters and sections of a Latex document. As such, for VFCbox, a file is viewed as a list of semantic zones to which different consistency guarantees may be applied. According to each user interest, each semantic zone is associated with a consistency level and assigned with some VFC limits, namely sequence (σ) and time (θ) limits.

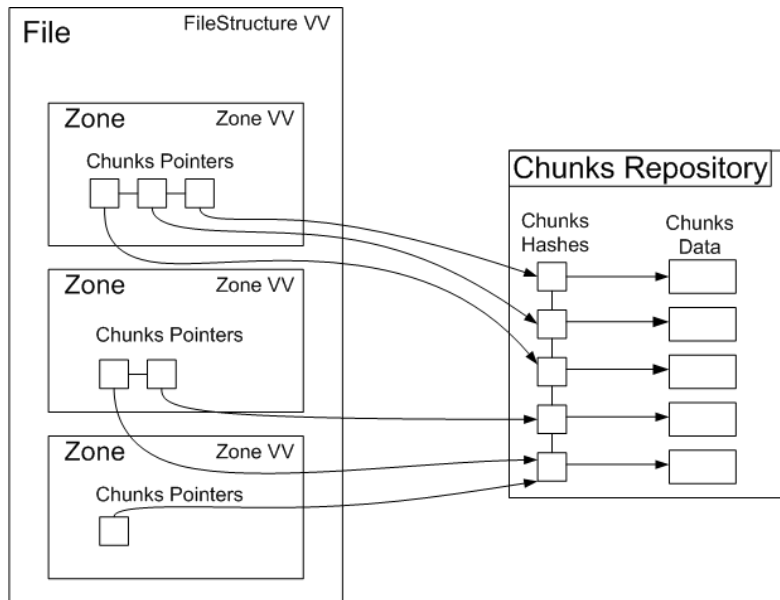


Figure 3.19: Example of a file format to VFCbox. A file is composed by a list of zones and each zone is composed by a list of references to chunks.

Each semantic zone (e.g. chapter/section of a file) is composed by data that is deduplicated. As such, for our system a file is viewed as a list of semantic zones in which each zone is composed by a list of references (chunks pointers) to data chunks. As represented in Figure 3.19 the data chunks are stored in the chunk repository.

In order to apply different consistency guarantees over multiple consistency zones of a file and consider them as independent objects, we mark each object with a version vector containing one version stamp per each know site. The version stamps are integer counters that are incremented by one unit for each new update.

Each file has one version vector for the structure of the file and one for each semantic zone. This allows the detection of modifications on the file structure (e.g. insertion of one zone) or on the content of each semantic zone.

Client nodes contain version vectors with two entries, one entry to the own version and one entry to the server version. Server nodes contain version vectors with $N+1$ entries, one entry to the own version and N entries to N clients (only the clients that share the file are included).

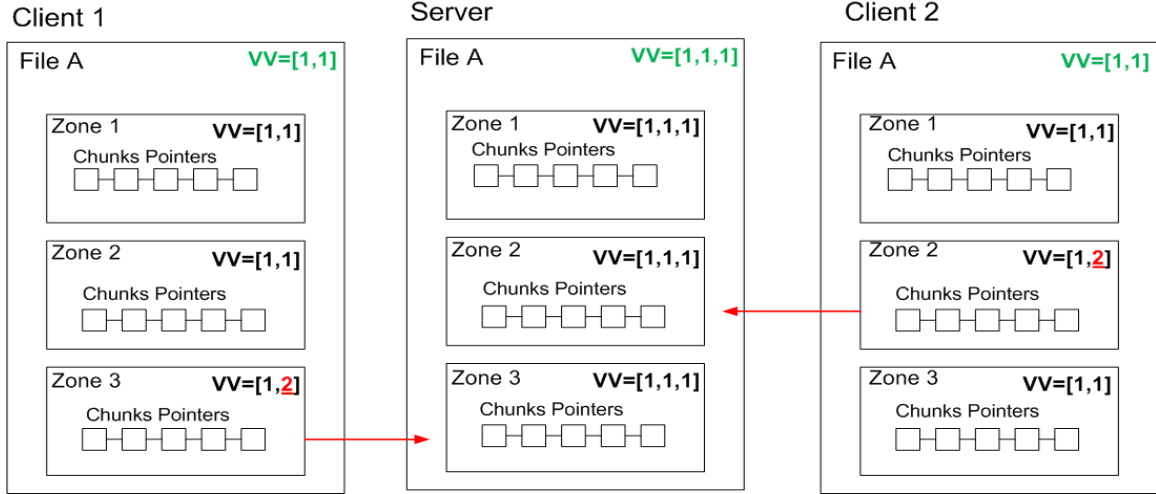


Figure 3.20: Example of two clients trying to update to the server two different zones of the same file. The underlined version stamps represent the new updates. The version vector at the top of each file represents the version of the document structure.

Figure 3.20 represents an example of two clients updating the same file to the server. Both clients share the same structure of the document, which can be seen by the version vectors ($VV=[1,1]$). Client1 is updating the 3rd semantic zone, in which his version vector is $VV=[1,2]$ / $VV=[\text{Server}, \text{Client1}]$. This means that the server is with one update in delay (version stamp = 1 in comparison to version stamp = 2). Client2 is updating the 2nd semantic zone, in which his version vector is $VV=[1,2]$ / $VV=[\text{Server}, \text{Client2}]$. Both semantic zones at the server are marked with the version vector $VV=[1,1,1]$ / $VV=[\text{Server}, \text{Client1}, \text{Client2}]$.

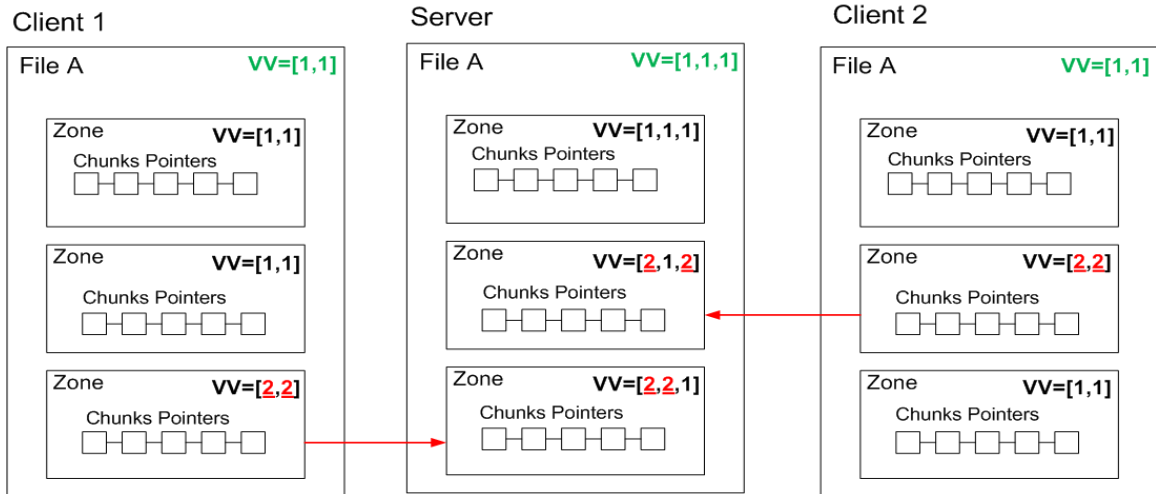


Figure 3.21: Example of two clients after updating to the server two different zones of the same file. This exemplifies the result obtained by example represented in Figure 3.20. The underlined version stamps represent the new updates (already synchronized to the server). The version vector at the top of each file represents the version of the document structure.

Figure 3.21 represents the result of the submission of both updates (from Client1 and Client2) to the server. By representing parts of a file as independent objects, we are not only capable of enforcing multiple consistency levels but also of avoiding conflicts. This is accomplished since we assign a version stamp per

each semantic zone. As such, modifications to different zones of a file are viewed as updates to different objects and thus not considered as a conflict. This situation may also be seen in this figure, where two clients updated the same file without creating a conflict.

3.7 VFC Enforcement

In this section of the document, we describe how the VFC model is enforced over the multiple updates. Information about how the system assigns different consistency levels to different zones of a document is also presented in this section. Additionally, details about the checking of VFC limits and used limits to bound the divergence between replicas are presented in this section.

After receiving an update, the server is responsible for enforcing the VFC model. The VFC model Manager is in charge of determining the consistency level of the current update, according to each user interest. Each semantic zone update is then assigned with a consistency level, a set of divergence bounds/limits and a set of clients to whom this update may concern.

After having assigned a consistency level, each semantic zone is inserted in a list of semantic zones. There are three semantic zone lists, one for each consistency level. Nevertheless, the system is prepared to be configured to have more consistency levels. Periodically, each list is iterated in order to update the time parameter and verify if any semantic zone has already exceeded the time limit to be propagated to a certain client. For instance, from x to x seconds, we update the time parameter of each semantic zone by x seconds. When the time parameter of any object exceeds the specified time limit to that consistency level zone, the object is elected to be propagated.

Identically, when a new update is received, the sequence parameter of the affected semantic zone is incremented by one unit. Then, this sequence parameter is compared to the sequence limit specified to that zone. If the sequence parameter exceeds the this limit, the object is elected to be propagated.

3.7.1 Consistency Level Assignment

After receiving an update to a certain semantic zone, the server has to assign a consistency level to it. As already mentioned, there are 3 different consistency levels, namely *very important zones* level, *important zones* level and *not important zones* level. To assign a certain level to a certain semantic zone update, we have to calculate the distance of the concerning zone to the pivot zone.

$$distance(Semantic_Zone; Pivot_Semantic_Zone(Client_X))$$

The distance calculation is made by calculating the number of zones that are between the semantic zone and the pivot semantic zone. Therefore, a distance of 3 units means that there are 3 semantic zones between the semantic zone and the pivot semantic zone.

3.7.2 VFC Limits

To bound the divergence between clients, a set of limits of the VFC model are imposed.

From the three main divergence criterias of the VFC original work, we decided to use two of them, namely Sequence (σ) and Time (θ). The Sequence criteria indicates the number of updates that have already been performed over a certain file. Each file save is considered to be as an update.

We decided not to use the Value (ν) divergence criteria since this criteria could have no meaning to users. For instance, a modification to the file formatting could indicate a high percentage of modification,

nevertheless the content of the sharing document is maintained.

The following table (Table 3.7.2) describes the default limits of divergence imposed by the VFC model. These limits may be configured.

Semantic Zone Level	Sequence (σ)	Time (θ)
Very Important	1 update	0 min.
Important	5 updates	5 min.
Not Important	10 updates	10 min.

Table 3.1: VFC model: semantic zone limits.

These limits are enforced over each semantic zone. According to the certain level, each zone has assigned some divergence bounds. For instance, a semantic zone that is considered as very important, has associated a sequence bound of 1 update and a time bound of 0 min. which means that an update to that semantic zone would be immediately propagated to the concerning client. On contrary, a semantic zone considered as an important zone, has associated a sequence bound of 5 updates and a time bound of 5 min.. This means that only after 5 updates or only after a delay of 5 min., the semantic zone would be transferred to the concerning client.

3.7.3 Checking VFC Limits

Time

The Time limit of a VFC consistency vector defines the maximum time without seeing any updates from a zone at a certain distance from the pivot zone. To enforce the time limits, we set a mechanism that signals timeouts for each consistency zone with a period defined by the consistency vector time limit.

To create these timeout events, we connected a function to a clock signal in the server, with a period equal to the minimum unit of measurement, i.e. one second. This function counts the elapsed time and checks, for every consistency zone, if the time limit has been exceeded.

$$Elapsed_Time[Semantic_Zone] > Time[distance(Semantic_Zone; Pivot_Semantic_Zone(Client_X))]$$

Sequence

The Sequence limit is defined as the maximum number of unseen updates from a certain semantic zone. The Sequence limits are checked in consequence of either the arrival of a new operation, or a change in the user pivot. This limit is checked simply by comparing the number of undelivered operations in a certain semantic zone with the maximum sequence limit defined by the according sequence divergence limit.

$$\#Updates[Semantic_Zone] > Sequence[distance(Semantic_Zone; Pivot_Semantic_Zone(Client_X))]$$

Chapter 4

Implementation

In this chapter we describe the most relevant details of the implementation of our solution.

