UNIVERSITAT ROVIRA I VIRGILI

Department of Computer Engineering and Mathematics

Architecture and Telematic Services Research Group

M.Sc. Thesis Dissertation

Developing a Bundling Mechanism to Make Peer-Assisted File Synchronization Feasible.

by

Raúl Sáiz Laudó

Thesis Advisor:

Dr. Marc Sánchez Artigas

Dissertation submitted to the Department of Computer Engineering and Mathematics in partial fulfilment of the requirements of the degree of

Master in Computer Engineering and Security

September 2014

Abstract

As tools for personal storage, file synchronization and data sharing, cloud storage services have quickly gained popularity. This services provide users a reliable data storage that can be automatically synced across multiple devices, and also shared among a group of users.To minimize the network overhead, cloud storage services employ binary diff on file chunks, file bundling, data compression, and other mechanisms when transferring updates among users.

The main problem is that these services mainly rely on the client-server communication paradigm to make their content available, missing the opportunity to benefit from the interest of users in the same content. Recent research papers have proved that a high percentage of file sizes are under $1MB$ [10], we propose different scenarios where may be feasible to use clients bandwidths to offload cloud servers applying bundling on this small files. In this regard, the content provider could cut bandwidth costs by making use of peer-to-peer content delivery by benefiting from the collaborative sharing of their upload capacity in terms of download speed.

As main goal in this thesis we will study the relation between peer-to-peer content delivery and bundling mechanisms in order to minimize the amount of data to be sent to cloud servers and relax its workload as well as bandwidth use.

Acknowledgments

This thesis is the result of stage at the Architecture and Telematic Services (AST) research group in the Universitat Rovira i Virigi, Tarragona.

First of all, I would like to thank my advisor Dr. Marc Sánchez Artigas for his support and guidance during the development of this thesis. I would also like to thank him for giving me the freedom that I had regarding to many aspects of this work contributing to my personal development.

I also want to thank all the people that have worked with me at the AST group during this period.

Last but not least, a warm thank you to my family for their moral support. Thank you very much for always being there for me.

Thank you very much!

Raúl.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Personal cloud storage services like Dropbox, Google Drive, or SkyDrive have become mainstream today. These services deliver an unprecedented amount of data in a daily basis to their users through the use of data centers to store, process and deliver such content as if their capacity were unlimited.

Under the umbrella of the FP7-ICT project CloudSpaces (http://cloudspaces.eu/), some authors already integrated BitTorrent [8], a well-known peer-to-peer protocol that leverages spare upstream capacity of users, to offload storage servers from doing all the serving [4]. Using BitTorrent, the authors of [4] have achieved excellent results for moderate to large files.

Unfortunately, one important challenge remains, which is understanding whether it is profitable to roll out a peer-to-peer content delivery mechanism to reconcile out-of-sync devices. While it is clear that this optimization is beneficial when many people simultaneously download a file, the typical scenario is, however, a user syncing files and folders across multiple computers and devices so that the same files exists everywhere. This usage pattern is very different from that found in file sharing applications and consequently, it is fundamental to determine when peer-to-peer content delivery is suitable. According to a recent measurement of Dropbox [10], 80% of storage flows consist of less than 1 MB. Such a little amount of data makes it unwise to use peer-to-peer content delivery at all times, since the associated overheads may outweigh any savings in server bandwidth as proven in [4]. This paper accepted on "P2P 2014" proves with certain formulas aid when it is useful to change HTTP by BitTorrent to do peer-to-peer content delivery.

One way to increase the amount of data is to batch together several file mutations, what is known as file bundling in the literature. For instance, Dropbox uses file bundling to transfer a batch of files, so that files could be pipelined, and this way, reduce both transmission latency and control overhead impact.

At first glance, file bundling should increase the amount of data to be transferred, making it more suitable the use of peer-to-peer technology to reduce bandwidth consumption at the server side. Understanding the interplay between file bundling and peer-to-peer content delivery is the main objective of this thesis.

1.1 Objectives of Thesis

- Determine to what extent file bundling in conjunction with BitTorrent can cut down bandwidth costs in typical file syncing scenarios. This issue is not trivial, as it depends in the way files and folders evolve over time by creating, copying, or mutating files, as well as the chunking algorithms used to detect those file mutations.
- Understand how bundling works in combination with other basic mechanisms like content-based chunking, techniques that are commonplace in commercial personal cloud services like Dropbox, Google Drive and SkyDrive, to name a few.

1.2 Contributions of this Thesis

Currently, fixed-size chunking is currently used by Dropbox and Google Drive, so the results of this thesis have the potential to reach out to both industry and researchers to quantify the effects of future design choices.

For instance, in typical collaborative editing scenario as that found in Wikipedia, file bundling in conjunction with BitTorrent can achieve up to 57% savings in cloud bandwidth.

This thesis also can be applied to support the design of realistic synthetic workloads helping to develop and evaluate new personal cloud storage services.

1.3 Thesis structure

After this brief introduction, Chapter 2 introduces the related work and background needed to understand concepts and definitions that we will use throughout this thesis. Among others, it discusses existing Cloud systems, analyzing different techniques and mechanisms regarding data chunking and data bundling.

In Chapter 3 is described the problem statement and the experiment types that are elected to characterize the expected world in which experiments must be executed.

In Chapter 4 is described the environment used to perform our experiments as well as key parameters like the window size to synchronize user data. In this context, it is performed a feasibility study of `Cloud systems behavior` and evaluated the key aspects such as data redundancy or the server offload to take advantage of the Cloud. Also, the main results on each experiment are presented and explained deeper.

To conclude this thesis in Chapter 5 we will draw the conclusions and enumerate the achieved goals through our study.

Chapter 2

Background and Related Work

2.1 Distributed Storage

Distributed storage systems are composed by several storage resources from different computers or dedicated devices put together to form a large storage service. Unlike local storage solutions, distributed storage services provide a more reliable, scalable and efficient solution.

Due to their decentralized nature, responsibility does not fall against a single entity, instead it is split among different storage nodes. A storage node is a network element that poses one or more physical storage devices to provide a simple storage service. That includes elements such as desktop and laptop computers, NAS devices or storage components from data centers.

Therefore, as each node is less important, it can implement simpler and cheaper hardware rather than complex and more expensive hardware needed in critical storage nodes. Errors such as hardware failures or a power outage is assumed to be normal. So if a node becomes unavailable, the software layer takes into account and, hence, has to guarantee this failure do not interfere the correct functioning of the system.

As nodes are distributed, network speed and latency become a more considerable concern than simply disk access speed, and it is essential to keep proper values in order to achieve a right performance of the system. This can be accomplished by reducing the number of hops between nodes, improving network switch speed, implementing caching systems, etc.

Besides, the system has to be keep track of multiples data stored across the nodes and ensure its availability. When a node is unavailable, the availability of the data that this node was storing decreases, and hence, the distributed storage system has to reassign that data into other nodes so that it increases its availability to levels considered correct. This maintenance task has to be performed repeatedly in order to keep a consistent system before any stored object is irreversibly lost.

Even though distributed storage systems are transparent for the end-user, they hold an immense responsibility for the society. Nowadays, these systems are used daily in a wide variety of services such as Facebook, Gmail or Flickr, to name a few.

However, due to its decentralized nature, distributed storage systems inherit some drawbacks that must be taken into account:

- ***Node failures.*** A node can become unavailable due to a variety of reasons —e.g., a power outage or a hardware error— and must be ready to overcome the issue and continue its normal functioning. Therefore, the system must ensure that the data stored on the failed node is replicated to other nodes by introducing data redundancy. Hence, the same rate of data availability is kept.

- ***Data placement strategy.*** Since nodes are heterogeneous, the system needs to distribute data among them intending to maintain the system well balanced and avoiding bottlenecks.

- ***Bandwidth limitations.*** Nodes may have different bandwidth limitations. Hence, the system must be designed taking into account this constraint.

- ***Parallel access.*** Distributed storage systems need to bear in mind that concurrent access to a data object may be possible. Thus, a mechanism to allow simultaneous read and write operations must be implemented.

2.1.1 Cloud storage

In Cloud storage systems data is stored in virtualized pools of storage hosted in large data centers. Data stored on Cloud storage systems is made available as a service through a web service application programming interface (API), a cloud storage gateway or through a Web-based user interface.

The Cloud storage definition is so broad that, in order to clarify it, a group of industry analyst dealing with IT professionals defined a set of requirements a system has to meet to be considered a Cloud storage system.

- ***Massively scalable***. The system has to handle growing amount of data in a capable manner or be able to be enlarged to accommodate that growth. In the same sense, if new storage nodes are added to the system, its performance improves proportionally to the capacity added.

- ***Geographic independent***. The service provided is not tied to an specific geographic location. Instead, it must allow access from any part of the glove.

- ***Billed on a usage basis***. Pricing is proportionally dependent on the resources consumed, i.e. the costs this resources imply for the provider. This fine-grained pricing model allows lowering the barriers to entry as the service does not have to purchase one-time fixed resources.

- ***Application agnostic***. The service has to be accessible to software and enables it to interact with cloud software in the same way the user interface facilitates interaction between humans and computers.

Architecture models

Cloud storage, can be roughly differentiated into three models. Each model corresponds to a layer that abstracts from the details of lower layers.

- ***Infrastructure as a service (IaaS).*** It is the most basic model and provides raw or file-based storage as a service. It aggregates many resources by virtualization in order to increase or decrease these resources on demand.

- ***Platform as a service (PaaS).*** In this model, an specific environment is provided to developers where they can build their own applications. Unlike IaaS solutions, PaaS providers may apply restrictions in terms like programming languages or storage capacity.

- ***Software as a service (SaaS).*** It is the highest abstraction layer. This term is used to describe applications that are managed and hosted by a third party and whose interface is accessed from the client side.

Major providers

Although Cloud storage services demands are growing through the IT world, many enterprises are continuously being created to provide these services. Following we briefly analyze what we considered to be the most important ones:

- **Amazon S3** [25] (Simple Storage Service) is part of the Amazon Web Services. The design has not made public, but according to Amazon it provides scalability, high availability and low latency. Amazon has data centers through the entire globe, locations such as North America, Europe or East Asia. They standard and reduced redundancy. They differ in the number of times a data object is replicated. Amazon claims that the standard redundancy option provides 99.999999999% durability and 99.99% availability. Amazon S3 pricing depends on a variety of aspects: redundancy level, storage used, number of request (GET, PUT, etc.), data transferred and the geographic region. As an example, storing 500 GB of data, performing 5.000 PUT and 50.000 GET requests, transferring 2 TB per month would cost us around US$ 300 using standard redundancy.

- **Google Cloud Storage** [26] is a RESTful service for storing and accessing data on Google's infrastructure. This service claims to combine the performance and scalability of Google's cloud with advanced security and sharing capabilities. Most Google's data centers are located in the USA, but also they have two in Europe and three more in East Asia. Google pricing is similar to Amazon S3. It is based on storage and bandwidth usage, number and type of requests and the geographic region.

- **Windows Azure** [3]. Microsoft has also developed a cloud computing platform. This platform consists of various on-demand services hosted in Microsoft data centers, among these services we will focus on Windows Azure. Microsoft's data centers are located along the USA, Europe and Asia. With CDN nodes located in 24 countries. Microsoft claims that data stored in their system is replicated three times in the same data center and replicated between two data centers on the same continent. Pricing is similar to the previously mentioned providers, with pay-as-you-go fares and averaging $0.125 per GB stored per month, 10,000 request at the price of $0.01.

2.1.2 Personal Cloud Storage

A Brief History of the Personal Cloud Term

In the past years there have been divergent views of the "Personal Cloud" concept. For example, in [17] authors propose an architecture and design for accessing and sharing computational resources in virtual machines. For them, a Personal Cloud is a collection of Virtual Machines running on unused computers at the edge. Another different view [2] focuses on collaborative work , where a web infrastructure is defined to provide a unified environment for handling activities and collaborations. Finally, a recent trend [28] goes further and defines the Personal Cloud as a cloud Operating System that offers a core set of services around identity, trust, data access and even programming models. In this paper, we focus on Personal Cloud Storage platforms that take care of data sync and sharing from heterogeneous devices. In fact, the term "Personal Cloud" have received a lot of attention with the recent research reports from Forrester [15] and Gartner [14]. Like us, these reports associate the term Personal Cloud with online cloud storage services such as Dropbox, or Google Drive among others.

The Personal Cloud is a unified digital locker for our personal data offering three key services: Storage, Synchronization and Sharing. On the one hand, it must provide redundant and trustworthy cloud data storage for our information flows irrespective of their type. On the other hand, it must provide syncing and file exploring capabilities in different heterogeneous platforms and devices. And finally, it must offer fine-grained information sharing to third-parties (users and applications).

Architectural complexity is well captured by file synchronization, as the success of Personal Cloud services lies in great measure in the scalability of their sync services. In this cloud paradigm, a desktop client software typically keeps all the files in a specified folder in sync with the servers, automatically synchronizing changes across the devices of the same user. Since files can be shared with other users, changes in shared folders must be also synced with every account that has been given access to the shared folders. Making this process scalable is not straightforward, as it involves the intricate interaction of many components. For instance, to improve scalability, Personal Clouds use numerous techniques to only transfer those parts of the files that have been modified since the last synchronization. To achieve this,

these systems internally do not use the concept of files, but split every file into chunks, each treated as an independent object. If a chunk is already stored in the storage servers, it is not transferred, saving traffic and storage. However, working at the chunk level requires the sync process to manage more metadata than operating at the file level, making it more critical the way metadata is internally processed. Further, as the amount of metadata is directly proportional to the number of chunks, the efficiency of the chunking algorithm impacts the performance of file syncing.

Personal storage

Such a novel business model presents the raw storage service as a utility, which facilitates the proliferation of innovative companies that deliver a value-added storage service on top of it. In this sense, Dropbox, Box.net or Sugarsync are clear examples of this new kind of storage companies (Personal Clouds); they offer sophisticated storage services to end-users by making use of raw storage provided by data-center owners. As main functionalities, these services typically include file backup and synchronization, folder sharing with other users and file versioning. Due to this simplicity and the set of features mentioned, Personal Clouds are a promising paradigm that have attracted a tremendous amount of users. Recent forecasts from Forrester research estimated a $12 billion market in 2016 involving massive user subscriptions to Personal Cloud services.

Personal storage benefits from the raw storage offered by providers and build a value-added service on top of it.

- **Dropbox** [12] is service that offers cloud storage, file synchronization, file versioning, folder sharing and an intuitive user interface. Dropbox makes use of Amazon's S3 storage system to store their data and offers their service on a Freemium[1] business model. Concretely they offer a free account of $2GB$, paid accounts of $50GB$, $100GB$ and team accounts starting from $1TB$. They incentive users to bring new customers by awarding them with extra space. Dropbox provides clients for major desktop operating systems and mobile systems. According to Forbes, in October 2011 Dropbox had 50 million users, of which 96% were using a free account.