In this work we implemented a file sharing system capable of synchronizing the creation/modification/deletion of Latex file documents. Although our prototype only implemented the sharing of Latex files, other types of files could be implemented by adding plugins to the file system monitor in order to detect their multiple data parts and make the translation of those files to the intermediate format.

VFCbox creates a shared folder that is automatically uploaded to the server node. The client node, is responsible for monitoring the shared folder. Any files dropped into this folder are indexed, hashed and then compared to other data chunks already sent to the VFCbox's server.

Additionally, the system contemplates a notification process that alerts the client to new arriving files or new updates to specific zones of a file.

The client of this file sharing system was implemented in C# .Net. The advantages of using C# .Net are:

- Building the file system monitor easily using the *FileSystemMonitor* from the .Net framework.
- Building the transfer protocol easily using .Net Remoting.
- Building the GUI using the Microsoft Forms from the .Net framework.
- Building the right click menu on windows explorer using the .Net framework.
- Wide set of cryptographic tools (e.g. SHA-1 implementation).
- Wide set of tools to deal with XML.

4.1 Client

In this section we overview the client node implementation and describe some of its most interesting details.

4.1.1 Content-based (variable-size) Chunking

The content-based chunking algorithm is based in Rabin Fingerprints. It consists in the hash calculation of a sliding window of 48 bytes and in the comparison of the hash value with a certain value. If the hash value matches with the predetermined value, a chunk bound has been found. To limit the chunk boundaries, we have configured a lower and upper size limit. The lower limit is of 16Kb and the upper limit is of 32Kb.

These values were chosen after a set of tests to infer which limit boundaries would bring a higher performance and bandwidth reduction results.

4.1.2 Interception of Updates

To intercept client's file updates we implemented a file system monitor. This monitor is responsible for detecting changes in the file system and report these modifications to the VFCbox system. In order to implement the file system monitor, we used the *FileSystemWatcher* from the .NET framework. The *FileSystemWatcher* is capable of monitoring several modifications of a certain folder of the file system, providing information about the type modification (file creation, file modification, file deletion, file name modification, etc.), about the modification time and others.

The implemented file system monitor also had the duty of detecting fake or duplicate updates. The *FileSystemWatcher* from the .Net framework normally duplicates the modification events. This situation happens due to the modification of the contents of a file and of its attributes, which represent 2 modifications on the file system. Thus, our implementation had to detect and ignore these duplicates.

4.1.3 Shell Extension Menu Handler

To create the right click menu on windows explorer we implemented a shell extension menu handler. This was accomplished by implementing the Windows interfaces *IShellExtInit* and *IContextMenu*. Additionally, we also had to implement the registration of the multiple created handlers in the Windows Registry. Figure 4.1 illustrates one of the created menus in which users may decide to share a folder or to upload a certain interest over a certain file of the concerning folder.

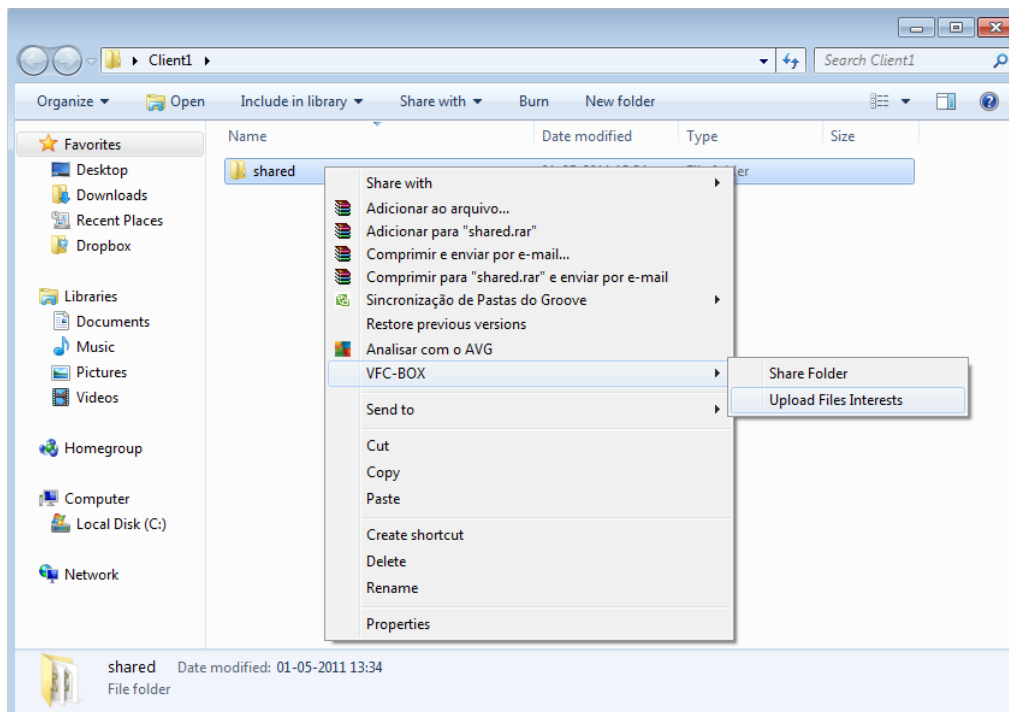


Figure 4.1: Right click menu on windows explorer exposing the VFCbox menu. In this case users may decide to share the current folder with a certain user or to upload some interests over certain files of the current folder.

4.1.4 Interests Uploading Interfaces

To upload the multiple user's interests, we implemented two interfaces. One interface is used to upload interests over whole files, i.e. users may indicate which files are more or less relevant to them. This interface is depicted in Figure 4.2.

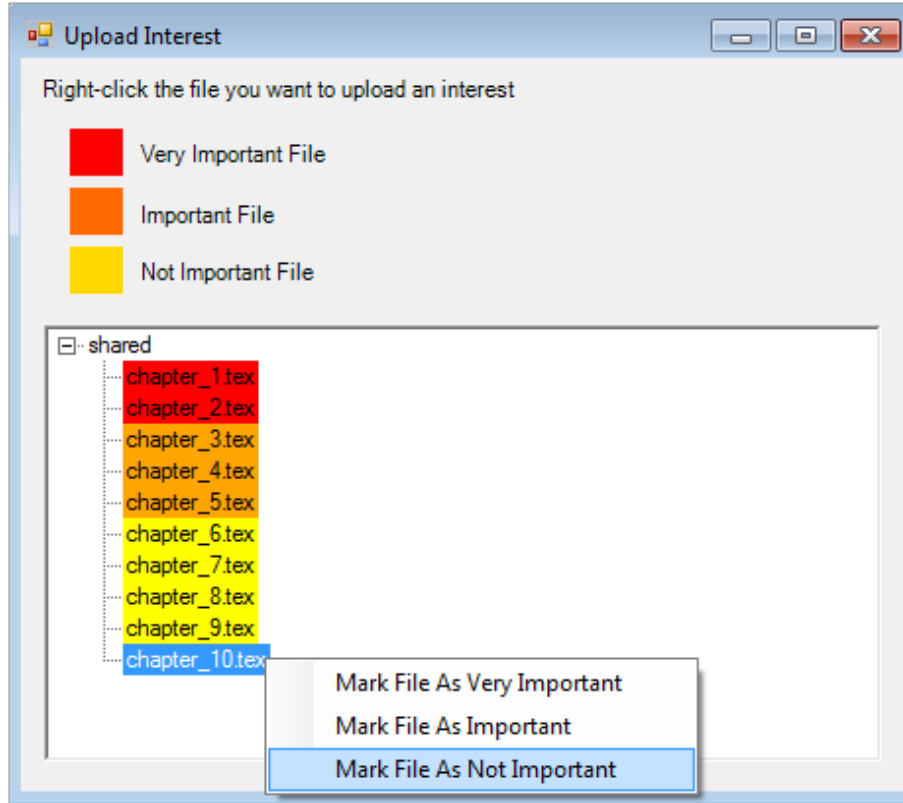


Figure 4.2: VFCbox interface, in which users may specify special interests over certain files of a shared folder.

To indicate more fine-grained interests, we implemented another interface to upload interests over specific parts of Latex documents. Since the distance between multiple parts of a Latex document depends of the document depth of each semantic zone, we implemented a special method to calculate that distance.

For instance, in Latex documents sections are deeper than chapters, since a Latex document is made of chapters and a chapter is made of sections.

As such, the distance is calculated according to the document depth of the semantic zone and the number of zones that are between the semantic zone and the pivot zone. We created the following distance calculation method:

If the pivot zone is a chapter zone, all the sections of that chapter will have assigned the *very important zone* level. The rest of the chapters will have the *important zone* level assigned. The rest of the sections will have the *not important zone* level assigned.

Figure 4.3 illustrates the above mentioned.

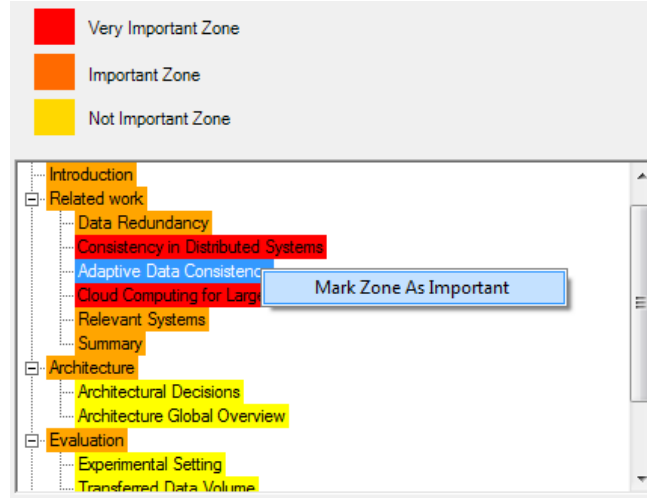


Figure 4.3: Example of the assigned consistency levels when a section of a Latex document is chosen as pivot zone.

If the pivot zone is a section or subsection zone, only the proximate sections will be considered as *very important zones*. The rest sections of the same chapter will be considered as *important zones*. The rest of the chapters will have assigned the *important zone* level. By last, the rest of the sections (contained by the other chapters) will have assigned the *not important zone* level.

Figure 4.4 illustrates the above mentioned.

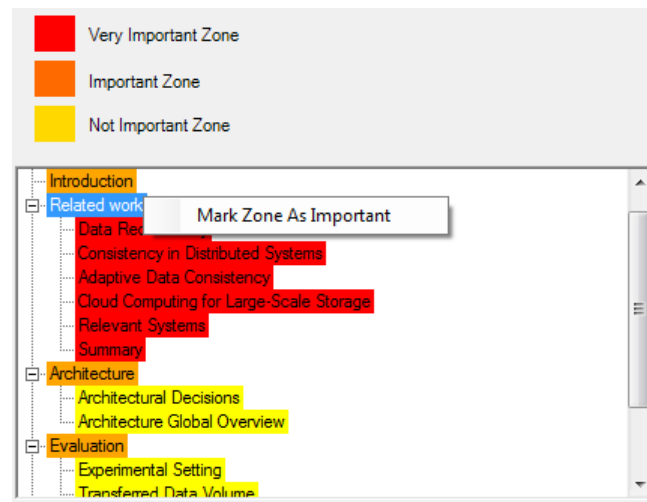


Figure 4.4: Example of the assigned consistency levels when a section of a Latex document is chosen as pivot zone.

4.2 Server

In this section we overview the server node implementation and describe some of its most interesting details.

The server runs as a daemon on the server machine. It can receive updates from several clients simultaneously. We support this by establishing a separate pipeline, for each connection.

In order to perform deduplication between different clients, the server uses the same Chunk Repository for every client. Thus, we can check which chunks are already stored and avoid storing them more than once. The references table cannot be shared between clients, thus we keep one for each client. The server serializes its structures at the end of an operation.

In order to check the vfc limits of each object to be propagated, we use a timer to periodically fire a signal. This signal creates an event that updates all time parameters of each semantic zone to be propagated. In order to detect if a certain semantic zone has to be propagated, each semantic zone time parameter is also compared to the limit imposed by the consistency level of the semantic zone.