- Google has recently entered on the personal storage by introducing **Google Drive** [11], which basically is an upgrade of the former Google Docs. Google Drive works in much the same way as the previously mentioned services. It launches on the web, for Mac, Windows, Android and iOS devices. It's strengths is its integration with other Google services like Gmail, Android or Google+. It also offers an API and provides users with $5GB$ of cloud storage free of charge and paid accounts ranging from 25GB at $\$2,49$ to 16TB at $799.99.

[1] Freemium is a business model by which a product is offered free of charge, but a premium product with advanced features at a charge.

2.1.3 Blocks and Interaction in a typical Cloud Storage

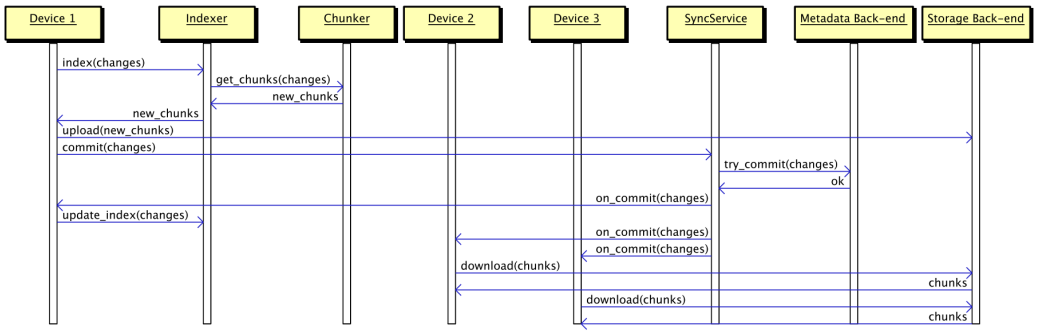


Figure 2.1: Blocks Personal Cloud Interactions.

Personal Clouds usually provide desktop clients that integrate with the OS file explorer capabilities. These clients include a *Watcher* component that monitors file changes in a specified synced folder. We will call this working folder the *Workspace*. When the *Watcher* is notified of any change in the Workspace by the OS, it then informs the *Indexer* component on the new changes. The *Indexer* maintains a local database with information about the contents of the Workspace including files, folders, hashes, and even versions. Internally, Personal Clouds typically do not use the concept of file, but rather operate on a lower level by splitting files into chunks, each treated as independent object and identified by a fingerprint (like SHA-256 hash). In that case, the local database maps the fingerprints to the corresponding files. The reason to work at the sub-file level is to transfer to the *Storage back-end* (Dropbox uses Amazon S3 as storage back-end) only those parts of files that have been modified since the last synchronization, saving traffic and storage costs. The component responsible for partitioning files and calculating the hash values is the *Chunker*. The system could either use fixed-sized chunks or variable-sized chunks. Either way, when a new file is added into the Workspace, all the new chunks will be indexed in the local database. If an existing file is updated, only the affected chunks will be indexed. Once the new chunks are indexed, the *Indexer* will then apply the appropriate source-based *deduplication* policy to transfer only the unique chunks to the *Storage back-end*. If deduplication is on a per user-basis, it suffices for the *Indexer* to compare the hashes of the new chunks with the ones in the local database. If some of the chunks already exist, only the new ones will be uploaded to the *Storage back-end*. If deduplication is cross-user, then the *Indexer* will have to ask first the synchronization service, *SyncService* for short, by sending to it the hash signatures of chunks. The *SyncService* will use the hash signatures to check for the existence of chunks and then notify the *Indexer* to upload the missing chunks to the *Storage back-end*. Here we find the first functional dependency between components, as the efficiency of the *Chunker* determines the amount of data to transfer to the *Storage back-end*. Ideally, only those parts of the file that have been modified should be sent

over the network, for which the choice of the chunk size is critical. Clearly, the best choice depends on both the number and granularity of changes. If a single character is changed in a file, a large-sized chunk will require transferring a lot of duplicate data, while a smaller chunk will offer a greater saving in network bandwidth. The downside of using small chunks is that it will increase the amount of metadata to be managed by the *SyncService*, directly impacting on the system's scalability.

Once the unique chunks are successfully submitted to the *Storage back-end*, the *Indexer* will communicate with the *SyncService* to commit the changes to the *Metadata back-end*, which is the component responsible for keeping versioning information. The *Metadata back-end* may be a relational database like MySQL or a non-relational data store, now frequently called NoSQL databases.

Irrespective of the chosen database technology , the *SyncService* needs to provide a consistent view of the synced files. Allowing new commit requests to see uncommitted changes may result in unwanted conflicts. As soon as more than one user works with the same file, there is a good chance that they accidentally update their local working copies at the same time. It is therefore critical that metadata is consistent at all times to establish not only the most recent version of each individual file but to record its (entire) version history. Although relational databases process data slower than NoSQL databases, NoSQL databases do not natively support ACID transactions, which could compromise consistency, unless additional complex programming is performed.

Finally, when the *SyncService* finishes the commit operation, it will then notify of the last changes to all out-of-sync devices. The device that originally modified the local working copy of the file will just update the *Indexer* upon the arrival of the confirmation from the *SyncService*. The other devices will both update their local databases and download the new chunks from the *Storage back-end*.

Here we are assuming that an efficient communication middleware mediates between devices and the *SyncService*. This middleware should support efficient marshaling and message compression to reduce traffic overhead. Very importantly, it should support scalable change notification to a high number of entities, using either pull or push strategies. To deduplicate files and offer continuous reconciliation , recall that the local database at the Indexer must be in sync with the *Metadata back-end*, for which notification must be performed fast.

Figure 2.1 illustrates the interaction between all the components of a file sync engine. Observe that not all the different components described in this section are present in the architecture of a Personal Cloud. In some architectures, our overall model could be simplified and one component could be responsible for many tasks. Some architectures can even lack some components.

2.2 Optimization Communication Mechanisms

There exist a big problem on cloud synchronization mechanism by itself. To be able to maintain all information synchronized its needed to backup information related to data modified on every synchronization. Cloud systems must know when and where the changes are made and which was the ancient state of data before the modification. That is translated to send information about the modification with the data modified itself. This additional information add and overhead to communication process between devices and cloud services. A big goal is to reduce this traffic on communications.

In Table 2.1 we can see the amount of data added to a file modification on a cloud service provider like Dropbox. Various mechanisms are used to reduce traffic between client devices and cloud server, we focus on two of them: chunking and bundling. They can be used in combination or by itself. In Table 2.2 we can see some of the capabilities of commercial cloud Providers extracted from [9]. This is explained on next sections.

File Size	Real Traffic	ratio Overhead
1 KB	6.8 KB	40.1
10 KB	13.9 KB	4.63
100 KB	118.7 KB	1.528
1 MB	1.2 MB	1.22

Table 2.1: Overhead traffic on Dropbox Sync Folder.

	Dropbox	SkyDrive	Google Drive
Chunking	*Static*, 4MB	*Dynamic*	*Static*, 8MB
Bundling	*yes*	*no*	*no*

Table 2.2: Capabilities implemented on some Cloud Services.

2.2.1 Bundling

Bundling techniques try to group changes. There are several ways of doing this, bundling on the same file and bundling with different files.

Bundling on the same file is usually implemented by defining a window of W seconds in which all changes are stored. When seconds window is exhausted, it is the last change which is sent to the server, thus saving a large amount of traffic caused by small changes. In combination with chunking must be a powerful tool to reduce traffic.

In Figure 2.2 it is shown that given a W seconds bundling window, if we do 4 modifications on a file, they are bundled to finally only send 3 of them. Changes A and C affect the same portion of file and only the last change is transmitted deprecating the changes before on the same window W.

When a batch of files needs to be transferred, files could be bundled and pipelined so that both transmission latency and control overhead impact are reduced. This technique is not implemented on every commercial cloud service.

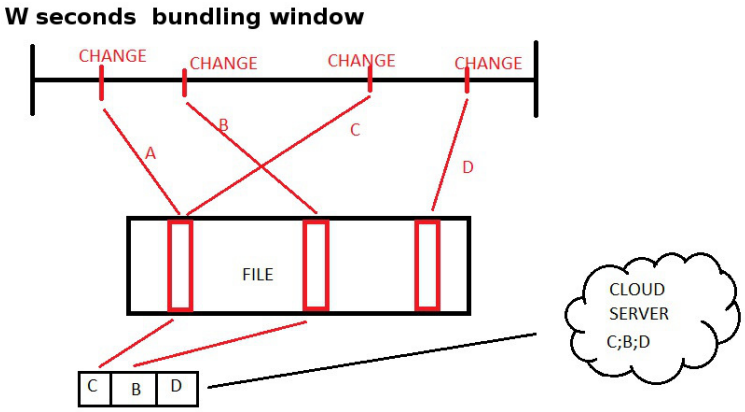


Figure 2.2: Bundling mechanism.

2.2.2 Chunking

Chunking are used to reduce the amount of data re-sent to servers. Imagine a file in which you are entering data and grows over time. Send the same file on every sync when perhaps only a few bytes have changed generate a huge amount of traffic. With some minimal changes maybe it is not needed to re-send the entire file, with this philosophy chunking techniques are applied. Static or dynamic chunking divide the file into blocks for deduplication and also allows that when modifications occur only it is sent those blocks that have changed since the last update, reducing a lot of traffic between clients and server.

Chunking is basically break a long byte sequence S into a sequence of smaller sized blocks or chunks, such that the chunk sequence is stable under local modification of S. Intuitively, that means that if we make a small change on S , turning it S', and apply chunking algorithm to S', most of chunks created first time on S remain unchanged.

Traditionally fixed size blocks are used to break a byte sequence, the main problem of this scheme is that modifying , inserting or deleting even a single bit, in the middle of the sequence will shift all the block and boundaries used changes. Then all following blocks must be rebuild and are different from originals even only one bit is modified.

There is a lot of mechanisms that try to solve this problem trying to adapt block size and reuse blocks unchanged. If we have 5 blocks and change one bit we can reuse 4 blocks and only rebuild one. On this basic idea resides the study of Dynamic chunking algorithms.

2.2.2.1 Chunking Methods

- **Whole-File chunking.** Calculates the hash of whole file contents, and uses it like file's identifier. If we use a SHA-1 here, and due to the collision resistant of this hash function, found two files with the same hash value usually have a very high probability to own the same content. Is simple method but only detects exact file duplication.

- **Fixed-Sized chunking.** All files are partitioned into fixed size blocks, and then SHA-1 function is used to calculate the hash value of all chunks as their identifiers. During process we compare hashes and if we found a same hash, we consider the chunk as redundancy; otherwise, store the chunk and its hash. This method is very sensitive to the insertion and deletion operation like its explained before.

- **Content-Defined chunking.** This is a variable-sized chunking. Chunking boundaries are determined on the contents, so it is more resistant to the insertion and deletion. On this thesis we use TTTD algorithm, one of most variants of this type.

- **TTTD chunking Algorithm.** This algorithm was proposed by HP laboratory [13]. It uses four parameters, the maximum threshold, the minimum threshold, the main divisor, and the second divisor. The maximum and minimum threshold are used to eliminate very large-sized and very small-sized chunks in order to control the variations of chunk-size. The main divisor makes the chunk-size close to our expected chunk-size. the second divisor assists algorithm to determine a backup breakpoint for chunks in case the algorithm cannot find any breakpoint by main divisor. That can be shown on Figure 2.3.

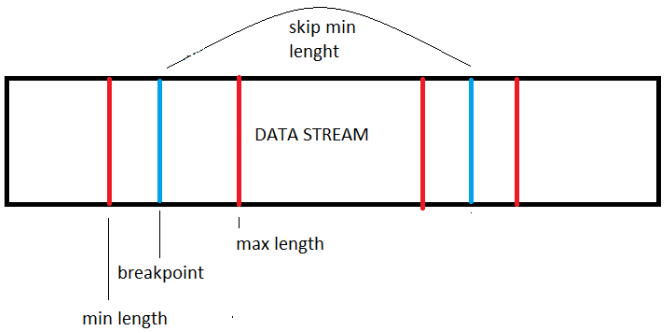


Figure 2.3: TTTD chunking mechanism.

A deep description on how it works is on [6] and a brief explanation of this method is as follows.

1. The algorithm shifts one byte at one time and computes the hash value.
2. If size from last breakpoint to current position is larger than minimum threshold, it starts to determine the breakpoint by second and main divisors.
3. Before the algorithm reaches the maximum threshold, if it can find a breakpoint by main divisor, then uses it as the chunk boundary. The sliding window starts at this position and repeats the computation and comparison until the end of file.
4. When the algorithm reaches the maximum threshold, it uses the backup breakpoint if it found any one, otherwise use the maximum threshold as a breakpoint.

2.3 Bittorrent

BitTorrent is a P2P application that capitalizes the resources(access bandwidth and disk storage) of peer nodes to efficiently distribute large contents. There is a separate torrent for each file that is distributed. Unlike other well-known P2P applications, such as Gnutella or Kazaa, which strive to quickly locate hosts that hold a given file, the sole objective of BitTorrent is to quickly replicate a single large file to a set of clients. The challenge is thus to maximize the speed of replication.

2.3.1 Terminology

The terminology used in the BitTorrent community is not standardized. For the sake of clarity, we define here the terms used throughout this document.

- **Torrent.** A torrent is the set of peers cooperating to download the same content using the BitTorrent protocol.
- **Tracker.** The tracker is the only centralized component of the system. It is not involved in the actual distribution of the content, but it keeps track of all peers currently participating in the download, and it collects statistics.
- **Pieces and Blocks.** Content transferred using BitTorrent is split into pieces, with each piece being split into multiple blocks. Although blocks are the transmission unit, peers can only share complete pieces with others.
- **Metainfo file.** The metainfo file, also called a torrent file, contains all the information necessary to download the content and includes the number of pieces, SHA-1 hashes for all the pieces that are used to verify received data, and the IP address and port number of the tracker.
- **Interested and Choked.** We say that peer A is interested in peer B when B has pieces of the content that A does not have. Conversely, peer A is not interested in peer B when B only has a subset of the pieces of A. We also say that peer A is choked by peer B when B decides not to send any data to A. Conversely, peer A is unchoked by peer B when B is willing to send data to A. Note that this does not necessarily mean that peer B is uploading data to A, but rather that B will upload to A if A issues a data request.
- **Peer Set.** Each peer maintains a list of other peers to which it has open TCP connections. We call this list the peer set, and it is also known as the neighbor set.
- **Local and Remote Peers.** When describing the choking algorithm, we take the viewpoint of a single peer, which we call the local peer. We refer to the peers in the local peer's peer set as remote peers.
- **Leecher and Seed.** A peer can be in one of two states: the leecher state, when it is still downloading pieces of the content, and the seed state, when it has all the pieces and is sharing them with others.