Chapter 5

Evaluation

In this chapter we present an analysis of our prototype, comparing it to Dropbox, SVN and LBFS, both qualitatively and quantitatively.

First, we describe some of the main advantages of the VFCbox system in qualitative terms.

Then, we present the results of the quantitative analysis regarding the bandwidth usage gains with the deduplication and the relaxed consistency model.

5.1 Qualitative Evaluation

VFCbox system may be used in similar a way to the Dropbox system. It is viewed as a synchronized folder where users may drop files/folders and may update them without having to explicitly synchronize them. When a new update is received, a notification in the icon bar is presented, identifying which file and zone was updated.

Since the system may be used similarly to Dropbox, we decided to qualitatively compare our prototype to this system.

In comparison to Dropbox, we improved the file sharing system w.r.t. application usability and file conflicts occurrence/resolution.

The main objective of this evaluation is to provide an analysis over the user's experience regarding file conflicts occurrence and respective resolution. Additionally, we also present another qualitative improvement: Data Latency Control.

5.1.1 Evaluation of File Conflicts Occurrence/Resolution

Comparing to Dropbox, the VFCbox prototype was improved to reduce the conflict rate of Latex files.

The VFCbox prototype is able to achieve this since it treats chapters and sections as independent objects of a Latex file, considering them as multiple consistency zones of the VFC model. As such, the version system assigns multiple versions to multiple chapters/sections of a Latex document.

This enables the modification by users of different chapters/sections to not be seen as a conflict. Therefore, a concurrent modification over different chapters/sections of a file are viewed as updates to different objects and thus, the updates are merged in the same file.

Figure 5.1 illustrates an example of the described situation, where two users are modifying the same document, and still no conflict occurs. Since the modifications are done in different sections of the Latex document, these are naturally merged and propagated to both users. Systems like Dropbox consider multiple modifications by different users to a file as a conflict, requiring users to solve the conflicts. This improvement of the VFCbox enables users to concurrently modify and change files, reducing the conflict rate.

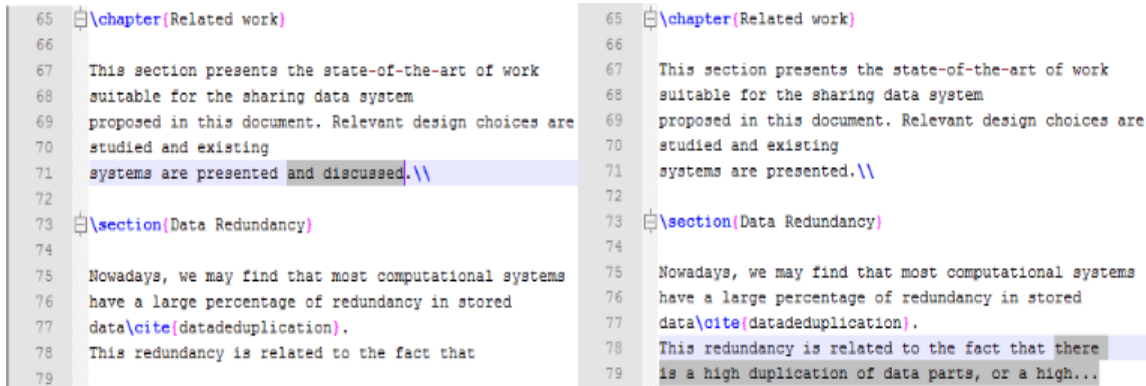


Figure 5.1: Two users modify the same document, but since they are modifying different sections of the document, it is not considered a conflict and the modifications are naturally merged.

However, when multiple users modify the same section of a file, this is treated as a conflict. As such, the system behaves like the Dropbox system, creating a conflict and requesting users to solve it. Nevertheless, and in contrary to systems like Dropbox, we provide an interface to help users in the resolution of conflicts. Figure 5.2, 5.3 and 5.4 illustrate the three steps of that interface. The interface uses an external application (WinMerge¹) to identify the differences between the two versions and help the user to resolve the multiple differences.

On the following we present the multiple steps of the interface used to resolve a file conflict:

Resolve a File Conflict - Step 1

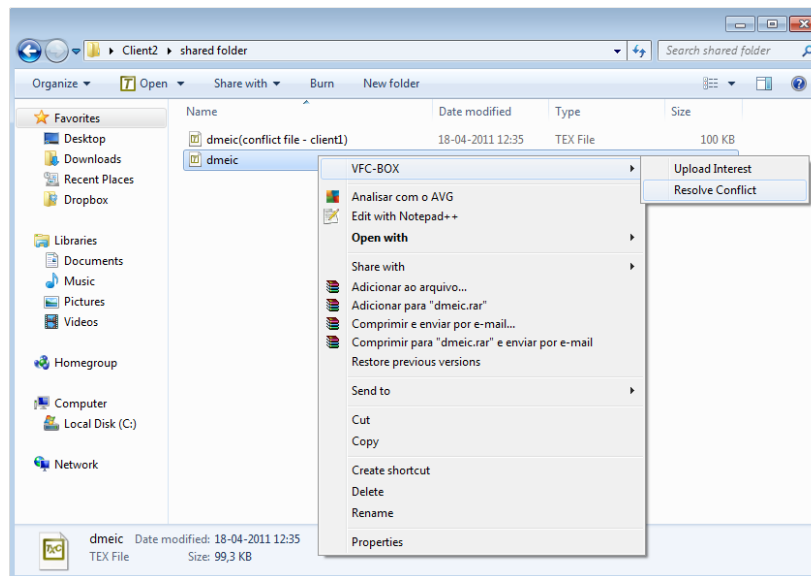


Figure 5.2: Resolve conflict - step 1: select the file to resolve the conflict.

¹<http://winmerge.org>

Resolve a File Conflict - Step 2

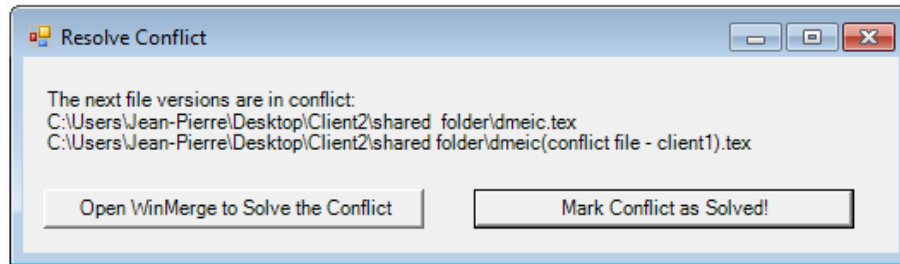


Figure 5.3: Resolve conflict - step 2: chose to open WinMerge to resolve the conflict or mark the current version as the resolved version.

Resolve a File Conflict - Step 3

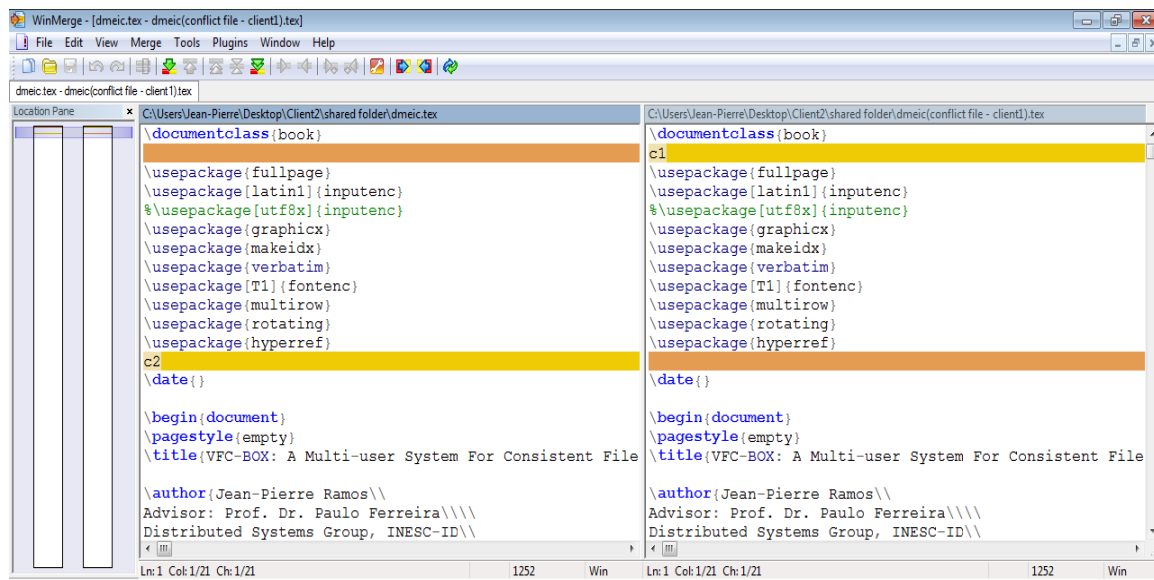


Figure 5.4: Resolve conflict - step 3: use WinMerge to resolve the conflict.

5.1.2 Data Latency Control

Another qualitative feature that is present in our prototype is related with the possibility of managing the latency in the synchronization of certain files. As already mentioned, the VFCbox solution allows the specification of interests over certain files or parts of a certain file, through an interface adapted to perform these specifications. Through this interests specification, the system is able of assigning different consistency guarantees over multiple files or file parts. Therefore, updates with higher importance levels are forwarded to users in advance of updates with lower importance levels.

5.1.3 Summary

In this section we present and discuss the qualitative advantages (in comparison to Dropbox) of VFCbox. After the analysis of these advantages, the following conclusions can me made:

- **VFCbox is capable of reducing the conflict rate and foster the concurrent work.** In our prototype a file is viewed as a set of independent objects. As such, users may modify and change

some of these objects without affecting the others. This enables the concurrent work and reduces the possibility of occurring a file conflict.

- **VFCbox provides a more user-friendly interface to resolve conflicts.** Our prototype provides to users an interface in which users may see the disparities between the conflicting versions and resolve the differences in an ease manner.
- **VFCbox provides to users the possibility of controlling multiple consistency levels and thus to control the latency of certain files.** In our prototype, users may specify interests over certain files or parts of a certain file. With this interests specification, users may control the latency seen in the synchronization of certain files. This may be achieved since updates with higher importance levels are forwarded to users in advance of the other updates.

5.2 Quantitative Evaluation

In this section we present the results of some conducted tests. These tests had the objective of determining the savings in the use of network resources.

5.2.1 Experimental Setting

To evaluate the system quantitatively, we simulate a user performing modifications to shared data. These modifications are then propagated to the server, which is then in charge of synchronizing the new updates with the other users. Three dedicated machines were required. Two of these machines were used to simulate two different VFCbox users that wanted to synchronize several data files. One of these machines was also used to intercept and measure the network traffic. The network measures were always performed on the machine that is receiving/downloading new updates. Another machine was used for the VFCbox server, which performed synchronization operations and conflict detection between clients. All the three machines were running Windows 7 Professional (32-bit) on a AMD Turion 64 Mobile Technology ML-37 2.00GHz with 1Gb of RAM. Finally, a 100mbps full-duplex Ethernet LAN was used for the tests.

5.2.2 Compared Solutions

For comparison with VFCbox's results we chose three widely different file sharing systems, namely the Dropbox², the LBFS[23] and the SVN[15].

- **Dropbox** - Delta-encoding Techniques.
- **LBFS** - Compare-by-Hash (Variable-size Block Hashing).
- **SVN** - Delta-encoding Techniques.

We chose this three systems due to the fact of being significantly representative of the state of the art techniques used by most file sharing systems. The comparison of the implemented prototype with these systems permitted to compare 2 main issues:

- **Deduplication** - compare the deduplication results with the results of widely used systems.
- **Total Consistency versus VFC** - compare the results of the VFC model against the traditional total consistency model (used by most file sharing systems).

²Dropbox: Secure backup, sync and sharing made easy. <https://www.dropbox.com>.

5.2.3 Workload Description

To exercise the system, several tests were performed using two different kinds of workloads, namely synthetic samples and master's thesis samples.