All this interactions can be shown on Figure 2.4.

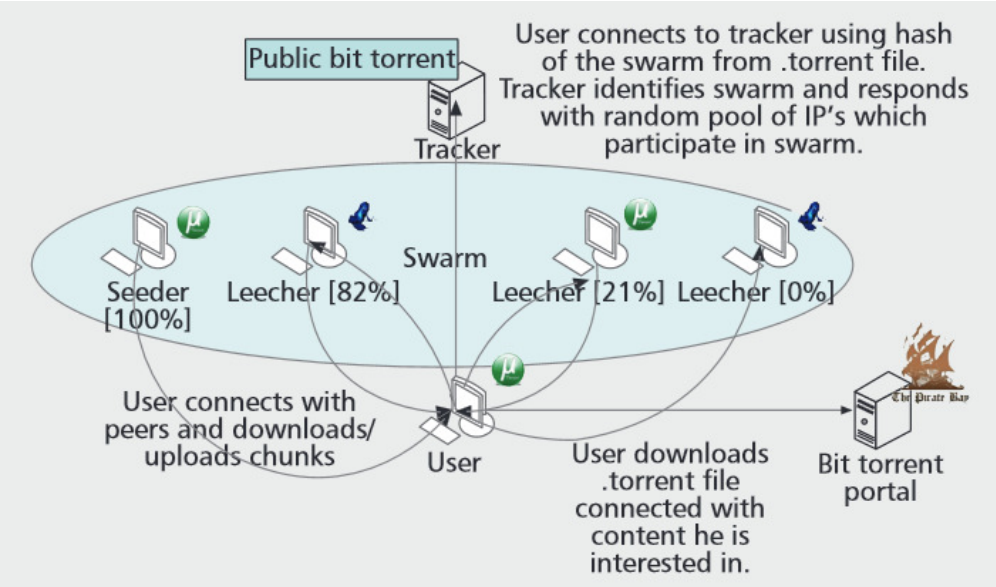


Figure 2.4: BitTorrent protocol Flow.

2.3.2 BitTorrent Operation

Prior to distribution, the content is divided into multiple pieces, and each piece into multiple blocks. The metainfo file is then created by the content provider. To join a torrent, a peer P retrieves the metainfo file out of band, usually from a well-known website, and contacts the tracker that responds with a *peer set* of randomly selected peers, possibly including both *seeds and leechers*. P then starts contacting peers in this set and requesting different pieces of the content.

Most clients nowadays use the rarest-first algorithm for piece selection. In this manner, *peer* selects the next piece to download from its rarest-pieces set. A local peer determines which pieces its remote peers have based on bitfield messages exchanged upon establishing new connections, which contain a list of all the pieces a peer has. *Peers* also send have messages to everyone in their *peer set* when they successfully receive and verify a new piece. A *peer* uses the choking algorithm to decide which peers to exchange data with. The algorithm generally gives preference to those peers who upload data at high rates. Once per rechoke period, typically set to ten seconds, a peer re-calculates the data receiving rates from all peers in its peer set. It then selects the fastest ones, a fixed number of them, and uploads only to those for the duration of the period.

A user joins an existing *torrent* by downloading a torrent file (usually from a Web server), which contains the IP address of the *tracker*. To initiate a new *torrent*, one thus needs at least a Web server that allows to discover the *tracker* and an initial *seed* with a complete copy of the file. To update the tracker's global view of the system, active clients periodically (every 30 minutes) report their state to the tracker or when joining or leaving the torrent. Upon joining the torrent, a new client receives from the tracker a list of active peers to connect to. Typically, the tracker

provides 50 peers chosen at random among active peers while the client seeks to maintain connections to 20-40 peers. If ever a client fails to maintain at least 20 connections, it recontacts the tracker to obtain additional peers. The set of peers to which a client is connected is called its *peer set*.

The clients involved in a torrent cooperate to replicate the file among each other using warming techniques: the file is broken into equal size chunks (typically $256kB$ each) and the clients in a peer set exchange chunks with one another. The swarming technique allows the implementation of parallel download where different chunks are simultaneously downloaded from different clients .

Each time a client obtains a new chunk, it informs all the peers it is connected with. Interactions between clients are primarily guided by two principles. First, a peer preferentially sends data to peers that reciprocally sent data to him. This "tit-for-tat" strategy is used to encourage cooperation and ban "free-riding".Second, a peer limits the number of peers being served simultaneously to 4 peers and continuously looks for the 4 best downloaders (in terms of the rate achieved) if it is a seed or the 4 best uploaders if it is a leecher.

BitTorrent implements these two principles, using a "choke/unchoke" policy. "Choking" is a temporary refusal to upload to a peer. However, the connection is not closed and the other party might still upload data. A *leecher* services the 4 best uploaders and chokes the other peers. Every 10 seconds, a peer re-evaluates the upload rates for all the peers that transfer data to him. There might be more than 4 peers uploading to him since first, choking is not necessarily reciprocal, and second, peers are not synchronized. He then *chokes* the peer, among the current top 4, with the smallest upload rate if another peer offered a better upload rate. Also, every 3 rounds, that is every 30 seconds, a *peer* performs an optimistic *unchoke*, and unchokes a peer regardless of the upload rate offered. This allows to discover peers that might offer a better service (upload rate). Seeds essentially apply the same strategy, but based solely on download rates. Thus, *seeds* always serve the *peers* to which the download rate is highest.

2.4 Related work

There exist some papers that try to model how some personal cloud systems behave and try to improve its performance. On this field Dropbox is characterized on [16], describing typical usage, traffic patterns, and possible performance bottlenecks. but realizes that a characterization of underlying client processes that generate workload to the system is still lacking. Another authors , like in [23] proposes and design a middleware to improve comunication between clients and server by reducing the overhead caused by session maintenance traffic, while preserving the rapid file synchronization, Also, show how cloud storage applications exhibits pathological inefficiencies in the presence of frequent, short updates to user data.

On BitTorrent utilization research we have found a pioneer contribution on [5] where authors modelize the behavior of bittorrent with small files through formulas that measure the gain and the offload in front of regular HTTP transfers. Also, in [22] we have found a needed ideas to calculate overheads on communication using peer-assisted content distribution. The utilization of peer-to-peer like an alternative to classical server-based content distribution is discuss on [19] and authors demonstrate that really , BitToirrent is a realistic and inexpensive alternative. In fact, the efficiency of the BitTorrent protocol makes it especially suitable for massive content distribution while reducing bandwidth costs in the Cloud like is extracted from [4]. Here authors develop a tool to measure the benefits of use BitTorrent in front of classical HTTP protocol to synchronize devices and finally conclude that BitTorrent proved to be a very efficient technology that outperforms classical centralized transfer solutions.

As explained on [20] peer-to-peer protocols are used on many well known applications like Twitter and Facebook, but not always is reached a minimum distribution time desired on data centers, in this paper is presented a new P2P application that uses more bandwidth eliminating overheads inherents to Internet protocols. [21] discuss the application of BitTorrent as a distribution data protocol on data centers. The same paper examines some variants of BitTorrent like Murder, BitTornado and LANTorrent to name a few, and how the configuration parameters can affect its performance. [7] proposes a global management architecture and a set of algorithms that improve the transfer times of communication on datacenters.

The research of file systems behavior on a cloud system is not characterized in any way, but we have found valuable information about file systems characteristics and its behavior over time in [1] and [27] where authors develop a generic model of file system changes based on properties measured on terabytes of real, diverse storage systems. Aiding us to build a synthetic scenario that try to emulate a shared file system on a cloud server.