The synthetic samples were composed by a workload of 5 different Latex file samples, representing files of different sizes. Each file sample was composed by 20 chapters and 100 sections (5 sections per chapter). The content of each section was filled with random data. On the following we describe the proximate size of each file sample: **1.tex**: 1 MB (1.024KB); **2.tex**: 5 MB (5.120KB); **3.tex**: 10 MB (10.240KB); **4.tex**: 50 MB (51.200KB); **5.tex**: 100 MB (102.400KB).

The master's thesis samples were composed by 10 different Latex files, representing the content of some MSc dissertations of students from Instituto Superior Técnico. These samples were composed by the regular chapters of a dissertation, namely *Introduction*, *Related Work*, *Architecture*, *Implementation*, *Evaluation* and *Conclusion*, and several sections. The medium size of each Latex file was of about 200KB. Thus, this workload has a total proximate size of 2000KB.

5.2.4 VFC Limits

To evaluate the system, the following configuration values of the VFC model were selected:

Zone	Sequence (σ)	Time (θ)
Very Important	1 update	0 min.
Important	5 updates	5 min.
Not Important	10 updates	10 min.

Table 5.1: VFC model: zone limits settings.

5.2.5 Bandwidth Usage Analysis: File Download Stress Test

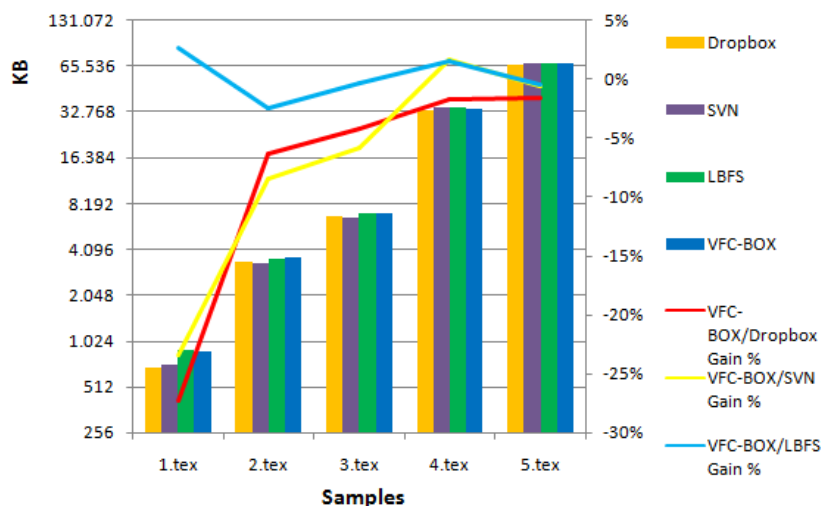


Figure 5.5: Bandwidth usage (in KB) of downloading 5 different file samples.

We start by discussing the results obtained for the basic test of downloading an entire file. These results are illustrated in Figure 5.5 (the results table is presented in Appendix). This workload is comprised of 5

Latex document files that have already been described in Section 5.2.3. The results present the comparison between the bandwidth used by VFCbox and three other solutions to download 5 file samples in the client side.

The presented graph shows that for all the solutions, the bandwidth used to download each file sample is always inferior to the total file size. This is explained with the usage of compression techniques and not to any type of deduplication, since for this specific test, these samples do not contain any type of redundancy.

The results of this test show that the VFCbox has almost the same performance as the other solutions, having a overload (between 27% and 2% of overload) that becomes more dissipated with the increase of the file size. This overload of the VFCbox system is explained by an increased meta-data transmission, since unlike Dropbox and SVN it has to transmit some extra meta-data related with the chunk's hash values. Comparing to LBFS, the results are very similar, which was expected, since both use the same deduplication techniques and have to transmit extra meta-data w.r.t. Dropbox and SVN.

5.2.6 Bandwidth Usage Analysis: File Update Stress Test

This stress test presents an analysis to the bandwidth used to download a file update. This represents the ability of the system to perform deduplication between multiple versions of the same file. As such, we performed some minor changes (insertion of 1 byte) to files that have already been synchronized and then measured the amount of bandwidth that was used to propagate the modifications/updates. The workload was comprised of 5 Latex document files that have already been described in Section 5.2.3.

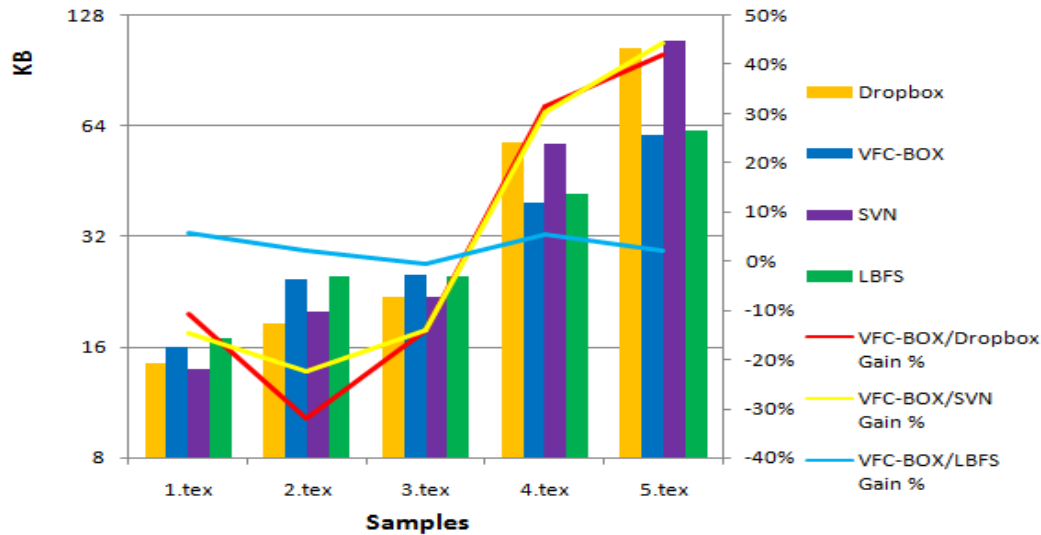


Figure 5.6: Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the beginning of each file.

Figure 5.6, 5.7 and 5.8 present the used bandwidth to download multiple file updates (the results tables are presented in Appendix). These file updates consist in the insertion of 1 byte on the beginning of each file (5.6), on the middle of each file (5.7) and on the end of each file (5.8).

Since Dropbox and SVN makes use of Delta-encoding techniques, this test obviously favors them due to

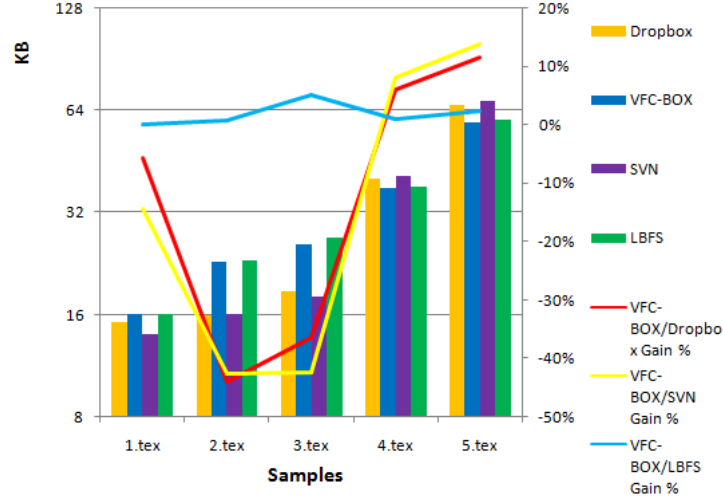


Figure 5.7: Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the middle of each file.

the updates being composed by minor changes (single insertion of 1 byte). In the case of VFCbox and LBFS, the insertion of 1-byte modifies at least one entire chunk, which results in the transmission of an entire chunk plus the rest of the hash values of the remaining chunks. Nevertheless, VFCbox and LBFS are still capable of using less bandwidth for some cases, more specifically for files of bigger sizes. Once more, we may see the similarity between the obtained results by VFCbox and LBFS.

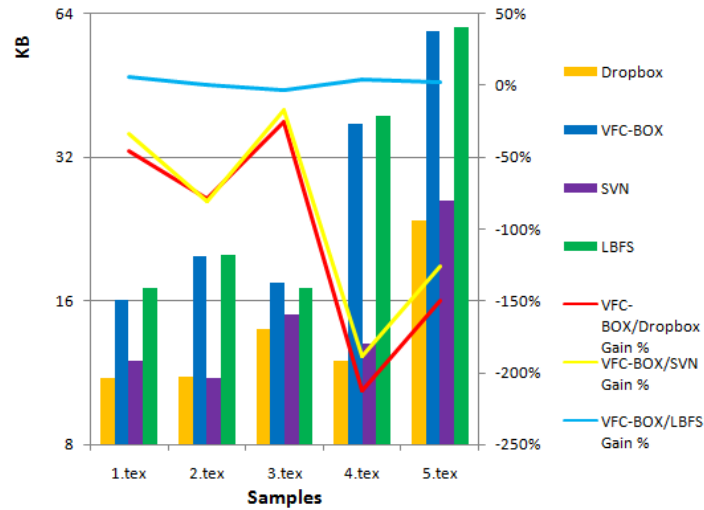


Figure 5.8: Bandwidth usage (in KB) of downloading 5 different file updates. The updates consisted in the insertion of 1 byte in the end of each file.

Although there is a high overload percentage of VFCbox in comparison to the Dropbox and the SVN system, this stress test contemplates the worst case to VFCbox and the best one to the other two solutions. Additionally, in absolute terms the overload is not so significantly since it only occurs for small changes and small amounts of exchanged data (in this test in the order of dozens of kilobytes).

5.2.7 Bandwidth Usage Analysis: Deduplication Stress Test

This stress test is composed by the download of multiple iterations of redundant file samples. The sizes of the file samples are described in Section 5.2.3. These file samples were increasingly filled with redundant data. To reproduce the redundant data, random sections were repeated through the Latex documents according to the redundancy percentage. Figure 5.10, 5.11, 5.12 and 5.13 illustrate the bandwidth used by the compared solutions, to download each iteration of the 5 file samples (the results tables are presented in Appendix). On each iteration the redundancy percentage is increased by 10%.

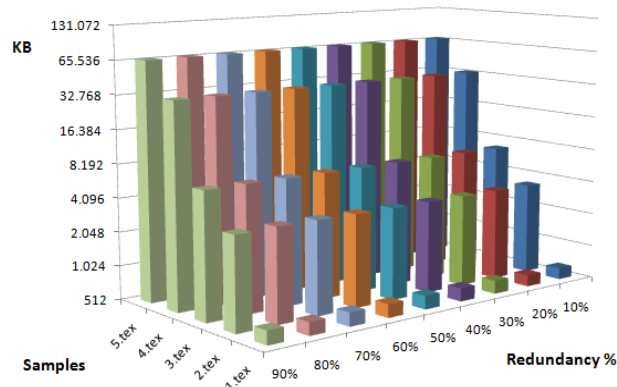


Figure 5.9: **Dropbox's** bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.

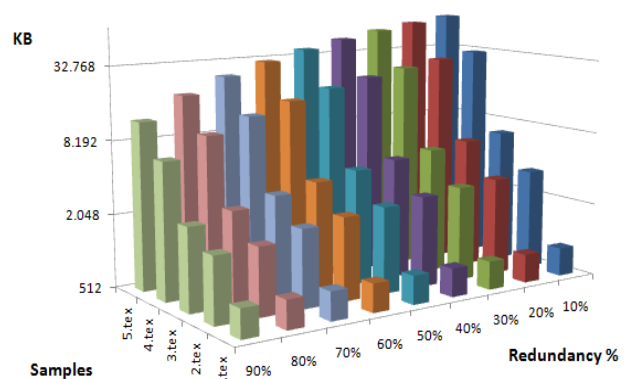


Figure 5.10: **VFCbox's** bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.

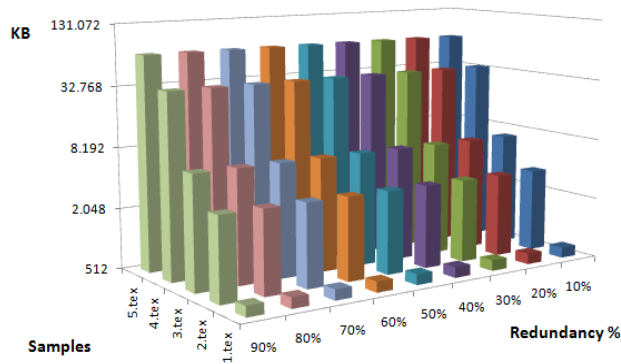


Figure 5.11: **SVN's** bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.