Any study has linked the influence of chunking mechanism, Bittorrent and Bundling mechanisms to offload Cloud Servers when we use small size files, and this thesis deeps on this topic.

Chapter 3

Understanding the Interplay between Bundling and P2P content Delivery

To study if file bundling in conjunction with BitTorrent can cut down bandwidth costs on file syncing scenarios and how bundling works in combination with other basic mechanisms like content-based chunking, we have designed some experiments on different typical syncing scenarios and variables to model this scenarios.

Since users of Personal cloud services are not characterized by the moment, we have thought about three typical scenarios where our study could be focused. This scenarios may permit to apply chunking and bundling techniques over its content and may permit different devices/users to sync this content simultaneously.

- First, we try to recreate a **real file system scenario** following norms and formulas in [1].
- Second, we will simulate a **frequent but small updates scenario** with the same file and select a real Apache server Log file in which all updates are small (a few bytes each) and are appended to the end of file.
- Finally, we thought on a scenario in which many users/devices can edit the same file simultaneously, that is a **collaborative editing scenario**, and pick two files from Wikipedia on a selected date where a lot of modifications were made to.

To implement the bundling mechanism we define a time window W that group changes before they are transmitted to the cloud server. This W has three different values: 30 seconds, 60 seconds and 120 seconds. These time windows have been selected taking as reference the 60 seconds window estimated for Dropbox according to [10] and [16], then, we have decided to half and double it in order to study how bundling mechanisms affect on servers workload for different W.

To model BitTorrent behavior on this scenarios we will use formulas found in [5]. The main idea here is to compare the HTTP and BitTorrent yields in terms of offloading cloud servers and take benefit of client bandwidths to do the synchronization.

3.1 BitTorrent Model used

Cloud services require a huge amount of storage and bandwidth. Our Research Group is pioneer on findings about reducing costs in personal cloud systems by introducing Bittorrent protocol for synchronization purposes.

It is assumed that BitTorrent is only effective for large files. However, this thesis is focused on the study of cloud bandwidth savings without degrading download time. Like explained in [5] [22], we need indicators to decide which option is better : HTTP or BitTorrent.

We consider a classic personal cloud system the one in which the storage server is responsible for storing the clients files and managing the corresponding requests. Each client i has an upload bandwidth u_i and a download bandwidth d_i.

The two following common file distribution scenarios could benefit from this hybrid download strategy:

1. *Synchronization:* User A is adding a new file f to his personal account. During the synchronization process, the same file will be download by all the other synchronized devices of the user. (Part (a) of Fig. 3.1)

2. *Sharing:* User A is sharing a file f with other users. In this case, the file will be downloaded by all the synchronized devices of the users. (Part (b) of Fig. 3.1)

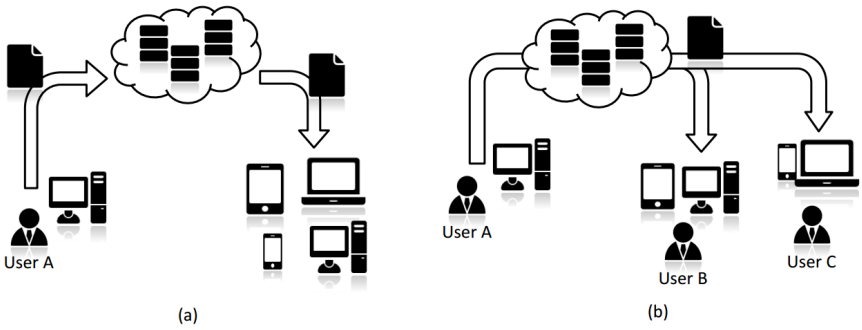


Figure 3.1: Synchronisation and sharing in personal cloud systems

Both cases can be modeled by the problem of distributing a file f of size F from the cloud server to L distinct nodes. We denote by $\mathcal{S}$ the set of cloud seeds serving f and by $u(\mathcal{S})$ their aggregated upload speed dedicated to f. In personal clouds, file synchronization and content distribution follow a client-server model centralized in the cloud storage provider. The download protocol adopted by these providers is HTTP. The choice of this protocol is made because it uses the *port 80* which is generally kept open. While this kind of architecture (*client-server*) is appropriate for some uses cases, it is not optimal when the number of nodes requesting the same content is high, as it might result in bandwidth bottlenecks in the cloud.

A possible solution is to benefit from the high number of requests and use the clients' spare upload bandwidth to offload the server.

The authors of [4] define Gain and Offload indicators as follows.

3.1.1 Notation

We will consider the following notation on next formulas used on this thesis mathematical calculations:

- F: size of the requested file f
- $\mathcal{S}$: set of providers of f (seeder nodes)
- $\mathcal{L}$: set of requesters of f (leecher nodes), L= $|\mathcal{L}|$ is the number of requesters
- $\mathcal{I}$: set of all the nodes, $\mathcal{I} = \mathcal{L} \cup \mathcal{S}$
- u_i: upload speed of node $i \in \mathcal{L}$
- d_i: download speed of node $i \in \mathcal{L}$
- $d_{min} = \min_{i \in \mathcal{L}}(d_i)$: download speed of the slowest leecher requesting f
- $u(\mathcal{A}) = \sum_{i \in \mathcal{A}} u_i$: aggregated upload bandwidth of $\mathcal{A} \subseteq \mathcal{I}$
- $d(\mathcal{A}) = \sum_{i \in \mathcal{A}} u_i$: aggregated download bandwidth of $\mathcal{A} \subseteq \mathcal{I}$
- $C_f = \{(u_i, d_i),\ \forall i \in \mathcal{L}\}$: set of upload and download bandwidths of all the leechers interested in f.

3.1.2 The Distribution Time for Small Files in BitTorrent

To get an estimation of the download time in BitTorrent-like systems, we borrow the following formula proposed in [22] by Kumar and Ross:

$$T_{pa}\left(u(\mathcal{S}), C_f, F\right) = \frac{F}{\min\left\{d_{min}, \frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}}, \tag{3.1}$$

where T_{pa} is the minimum time needed to distribute a file of size F to L leechers. This time depends on the download speed of the slowest peer d_{min}, the aggregated upload bandwidth of all the nodes divided equally between all the L leechers, and the upload bandwidth of the cloud seed(s). The authors presented in their paper a complete proof of the download time. The proof is organized into the following exhaustive cases depending on the parameter that may be responsible for the transfer bottleneck:

1. Case A: $d_{min} \leq \min\left\{\frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$ and $d_{min} \leq \frac{u(\mathcal{L})}{\mathrm{L}-1}$:
   In this case, the download speed of the peers is limited by the download bandwidth of the slowest peer in the swarm d_{min}.

2. Case B: $d_{min} \leq \min\left\{\frac{u(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}$ and $\frac{u(\mathcal{L})}{\mathrm{L}-1} \leq d_{min}$:
   In Case B, the transfer is limited by the maximum speed at which a leecher can get data from the other leechers, that is $\frac{u(\mathcal{L})}{\mathrm{L}-1}$.

3. Case C: $\frac{u(\mathcal{I})}{\mathrm{L}} \leq \min\{d_{min}, u(\mathcal{S})\}$:
   The transfer bottleneck in this case is limited by the aggregated upload speed of the network $u(\mathcal{I})$ divided equally between the L leechers.

4. Case D: $u(\mathcal{S}) \leq \min\left\{d_{min}, \frac{u(\mathcal{I})}{\mathrm{L}}\right\}$:
   In this case, the upload bandwidth of the seed $u(\mathcal{S})$ is the maximum limit at which each peer can download "fresh" content.