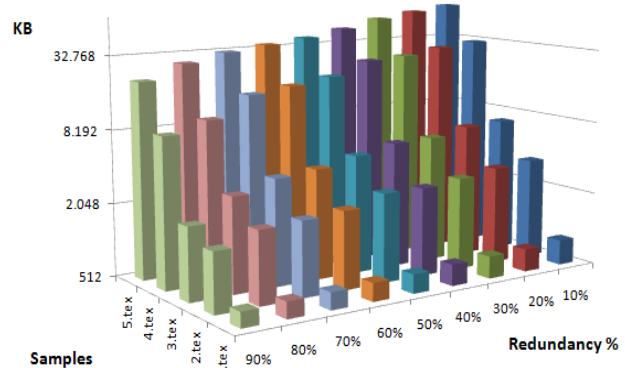


Figure 5.12: **LBFS's** bandwidth usage (in KB) of downloading several iterations of 5 different file samples. Each iteration was composed by files with different percentages of redundancy.

These results show that the Dropbox and the SVN systems use always the same bandwidth to download each file (Figure 5.9 and 5.11), i.e., either for files with 10% of redundancy or 90% of redundancy, the same bandwidth is used. For VFCbox and LBFS, the same does not happen. As the redundancy percentage increases, the used bandwidth decreases (Figure 5.10 and 5.12).

With this analysis we may conclude that the Dropbox and the SVN system are unable to detect any redundancy in these files since they are only capable of detecting redundancy between multiple versions of the same file. VFCbox and LBFS in turn, are capable of detecting this redundancy having bandwidth savings proximate to the redundancy percentage.

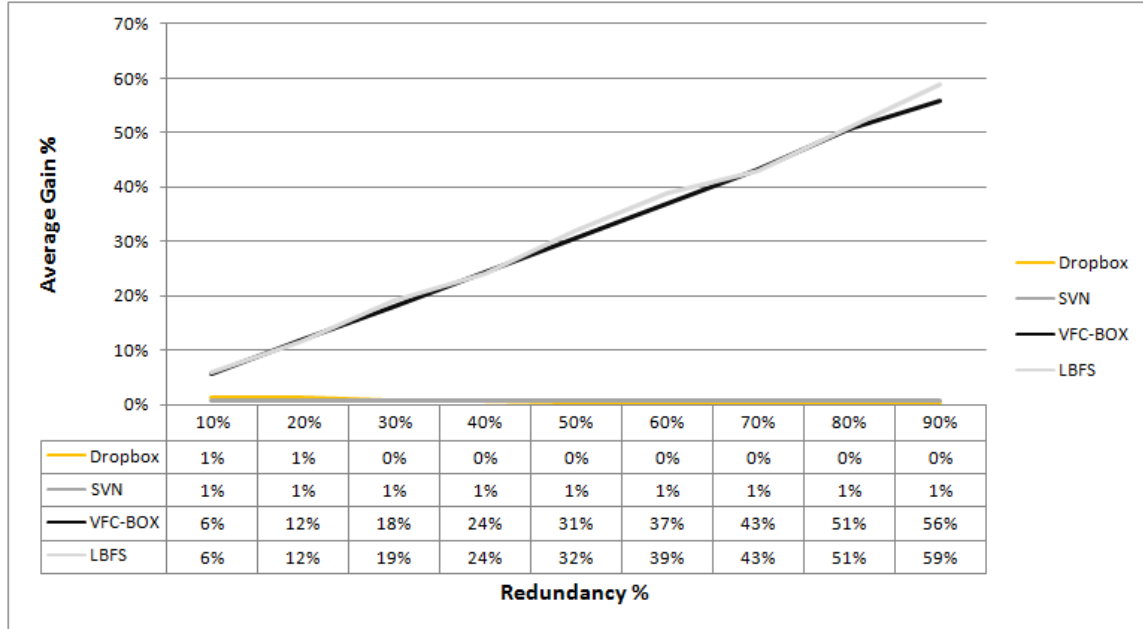


Figure 5.13: Average gain percentage of bandwidth usage to download sets of samples with different redundancy percentages.

Figure 5.13 illustrates the above mentioned results in a percentage gain perspective.

Once more, these results illustrate that Dropbox and the SVN have almost 0% of bandwidth reduction in all iterations.

VFCbox and LBFS in turn are capable of making savings of more than 55%.

5.2.8 Bandwidth Usage Analysis: Consistency Model Stress Test

The objective of this test is to stress the consistency model and measure the reduction on bandwidth resources by using the VFC model.

To accomplish this, the VFCbox client that is downloading new updates, assigned to each file sample an interest over a certain section of the document.

With this explicit interest of the client, 3 sections of each document were considered as very important sections, 22 sections were considered as important sections and 75 sections were considered as not important sections.

This test is composed of 10 file updates of 5 file samples described in Section 5.2.3. These file samples do not contain any redundancy. The file updates consist in a total replacement of the content of each file.

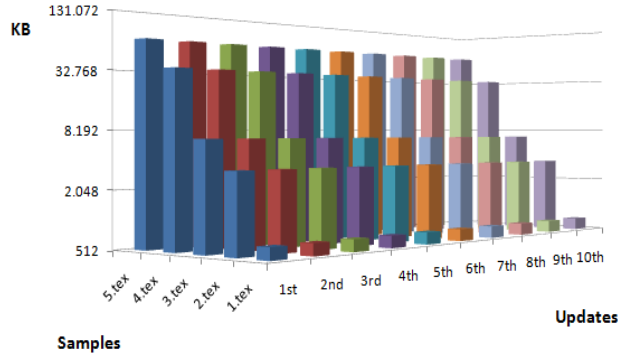


Figure 5.14: **Dropbox's** bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.

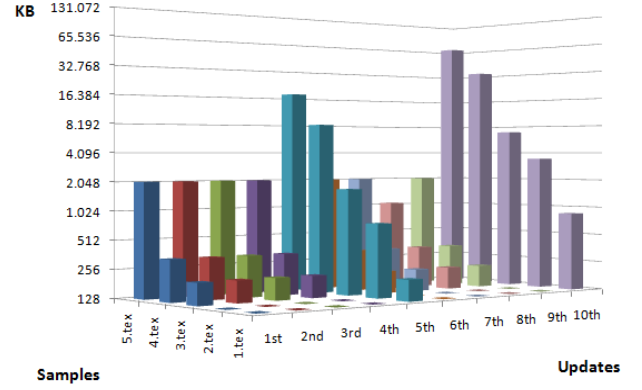


Figure 5.15: **VFCbox's** bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.

Figure 5.14, 5.15, 5.16 and 5.17 illustrate the results of this analysis of the consistency model (the results tables are presented in Appendix).

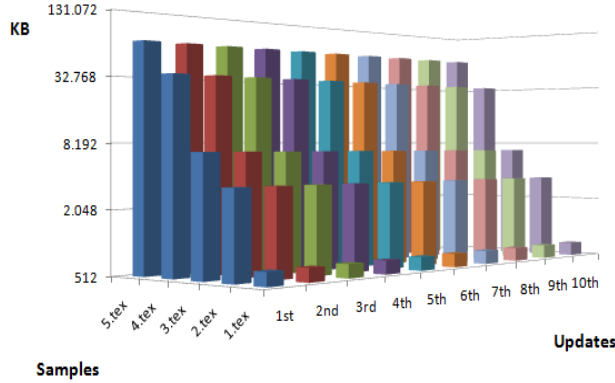


Figure 5.16: **SVN's** bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.

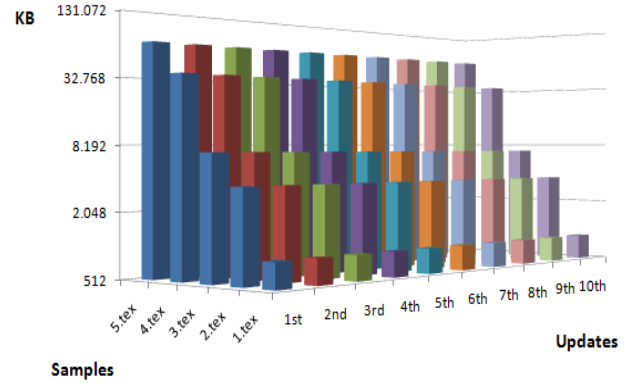


Figure 5.17: **LBFS's** bandwidth usage (in KB) of downloading 10 file updates of 5 different file samples. The file updates were composed by a total replacement of the content of each file.

Given these results, we may conclude that the Dropbox, the SVN and the LBFS system, make use of the same bandwidth to download each file update (Figure 5.14, 5.16 and 5.17). This was an expected result since it means that for each update the whole content of the file was transmitted. Analyzing the results of VFCbox (Figure 5.15), we may find that there are 3 distinct steps on the bandwidth usage results. These steps represent the multiple consistency levels of the VFC model. The lowest step (1st to 4th update and 6th to 9th update) represents the downloading of the sections considered as very important. Thus, a reduced amount of bandwidth is used. The median step (5th update) represents the downloading of the sections considered as very important and important. This is explained with the fact that on the 5th update the vfc sequence limit, of the important sections region, is exceeded. The biggest step (10th update) represents the downloading of all the sections (very important, important and not important sections). This is explained with the

fact that on the 10th update the VFC sequence limits of all regions are exceeded. At this point, the client that is downloading the new file updates, downloads the whole file making it consistent with the other clients.

Comparing the three other systems to the VFCbox system, we may conclude that the VFCbox may achieve a higher performance in terms of the consistency model and w.r.t. bandwidth resources reduction. Our prototype is capable of delaying some specific updates that clients pointed as updates of lower interest. By delaying the updates, they eventually are replaced by other updates making it unnecessary to transmit them.

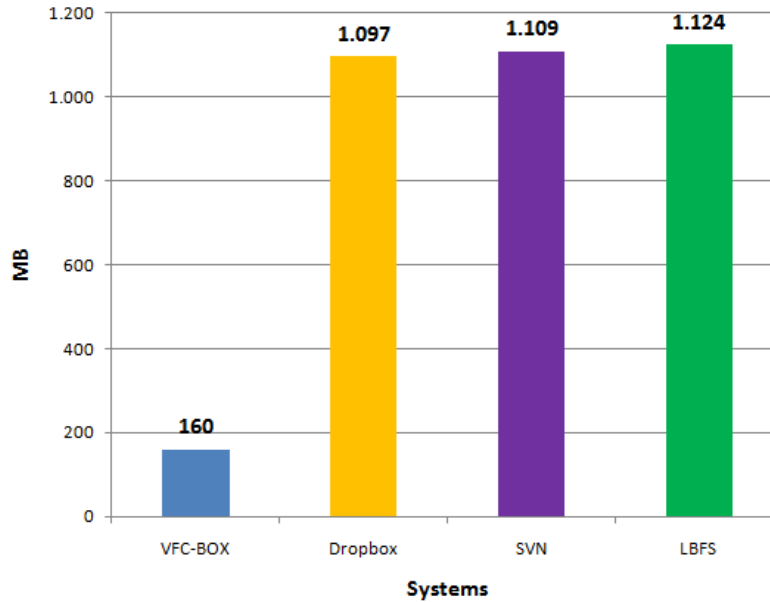


Figure 5.18: Total used bandwidth (in MB) to download 10 file updates of 5 different file samples. It represents the sum of the used bandwidth of the stress test illustrated in Figures 5.14, 5.15, 5.16 and 5.17.

Figure 5.18 illustrates the total sum of the used bandwidth of the above mentioned results (Figures 5.14, 5.15, 5.16 and 5.17, the results tables are presented in Appendix).

By these results of the sum of the total used bandwidth we may compare the total used resources between VFCbox and the other solutions. The VFCbox savings after 10 file updates are proximate to 85%. Since all the other three solutions use a total consistency model, they all have more-less the same performance.

5.2.9 Bandwidth Usage Analysis: Global System Stress Test

The objective of this test is to stress the consistency model and the deduplication at the same time, and measure the reduction on bandwidth resources by using both techniques.