For each of the cases listed above, the authors in [22] constructed a seeding rate profile $s_i(t)$ which denotes the bit rate at which the seeds send pieces to leecher i at time t.
The adopted distribution scheme is the following: As soon as a leecher li begins to receive data from the seed, it replicates it to each of the other $(\mathrm{L}-1)$ leechers at a rate $x_i(t)$, where $x_i(t) \leq s_i(t)$. For each case, the distribution scheme consists of L application-level multicast trees, each rooted at a specific seed, passing through one of the leechers and terminating at each of the $\mathrm{L}-1$ other leechers.

To calculate the offload ratio in the following section, we need to measure the volume of data offloaded from the cloud. The authors of [4] present the seeding rate for each case. This rate, denoted by $s_i(t)$ for the sake of clarity, depends on the time t, the file size F, the upload speed of the seeds $u(\mathcal{S})$, and the set of upload and download speeds of all the leechers C_f. For a complete proof and more details regarding these formulas, we kindly refer the reader to the original paper [22].

$$s_i(t) = \begin{cases} \frac{u_i \times d_{min}}{u(\mathcal{L})} & \underline{Case\ A} \\ \frac{u_i - u(\mathcal{L})}{\mathrm{L}-1} + d_{min} & \underline{Case\ B} \\ \frac{u_i - u(\mathcal{L})}{\mathrm{L}-1} + \frac{u(\mathcal{I})}{\mathrm{L}} & \underline{Case\ C} \\ \frac{u_i \times u(\mathcal{S})}{u(\mathcal{L})} & \underline{Case\ D} \end{cases} \tag{3.2}$$

3.1.3 BitTorrent overheads

One of the limitations of (3.1) is that it does not take into consideration the overhead that peer-assisted systems may present compared to the client-server ones. These overheads may be neglected for large files. However, they cannot be ignored for the small ones, for which the download time is in the order of a few seconds.

These overheads can be mainly of two types, each related to a different phase of BitTorrent:

- *Overhead related to the start-up phase:* Before starting the download, there are a few steps that each leecher needs to perform: First, the leecher has to get and read the *.torrent* file that contains all the meta-info data about the requested content. And then, it needs to contact the tracker(s) to get a list of other peers sharing or downloading the same file. After locating and connecting to the peers, the leecher can finally begin the transfer.

This overhead is relative to the architecture of the system. It can be monitored and dynamically adapted based on the load of the system. In [5] this overhead is experimentally studied and it can be simply modeled as a constant duration α_{bt} added to the download time.

- *Overhead related to the download phase:* In BitTorrent, peers upload to each other even though they may only have parts of the file. This can result in upload interruptions when the uploader has no pieces to offer to his unchoked peers.
  Fortunately, this problem has already been tackled in [24], where the authors introduced a parameter to scale down the upload speed of leechers. This parameter, denoted as $\eta \in [0,1]$, measures the effectiveness of file sharing. It can be computed as follows:

$$\eta = 1 - \mathbb{P}\left\{ \begin{array}{c} \text{downloader } i \text{ has no piece that} \\ \text{his unchoked peers need} \end{array} \right\}.$$

Considering the above listed overheads, authors of [4] were able to extend Eq. (3.1) in order to provide an accurate estimation of the download time in BitTorrent as follows:

$$T_{bt}\left(u(\mathcal{S}), C_f, F\right) = \frac{F}{\min\left\{d_{min}, \frac{u'(\mathcal{I})}{\mathrm{L}}, u(\mathcal{S})\right\}} + \alpha_{bt}, \tag{3.3}$$

where $u'(\mathcal{I}) = u(\mathcal{S}) + \eta\, u(\mathcal{L})$ is the scaled aggregated upload speed of all the nodes, including both the seeders and the leechers.

3.1.4 Gain

This variable is an indicator that help us decide if BitTorrent is a better option than HTTP synchronization, the major purpose of this thesis. If this calculated value is positive , implies that synchronization with BitTorrent is faster than HTTP. Conversely, if this value is negative, implies that traditional HTTP synchronization is better than BitTorrent. Formula on 3.4, is applied on four cases that vary depend on speed connection of sharing users and where the bottleneck on its interconnection resides.

On *Case I* this bottleneck is the speed of slowest peer. On *Case II* is $\frac{u(\mathcal{S})}{\mathrm{L}}$, while it is equal to d_{min} in BitTorrent. On *Case III* is $\frac{u'(\mathcal{I})}{\mathrm{L}}$. and finally on *Case VI* is $\frac{u(\mathcal{S})}{\mathrm{L}}$ is always $\leq d_{min}$.

$$Gain\left(u(\mathcal{S}), C_f, F\right) = \begin{cases} -\frac{\alpha_{bt} \times d_{min}}{F} & \textit{Case I} \\ 1 - \frac{u(\mathcal{S})}{\mathrm{L}.d_{min}} - \frac{\alpha_{bt}.u(\mathcal{S})}{F \times \mathrm{L}} & \textit{Case II} \\ 1 - \frac{u(\mathcal{S})}{u'(\mathcal{I})} - \frac{\alpha_{bt}.u(\mathcal{S})}{F \times \mathrm{L}} & \textit{Case III} \\ 1 - \frac{1}{L} - \frac{\alpha_{bt} \times u(\mathcal{S})}{F \times \mathrm{L}} & \textit{Case IV} \end{cases} \tag{3.4}$$

3.1.5 Offload

This variable indicates the percentage data saved if we use BitTorrent instead of HTTP synchronization mechanisms. It will be calculated only when Gain 3.4 is a positive value. This definition is explained deeper in [22] [5] .

$$Offload\,(u(\mathcal{S}), C_f, F) = \begin{cases} 1 - \frac{1}{\mathrm{L}} & \underline{Case\ A} \\ \frac{\eta.u(\mathcal{L})}{\mathrm{L} \times d_{min}} & \underline{Case\ B} \\ 1 - \frac{u(\mathcal{S})}{u'(\mathcal{I})} & \underline{Case\ C} \\ 1 - \frac{1}{\mathrm{L}} & \underline{Case\ D} \end{cases} \tag{3.5}$$

3.2 Methodology

In this section, we describe the methodology that will follow to determine the effect of bundling in the tree scenarios we consider in this thesis.

Before going any further, we need first to clarify some of the concepts and metrics that will be used to assess the impact of bundling on the amount of data to be pushed in a peer-to-peer fashion to the out-of-sync devices:

- W is the seconds window size that we choose to implement the bundling.
- τ is a quality service restriction. When this value is positive we are forcing that BitTorrent to be faster than HTTP synchronization. When this value is negative we force BitTorrent to be slower than HTTP synchronization. If this value is zero, the two synchronization protocols will have the same benefits.
- *The Gain* is a value that shows us when is advisable to change from HTTP to BitTorrent synchronization. We look for positive values to calculate the quantity of data offloaded in each W.
- *The Offload* is the quantity of data offloaded from the cloud servers in a given W.

What it is done on all experiments of each scenario of 3.3 is to calculate the quantity of data that could be transferred to *cloud server* for all possible W with static and dynamic chunking. Then, we apply formulas from [5] to calculate the *Gain* varying τ values from -1 *to* 1 on each W. If $\tau = -1$ we are forcing that *BitTorrent* is twice slower than *HTTP*. However, if $\tau = 1$ we are forcing that *BitTorrent* is twice faster than *HTTP*. When $Gain \geq \tau$ then we apply the formulas in [5] again to calculate the *Offloaded* data on these W. Even when *BitTorrent* is twice slower than *HTTP*, if the amount of offloaded data is so big, the use of this P2P protocol may be advisable.