This test was composed by 10 file updates to 20 file samples. Each file sample had a total size of 10MB and all the file samples had always in common a section of the document, which produces a total of 33% of redundancy. Each file update consist in a total replacement of the content of the file.

To accomplish this, the VFCbox client that is downloading new updates, assigned different interests over different files.

Contrary to the test described in Section 5.2.8, in this test the VFCbox client's interests were assigned at a different granularity. Instead of selecting a chapter/section of a Latex document as an important object, in this test the VFCbox client selected the files that were considered as more important. As such, 5 files were selected as very important, 5 files as important and 10 files as not important.

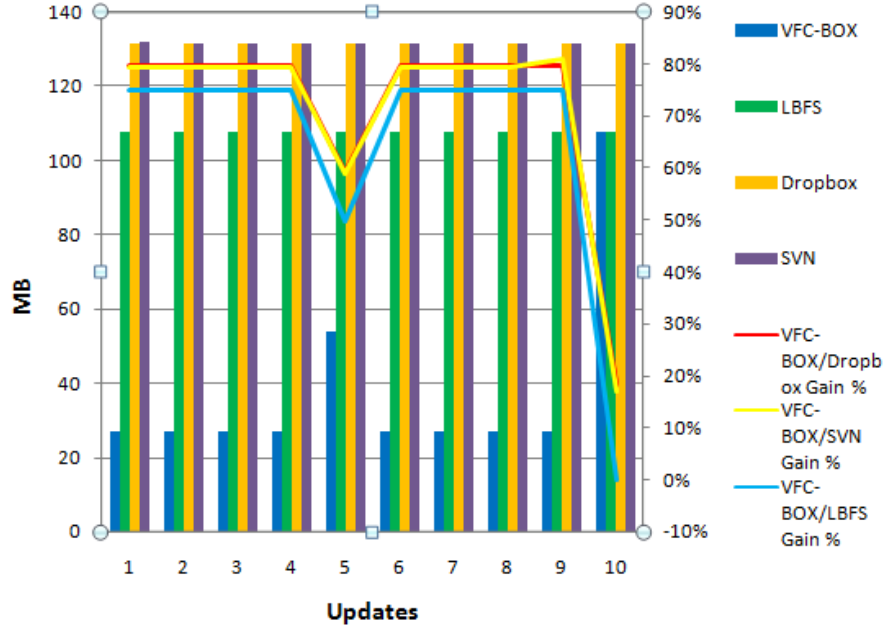


Figure 5.19: Bandwidth usage (in MB) of downloading 10 file updates of a set of 20 different file samples (10MB each file). The file updates were composed by a total replacement of the content of each file. These set of files have in common the same introduction, which produces a total of 33% of redundancy.

Figure 5.19 illustrates the results of the used bandwidth (in MB) to download each file update.

Given these results, we may determine that the Dropbox, the SVN and the LBFS system, make use of the same bandwidth to download each file update. This was an expected result since it means that for each update the whole content of the file was transmitted.

Analyzing the results of VFCbox, we may find that there are 3 distinct steps on the bandwidth usage results. These steps represent the multiple consistency levels of the VFC model.

The lowest step (1st to 4th update and 6th to 9th update) represents the downloading of the files considered as very important. Thus, a reduced amount of bandwidth is used.

The median step (5th update) represents the downloading of the files considered as very important and important. This is explained with the fact that on the 5th update the vfc sequence limit, of the important files region, is exceeded.

The biggest step (10th update) represents the downloading of all the files (very important, important and not important files). This is explained with the fact that on the 10th update the vfc sequence limits of all regions are exceeded. By this point, the client that is downloading the new files updates, downloads the whole files making them consistent with the other clients.

In this stress test we may observe that the Dropbox, the SVN and the LBFS use always the same amount of bandwidth to download each update. Nevertheless, the LBFS system still overcomes the Dropbox and the SVN system in terms of used bandwidth. This is explained due to the existence of 33% of redundancy

in each file. Since Dropbox and SVN use delta-encoding techniques, they are not able to detect this type of redundancy (intra-file redundancy) and thus have a lower performance in this test.

Analyzing the VFCbox gain percentage line, we may reinforce the above mentioned. As we may see, the lowest step of VFCbox bandwidth usage (1st to 4th update and 6th to 9th update) is the one that has the highest percentage of bandwidth reduction in comparison to the other systems, following by the median step and the highest step.

In comparison to Dropbox and SVN, VFCbox is capable of reducing the total amount of used bandwidth from 17% up to 80%.

In comparison to LBFS, VFCbox is capable of reducing the total amount of used bandwidth from 0% to 75%.

Figure 5.20 illustrates the sum of the used bandwidth of the above mentioned results (Figure 5.19).

In comparison to Dropbox and SVN, the VFCbox results show a total saving of 71% in bandwidth resources after 10 updates to 20 files.

In comparison to LBFS, the VFCbox results show a total saving of 65% in bandwidth resources after 10 updates to 20 files.

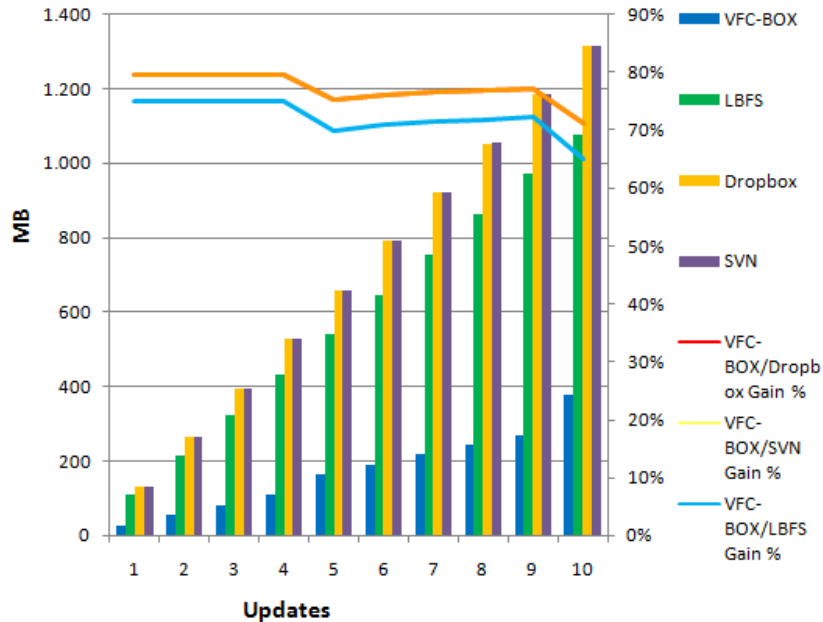


Figure 5.20: Bandwidth usage (in MB) sum of the stress test illustrated in Figure 5.19.

5.2.10 Bandwidth Usage Analysis: Real Workload Stress Test

The objective of this test is to stress the consistency model and the deduplication at the same time, and measure the reduction on bandwidth resources by using both techniques.

This test is similar to the previous test, however, in this one we opted to take a more realistic approach using thesis samples (described in Section 5.2.3) and requesting real users to perform some minor changes

to these file samples.

To accomplish this, each user shared his MSc thesis with a single VFCbox user that was downloading new updates. This user assigned a special interest over the *Architecture* chapter of each file sample. Then, each user performed small changes (insertion of 1 byte) on each chapter of the sharing file. This step was repeated 10 times.

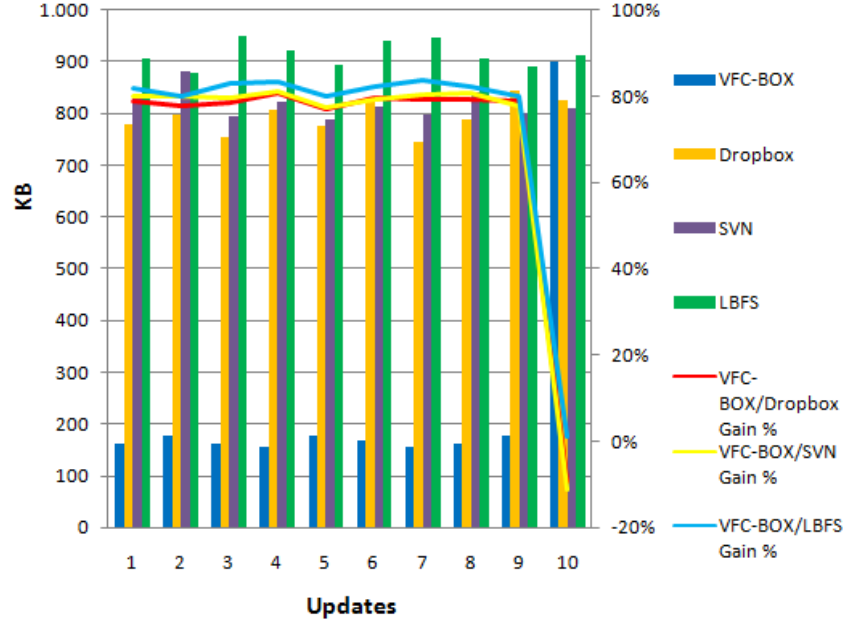


Figure 5.21: Bandwidth usage (in KB) of downloading 10 file updates of a set of 10 different thesis files. The file updates were composed by small changes on each chapter of each file.

Figure 5.21 illustrates the results of the used bandwidth (in KB) to download each file update.

By these results, we may determine that the Dropbox, the SVN and the LBFS systems, make use of more or less the same bandwidth to download each file update. This was an expected result since it means that for each update the multiple modifications to the content of the file were transmitted.

Analyzing the results of VFCbox, we may find that the same does not happen. We may see a huge difference between the 10th update and the others. This happens since the current VFCbox user has assigned a special interest over the Architecture chapter of each file. Thus, only this chapter and its sections are considered as important zones to the VFC model. Therefore, on each update only the modifications to this specific chapter are transferred, avoiding to transfer updates over the rest of the chapters. In the 10th update, the consistency limits (sequence limits) of all chapters/sections are exceeded and all modifications are transferred to the concerning client.

Analyzing the VFCbox gain percentage line, we may reinforce the above mentioned. As we may see, VFCbox has a high percentage gain in terms of bandwidth usage in comparison to the other solutions from the 1st to the 9th update.

In comparison to the other three solutions, VFCbox is capable of reducing the total amount of used bandwidth up to 82% on each update. The exception is in the 10th update, where all solutions propagate all

the modifications, and thus there are no bandwidth savings by VFCbox in comparison to the other systems.

Figure 5.22 illustrates the sum of the used bandwidth of the above mentioned results (Figure 5.21). In comparison to the three systems, the VFCbox results show a total saving from 70% to 74% in bandwidth resources after 10 updates to 10 files.

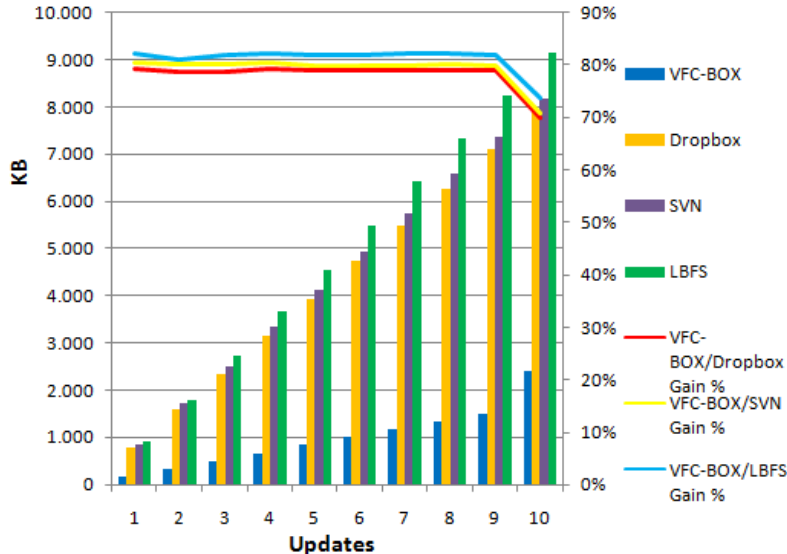


Figure 5.22: Bandwidth usage (in KB) sum of the stress test illustrated in Figure 5.21.

5.2.11 Summary

In this section we present and discuss the amount of data transferred per file download and file update.

In comparison to Dropbox, SVN and LBFS, VFCbox has almost the same performance in terms of capability to explore data redundancy.

Since Dropbox and SVN use delta-encoding techniques to explore cross-version redundancy, they are able to obtain better results in cases where small changes are performed to already existing files. VFCbox needs to transfer more meta-data (chunks hashes) than these systems, thus it spends more bandwidth to propagate the modifications. Nevertheless, we may refer that in absolute terms this overhead is not so substantial, since it only occurs for small changes and small amounts of exchanged data. W.r.t. bigger modifications, this VFCbox overhead becomes more dissipated. Additionally, VFCbox is capable of detecting cross-file redundancy and intra-file redundancy (redundancy between parts of the same file). Dropbox and SVN are only capable to explore cross-version redundancy, thus VFCbox overcomes these two systems when in presence of large amounts of these types of redundancy.

Comparing to LBFS, the results are very similar, which was already expected. Since both systems use compare-by-hash techniques, both incur in extra overheads to transfer the extra meta-data. However, and as already mentioned, they are capable to explore different types of redundancy which brings huge benefits when in presence of large amounts of data redundancy.

Summarizing, VFCbox may be seen as a system with equivalent deduplication techniques w.r.t. the compared solutions.

In the overall and in comparison to Dropbox, SVN and LBFS, VFCbox overcomes the capability of reducing bandwidth resources. Through an efficient and usable consistency model, VFCbox is capable to heavily reduce the use of network resources.

After the analysis of the obtained values, the following conclusions can be made:

- **Deduplication based on hash values comparison can greatly reduce the amount of redundant data between sets of files in comparison to delta-encoding techniques.** Although delta-encoding may have a higher performance when making small changes to a single file, it is not able to detect other types of redundancy. Our analysis shows that the benefits that are taken from the reduction of redundant data between sets of files is much gainful in comparison to the overhead that the deduplication based on hash values has when deduplicating 2 versions of the same file.
- **VFCbox is capable of reducing up to 80% of the used bandwidth in comparison to the compared solutions by using a consistency model based on user's interests.** Using the VFC model, our prototype is capable of postponing updates, which eventually are replaced by newer updates. Thus, it prevents the transfer of updates that are explicitly marked by users as updates of low interest. As such, users do not really suffer with delayed updates, since they have already marked those updates as non interesting. Further, it can also be an advantage to users due to the fact that they can receive important data in advance, reducing the latency seen by users.

Chapter 6

Conclusion

The need for file sharing systems motivated the design and implementation of systems capable of efficiently sharing and storing files. Furthermore, the interest of collaborators over parts of the shared information improve the ability of systems to perform data synchronization while reducing the use of network resources and shrinking the latency of data seen by users. This document presented a file sharing system capable of performing a more efficient synchronization: VFCbox.

The system achieves a better transfer efficiency through the use of deduplication techniques and the use of the VFC model. Through deduplication, the VFCbox is able of detecting redundant data between multiple versions of the same file or even between multiple files. With this redundant data detection, network bandwidth may be saved since the redundant data has no longer to be transferred. This system is also capable of performing a selective scheduling of updates based on its importance, determined through user specification. Therefore, multiple consistency levels are applied over multiple data sets (files or file parts), which gives the system the ability of postponing certain updates. These postponed updates end to be overlapped by new updates, avoiding the need of synchronizing them which saves network resources.

A prototype was built for running in Windows platforms. The prototype consists in a multi-user client-server application that allows to perform synchronization operations. Additionally, the client side application is equipped with an interface in which users may specify their interests over the shared files or even over parts of the shared files.

The VFCbox prototype was evaluated using several samples with different sizes and contents. The results of the evaluation were then compared with three different solution, which are representative of the state-of-the-art techniques used by most file sharing systems (deduplication execution and total consistency enforcement). From this evaluation, we concluded that VFCbox achieves equivalent results regarding the deduplication execution. Nevertheless, VFCbox provides an additional consistency model in comparison to the total consistency model: the VFC model. This model proved to bring huge benefits w.r.t. the total used network bandwidth. In our evaluation, and only with a small experiment we proved that using the VFC model against the total consistency model, the system could save up to 80% of the used bandwidth. Therefore, the system proved to use very low bandwidth compared to other solutions, which makes it a suitable file sharing system.

In conclusion, VFCbox is an efficient file sharing system that uses low bandwidth to synchronize file updates by using several techniques such as data deduplication, data compression and a relaxed consistency model that makes an intelligent schedule of updates according to its importance to each user.

6.1 Future Work

In this section, we present modifications or improvements that may be done in the future:

- **Expand the concept of interest management to the data uploading perspective.** The current management of user's interest is only used to perform a selective scheduling of updates from the downloading perspective. I.e., user's interests are only related with the consistency level that a user wants to guarantee over certain incoming updates. Nevertheless, this concept could be expanded to outgoing updates. By this, users could decrease the upload frequency of files that are being frequently updated and have no interest of uploading the file at each modification. Further, this could also be used in order to specify an uploading order. Users could then specify for example that a certain file should be uploaded in advance of others. Therefore, users could control the uploading rate of each file or file zone decreasing the latency of data considered as more important.
- **Specify special interests over certain file types.** Instead of specifying user's interests for each file, the system could enable the possibility of specifying interests over file types. For instance, a certain user could want to specify a special interest over all the Latex files. Another user could want to specify that he has no interest of receiving updates of temporary files.
- **Automatic extraction of user's interests.** The system could have an automatic mechanism to extract user's interests without having to ask the user to specify them. For instance, higher frequently opened files could have higher consistency guarantees applied and lower frequently opened files could have lower consistency guarantees applied.

Bibliography

- [1] M. Armbrust, A. Fox, R. Griffith, A. Joseph, and R. Katz. A view of cloud computing. *In Magazine Communications of the ACM*, Volume 53 Issue 4:50–58, 2010.
- [2] R. Balan, M. Ebling, P. Castro, and A. Misra. Matrix: Adaptive middleware for distributed multiplayer games. *In Middleware '05: ACM/IFIP/USENIX 6th International Middleware Conference*, Volume 3790:390–400, 2005.
- [3] S. Balasubramaniam and B. Pierce. What is a file synchronizer. *In MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, 1998.
- [4] J. Barreto. Optimistic replication in weakly connected resource-constrained environments. *PhD Thesis, Instituto Superior Tecnico*, 2008.
- [5] J. Barreto and P. Ferreira. A replicated file system for resource constrained mobile devices. *In Proceedings of IADIS International Conference on Applied Computing*, 2004.
- [6] J. Barreto and P. Ferreira. A highly available replicated file system for resource-constrained windows ce .net devices. *In 3rd International Conference on .NET Technologies*, 2005.
- [7] J. Barreto and P. Ferreira. Efficient locally trackable deduplication in replicated systems. *In Middleware'09: Proceedings of the ACM/IFIP/USENIX 10th international conference on Middleware*, 2009.
- [8] W. Bolosky, S. Corbin, D. Goebel, and J. R. Douceur. Single instance storage in windows 2000. *In WSS'00 Proceedings of the 4th conference on USENIX Windows Systems Symposium*, Volume 4, 2000.
- [9] M. Cart and J. Ferrie. Asynchronous reconciliation based on operational transformation for p2p collaborative environments. *In COLCOM '07: Proceedings of the 2007 International Conference on Collaborative Computing: Networking, Applications and Worksharing*, pages 127–138, 2007.
- [10] P. Cederqvist and et al. Version management with cvs. <http://www.cvshome.org/docs/manual/>. 1993.
- [11] J. Costa, P Ferreira, and L.. Veiga. Vector-field consistency for cooperative work. *Msc Thesis, Instituto Superior Tecnico*, 2010.
- [12] L. Cox, C. Murray, and B. Noble. Pastiche: Making backup cheap and easy. *In OSDI '02: Proceedings of the 5th symposium on Operating systems design and implementation*, pages 285–298, 2002.
- [13] F. Douglass and A. Iyengar. Application-specific delta encoding via resemblance detection. *In Proceedings of the 2003 USENIX Annual Technical Conference*, pages 113–126, 2003.
- [14] D.E. Eastlake and P.E. Jones. Us secure hash algorithm 1 (sha1). <http://www.ietf.org/rfc/rfc3174.txt?number=3174>, 2001.
- [15] Collins-Sussman et al. Version control with subversion. *O'Reilly*, 2004.
- [16] K. Morse et al. Interest management in large-scale distributed simulations. *Information and Computer Science, University of California, Irvine*, 1996.

- [17] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. In *Journal ACM Transactions on Programming Languages and Systems (TOPLAS)*, Volume 11 Issue 3:345–387, 1989.
- [18] J. J. Hunt, K.-P. Vo, and W. F. Tichy. An empirical study of delta algorithms. In *ICSE '96: Proceedings of the SCM-6 Workshop on System Configuration Management*, pages 49–66, 1996.
- [19] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Magazine Communications of the ACM*, Volume 21 Issue 7, 1978.
- [20] Y. Lu, Y. Lu, and H. Jiang. Adaptive consistency guarantees for large-scale replicated services. In *NAS '08: Proceedings of the 2008 International Conference on Networking, Architecture, and Storage*, 2008.
- [21] N. Mandagere, P. Zhou, M. Smith, and S. Uttamchandani. Demystifying data deduplication. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, 2008.
- [22] F. Mattern. Virtual time and global states of distributed systems. In *Parallel and Distributed Algorithms: proceedings of the International Workshop on Parallel and Distributed Algorithms*, 1989.
- [23] A. Muthitacharoen, B. Chen, and D. Mazieres. A low-bandwidth network file system. In *SOSP '01: Proceedings of the eighteenth ACM symposium on Operating systems principles*, Volume 35 Issue 4:174–187, 2001.
- [24] J. Paiva and P. Ferreira. Backupchunk: A chunk-based backup system. *Msc Thesis, Instituto Superior Tecnico*, 2009.
- [25] M. Palankar, A. Iamnitchi, M. Ripeanu, and S. Garfinkel. Amazon s3 for science grids: a viable solution? In *DADC '08: Proceedings of the 2008 international workshop on Data-aware distributed computing*, 2008.
- [26] K. Petersen, M. Spreitzer, D. Terry, and M. Theimer. The bayou architecture: Support for data sharing among mobile users. In *WMCSA '94: Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, pages 2–7, 1994.
- [27] B. Pierce and J. Vouillon. Whats in unison? a formal specification and reference implementation of a file synchronizer. *Technical Report MS-CIS-03-36, Department of Computer and Information Science University of Pennsylvania*, 2004.
- [28] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *First USENIX conference on File and Storage Technologies, Monterey, CA*, 2002.
- [29] M. Rabin. Fingerprinting by random polynomials. *Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University*, 1981.
- [30] D. Ratner. Roam: A scalable replication system for mobile and distributed computing. *PhD Thesis 970044, University of California*, 1998.
- [31] R.L. Rivest. The md5 message-digest algorithm (rfc 1321). <http://www.ietf.org/rfc/rfc1321.txt?number=1321>, 1992.
- [32] Y. Saito and M. Shapiro. Optimistic replication. In *Journal ACM Computing Surveys (CSUR)*, Volume 37 Issue 1(1):42–81, 2005.
- [33] N. Santos, L. Veiga, and P. Ferreira. Vector-field consistency for ad-hoc gaming. *INESC-ID/Technical University of Lisbon, Distributed Systems Group Rua Alves Redol N 9, 1000-029 Lisboa*.
- [34] M. Satyanarayanan. The evolution of coda. In *Journal ACM Transactions on Computer Systems (TOCS)*, Volume 20 Issue 2:85–124, 2002.

- [35] H. Seunghyun, L. Mingyu, and Dongman L. Scalable interest management using interest group based filtering for large networked virtual environments. *In VRST '00: Proceedings of the ACM symposium on Virtual reality software and technology*, 2000.
- [36] A. Shiferaw, V. Scuturici, and L. Brunie. Interest-awareness for information sharing in manets. *In MDM '10: Proceedings of the 2010 Eleventh International Conference on Mobile Data Management*, 2010.
- [37] W. F. Tichy. Rcs - a system for version control. *Department of Computer Sciences Purdue University West Lafayette, Indiana 47907*, 1991.
- [38] A. Tridgell and P. Mackerras. The rsync algorithm. *Australian National University*, 1998.
- [39] P. Vala and P. Ferreira. Shiftback: Efficient and time-shifting backup. *Msc Thesis, Instituto Superior Tecnico*, 2010.
- [40] L. Veiga and P. Ferreira. Semantic-chunks: A middleware for ubiquitous cooperative work. *In ARM '05 Proceedings of the 4th workshop on Reflective and Adaptive Middleware Systems*, 2005.
- [41] L. Veiga, A. Negrao, N. Santos, and P. Ferreira. Unifying divergence bounding and locality awareness in replicated systems with vector-field consistency. *INESC-ID/Technical University of Lisbon, Distributed Systems Group Rua Alves Redol N 9, 1000-029 Lisboa*, 2010.
- [42] W. Vogels. Eventually consistent. *In Magazine Communications of the ACM Communications of the ACM - Rural engineering development CACM*, Volume 52 Issue 1:40–44, 2009.
- [43] A.-I. Wang, P. L. Reiher, and R. Bagrodia. Understanding the conflict rate metric for peer optimistically replicated filing environments. *In DEXA '02: Proceedings of the 13th International Workshop on Database and Expert Systems Applications*, 2002.
- [44] M. Weiser. The computer for the twenty-first century. *In ACM SIGMOBILE Mobile Computing and Communications Review*, Volume 3 Issue 3, 1991.
- [45] H. Yu and A. Vahdat. Design and evaluation of a conflict-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems (TOCS)*, Volume 20 Issue 3:239–282, 2002.
- [46] H. Yu and A. Vahdat. The costs and limits of availability for replicated services. *In Journal ACM Transactions on Computer Systems (TOCS)*, Volume 24 Issue 1:70–113, 2006.

Appendix







	1.tex	2.tex	3.tex	4.tex	5.tex
 Dropbox	686	3.381	6.746	33.750	67.397
 SVN	708	3.316	6.646	34.886	68.012
 LBFS	898	3.512	7.011	34.876	68.102
 VFC-BOX	874	3.596	7.033	34.321	68.425
 VFC-BOX/Dropbox Gain	-27%	-6%	-4%	-2%	-2%
 VFC-BOX/SVN Gain	-23%	-8%	-6%	2%	-1%
 VFC-BOX/LBFS Gain	3%	-2%	0%	2%	0%

Figure 6.1: Table results of the evaluation test illustrated in Figure 5.5.








	1.tex	2.tex	3.tex	4.tex	5.tex
 Dropbox	14	19	22	58	105
 VFC-BOX	16	24	25	40	61
 SVN	14	20	22	57	109
 LBFS	17	25	25	42	62
 VFC-BOX/Dropbox Gain	-11%	-32%	-14%	31%	42%
 VFC-BOX/SVN Gain	-15%	-22%	-14%	30%	44%
 VFC-BOX/LBFS Gain	6%	2%	0%	5%	2%

Figure 6.2: Table results of the evaluation test illustrated in Figure 5.6.








	1.tex	2.tex	3.tex	4.tex	5.tex
 Dropbox	15	16	19	40	66
 VFC-BOX	16	23	26	38	59
 SVN	14	16	18	41	68
 LBFS	16	23	27	38	60
 VFC-BOX/Dropbox Gain	-6%	-44%	-37%	6%	12%
 VFC-BOX/SVN Gain	-14%	-43%	-42%	8%	14%
 VFC-BOX/LBFS Gain	0%	1%	5%	1%	2%

Figure 6.3: Table results of the evaluation test illustrated in Figure 5.7.








	1.tex	2.tex	3.tex	4.tex	5.tex
 Dropbox	11	11	14	12	24
 VFC-BOX	16	20	17	38	59
 SVN	12	11	15	13	26
 LBFS	17	20	17	39	60
 VFC-BOX/Dropbox Gain	-45%	-79%	-25%	-212%	-149%
 VFC-BOX/SVN Gain	-34%	-80%	-17%	-189%	-126%
 VFC-BOX/LBFS Gain	6%	1%	-3%	4%	2%

Figure 6.4: Table results of the evaluation test illustrated in Figure 5.8.

	5.tex	4.tex	3.tex	2.tex	1.tex
90%	67.370	33.682	6.749	3.380	680
80%	67.370	33.680	6.765	3.447	669
70%	67.348	33.713	6.747	3.433	677
60%	67.628	33.676	6.752	3.395	675
50%	67.436	33.703	6.759	3.381	675
40%	67.432	33.677	6.746	3.382	671
30%	67.447	33.705	6.756	3.382	670
20%	67.349	33.680	6.761	3.381	648
10%	67.342	33.813	6.747	3.393	648

Figure 6.5: Table results of the evaluation test illustrated in Figure 5.9.

	5.tex	4.tex	3.tex	2.tex	1.tex
90%	12.166	6.709	2.435	1.761	874
80%	18.457	9.806	2.884	1.812	874
70%	24.782	12.925	3.416	2.196	874
60%	31.097	16.036	3.946	2.416	874
50%	37.481	19.168	4.471	2.595	874
40%	43.854	22.241	4.995	2.812	874
30%	50.259	25.341	5.522	3.036	873
20%	56.432	28.472	6.046	3.214	874
10%	62.750	31.528	6.556	3.411	874

Figure 6.6: Table results of the evaluation test illustrated in Figure 5.10.

	5.tex	4.tex	3.tex	2.tex	1.tex
90%	67.538	34.013	6.777	3.403	653
80%	67.538	34.013	6.777	3.403	653
70%	67.538	34.013	6.777	3.403	653
60%	67.538	34.013	6.777	3.403	653
50%	67.538	34.013	6.777	3.403	653
40%	67.538	34.013	6.777	3.403	653
30%	67.538	34.013	6.777	3.403	653
20%	67.538	34.013	6.777	3.403	653
10%	67.538	34.013	6.777	3.403	653

Figure 6.7: Table results of the evaluation test illustrated in Figure 5.11.

	5.tex	4.tex	3.tex	2.tex	1.tex
90%	20.899	8.701	2.015	1.566	684
80%	27.532	10.685	3.080	2.004	698
70%	32.194	15.835	3.826	2.091	708
60%	35.275	17.500	4.103	2.221	723
50%	38.467	19.552	4.793	2.731	734
40%	43.945	25.132	5.566	2.707	763
30%	51.722	26.272	5.687	2.929	780
20%	57.698	29.320	6.505	3.234	783
10%	63.579	31.518	6.748	3.431	824

Figure 6.8: Table results of the evaluation test illustrated in Figure 5.12.

	5.tex	4.tex	3.tex	2.tex	1.tex
1st	67.715	33.774	6.748	3.383	686
2nd	67.715	33.774	6.748	3.383	686
3rd	67.715	33.774	6.748	3.383	686
4th	67.715	33.774	6.748	3.383	686
5th	67.715	33.774	6.748	3.383	686
6th	67.715	33.774	6.748	3.383	686
7th	67.715	33.774	6.748	3.383	686
8th	67.715	33.774	6.748	3.383	686
9th	67.715	33.774	6.748	3.383	686
10th	67.715	33.774	6.748	3.383	686

Figure 6.9: Table results of the evaluation test illustrated in Figure 5.14.

	5.tex	4.tex	3.tex	2.tex	1.tex
1st	2.054	348	217	114	33
2nd	2.049	348	217	115	33
3rd	2.055	348	218	114	40
4th	2.053	348	217	115	35
5th	17.986	8.200	1.684	757	213
6th	2.054	348	217	115	33
7th	2.053	347	217	114	33
8th	1.073	348	217	115	33
9th	2.051	348	217	115	33
10th	68.425	34.319	7.033	3.518	870

Figure 6.10: Table results of the evaluation test illustrated in Figure 5.15.

	5.tex	4.tex	3.tex	2.tex	1.tex
1st	68.712	33.881	6.828	3.411	689
2nd	68.712	33.881	6.828	3.411	689
3rd	68.712	33.881	6.828	3.411	689
4th	68.712	33.881	6.828	3.411	689
5th	68.712	33.881	6.828	3.411	689
6th	68.712	33.881	6.828	3.411	689
7th	68.712	33.881	6.828	3.411	689
8th	68.712	33.881	6.828	3.411	689
9th	68.712	33.881	6.828	3.411	689
10th	68.712	33.881	6.828	3.411	689

Figure 6.11: Table results of the evaluation test illustrated in Figure 5.16.

	5.tex	4.tex	3.tex	2.tex	1.tex
1st	68.513	35.100	7.062	3.616	881
2nd	68.513	35.100	7.062	3.616	881
3rd	68.513	35.100	7.062	3.616	881
4th	68.513	35.100	7.062	3.616	881
5th	68.513	35.100	7.062	3.616	881
6th	68.513	35.100	7.062	3.616	881
7th	68.513	35.100	7.062	3.616	881
8th	68.513	35.100	7.062	3.616	881
9th	68.513	35.100	7.062	3.616	881
10th	68.513	35.100	7.062	3.616	881

Figure 6.12: Table results of the evaluation test illustrated in Figure 5.17.

	1	2	3	4	5	6	7	8	9	10
VFC-BOX	27	27	27	27	54	27	27	27	27	108
LBFS	108	108	108	108	108	108	108	108	108	108
Dropbox	132	132	132	132	132	132	132	132	132	132
SVN	132	132	132	132	132	132	132	132	132	132
VFC-BOX/Dropbox Gain %	80%	80%	80%	79%	59%	80%	80%	80%	80%	18%
VFC-BOX/SVN Gain %	80%	80%	80%	79%	59%	80%	80%	80%	81%	17%
VFC-BOX/LBFS Gain %	75%	75%	75%	75%	50%	75%	75%	75%	75%	0%

Figure 6.13: Table results of the evaluation test illustrated in Figure 5.19.

	1	2	3	4	5	6	7	8	9	10
VFC-BOX	27	54	81	108	162	189	216	243	270	378
LBFS	108	216	323	431	539	647	755	863	970	1.078
Dropbox	132	264	395	527	659	791	922	1.054	1.186	1.318
SVN	132	264	396	527	659	791	923	1.054	1.186	1.318
VFC-BOX/Dropbox Gain %	80%	80%	80%	80%	75%	76%	77%	77%	77%	71%
VFC-BOX/SVN Gain %	80%	80%	80%	80%	75%	76%	77%	77%	77%	71%
VFC-BOX/LBFS Gain %	75%	75%	75%	75%	70%	71%	71%	72%	72%	65%

Figure 6.14: Table results of the evaluation test illustrated in Figure 5.20.

	1	2	3	4	5	6	7	8	9	10
VFC-BOX	163	176	160	154	178	168	154	162	178	902
Dropbox	780	798	756	806	777	828	745	788	846	825
SVN	828	882	796	824	789	815	798	843	802	812
LBFS	906	879	951	921	893	942	948	906	890	912
VFC-BOX/Dropbox Gain %	79%	78%	79%	81%	77%	80%	79%	79%	79%	-9%
VFC-BOX/SVN Gain %	80%	80%	80%	81%	77%	79%	81%	81%	78%	-11%
VFC-BOX/LBFS Gain %	82%	80%	83%	83%	80%	82%	84%	82%	80%	1%

Figure 6.15: Table results of the evaluation test illustrated in Figure 5.21.

	1	2	3	4	5	6	7	8	9	10
VFC-BOX	163	339	499	653	831	999	1.153	1.315	1.493	2.395
Dropbox	780	1.578	2.334	3.140	3.917	4.745	5.490	6.278	7.124	7.949
SVN	828	1.710	2.506	3.330	4.119	4.934	5.732	6.575	7.377	8.189
LBFS	906	1.785	2.736	3.657	4.550	5.492	6.440	7.346	8.236	9.148
VFC-BOX/Dropbox Gain %	79%	79%	79%	79%	79%	79%	79%	79%	79%	70%
VFC-BOX/SVN Gain %	80%	80%	80%	80%	80%	80%	80%	80%	80%	71%
VFC-BOX/LBFS Gain %	82%	81%	82%	82%	82%	82%	82%	82%	82%	74%

Figure 6.16: Table results of the evaluation test illustrated in Figure 5.22.