	Concept
W	Time window (seconds)
τ	Service restriction modifier
Gain	Change protocol indicator
Offload	Amount of data transferred to be saved

Table 3.1: Methodology parameters.

3.3 Typical Scenarios

In this section we will present some scenarios that will be studied to see if its offloaded percentage advise us to change from HTTP communication protocol strategy to BitTorrent. Different W are used for bundling in all these scenarios using static and dynamic chunking techniques, and τ values are applied from -1 to 1. We will also study how combining different techniques like bundling and chunking could affect synchronization on cloud servers.

3.3.1 File System Dataset

We reproduce here a corporate environment in which multiple users/devices modify a given file system. This modifications could be: the creation of a new file, the deletion of some files or modifications of any kind on some files too. The file system consists of files which sizes vary from a few KBs to a few MBs, so changes made to file system will always be of the same order of magnitude. Since most files are $< 1MB$ [1] the modifications will be of similar magnitude.

We use different techniques of chunking to see the impact on the file system sample. The aim of the study is to understand the behavior of a cloud service and how could be offloaded if a Corporation decides to move a file system, shared by a few users/devices, to a commercial cloud system.

We have been looking up for a real Dataset to run experiments. That is not trivial, the most common Datasets used on papers tend to be private. If the Dataset is public it is hard to reproduce it, or they are synthetic and without generation details [27].

The desired Dataset is the one that contains real data from users, which has the advantage of representing actual workloads. The main problem here is that this real data tend to be restricted and is not released for studies.

Since private dataset option is deprecated, we tend to find a public solution. The one selected has been "linux kernel source code". We hope that with this source code and applying the directives found in [27] we will be able to recreate a File System life cycle.

3.3.2 Log File Appending

Imagine a situation in which we need to store a log file with a fast update rate on a cloud personal service. This updates tend to be small ones, implying only a few bytes to be appended to the end of a file. We will discuss how a personal cloud can behave on this kind of situation applying chunking and bundling mechanisms. The idea is to study how to reduce the amount of data re-sent to the server and try to see how the traffic ratio could be improved, that is, how mechanism could improve communication traffic impact over this environment.

We have reproduced, from a real apache HTTP log file trace, an environment where: short updates of few bytes each one are appended to the same file. This

changes are done by a unique user/device, and this file is synchronized by multiple users/devices.

3.3.3 Collaborative Editing

Another real environment that is becoming more popular now a days is to share a document with multiple users that can do modifications over time (ex. evaluation spreadsheet on school), or maybe a file that is shared by different devices and always must be synchronized with the last version. These modifications tend to be small ones (a few bytes) and quickly replicated to all copies of the document. The main idea here is to study how techniques to improve communication like bundling and chunking affect the traffic and how could be offloaded the cloud server if a different protocol of synchronization is used. With this purpose BitTorrent is used trying to extrapolate which method must be used on future personal cloud services designs.

We have reproduced, from a real Wikipedia history log file, an environment where: short updates of few bytes each one are applied to a unique file. This changes are done by multiple users at the same time and this file is synchronized by multiple devices.

3.3.4 Scenarios Summary

In this section are summarized the experiments characteristics to clarify its differences. It can be done easily watching Table 3.2. The more relevant aspects to differentiate the three scenarios to be studied are:

- *Number of devices/users that could generate a modification.*
- *Average modification size when both bundling and chunking mechanisms are applied.*
- *Number of files affected on each modification, it includes all changes on the same window.*

	# Change Generator	Modification Size	# Files Affected
File System	*Various*	*Mbytes*	*Several*
Log File Appending	*One*	*Bytes*	*One*
Collaborative Editing	*Various*	*Bytes*	*One*

Table 3.2: Experiments Comparative.

Chapter 4

Experiments

Here we try to reproduce some environments to study the expected behaviour of cloud services and the traffic generated keeping in mind to study the impact of techniques like bundling and chunking always trying to offload cloud servers.

4.1 Experimental Setup

The aim of this experiments is to study if it is feasible to incorporate BitTorrent like synchronization protocol between devices when the amount of data to be updated is small

Dropbox is arguably the most popular cloud storage service, reportedly hitting more than 100 million users who store or update one billion files per day [18].

As shown in [23] 8.5% Dropbox users, $\geq$ 10% of their traffic is generated in response to frequent short updates .

Approximately 90% of files on a personal File System is smaller than $1MB$ and 50% of files are smaller than $4KB$ [1].

On StackSync Group, the static chunk size chosen is $512KB$ so this is our election to proceed with experiments on offload Cloud services servers with this type of chunking.

Dynamic chunking mechanism adopted on all experiments has been TTTD [13], which parameters are set like shows Table 4.1.

Maximum Size Chunk	32 KB
Minimum Size Chunk	8 KB
Main Backup point	20 KB
Secondary Backup point	10 KB

Table 4.1: TTTD parameters used.

As exposed in [10] and [16], the synchronization time window of Dropbox is about 60 seconds. This is the value we take as reference to create all the experiments. Also, we have experimented with time window sizes that are a half and a double of this 60 seconds trying to study the impact of bundling on workload servers.

To calculate Gains and Offload comparisons between BitTorrent and HTTP synchronization a set of variables must be fixed before using formulas in [4]:

- The upload speed of the seed: $u(\mathcal{S}) = 2$ Mbps. We remind our reader that $u(\mathcal{S})$ does not refer to the total upload bandwidth of the cloud but to the portion of its bandwidth allocated to each specific file/swarm.

- The clients are homogeneous and have an upload and download speed of 512 Kbps and 1 Mbps respectively.
- The peers discovery overhead in [4] is $\alpha_{bt} = 2.5$ seconds.

4.1.1 File System Dataset

This section introduces how we recreate a realistic file system dataset following indications in [27] to study how important could be the dynamic versus static chunking method on cloud storage systems in terms of offloading the server.

To recreate this initial File System we have chosen a linux kernel source code. It accomplishes all the items we need: a lot of small files $< 1MB$ in a high number of sub-folders. The approximate size of this Dataset was $500MB$ initially.

We have applied static and TTTD chunking techniques on the dataset and we have saved the results like original chunks. Later we simulate synchronizing chunks traffic to servers with different synchronization size windows 4.1.

In Figure 4.1 we can see Markov & Distribution state graph we have applied 100 times on the created file system to simulate realistic changes over time [27].

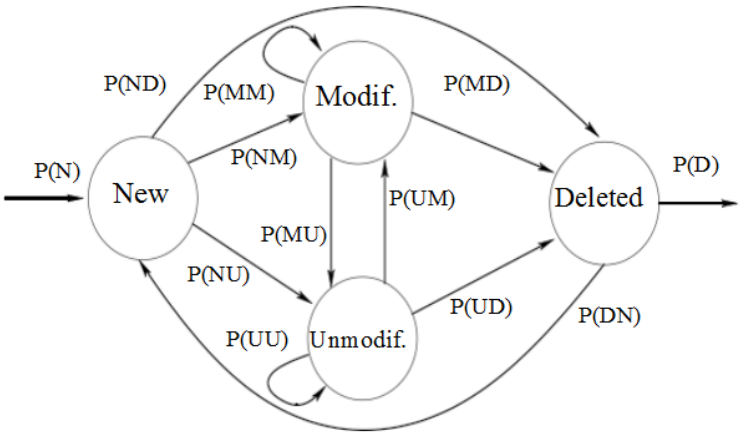


Figure 4.1: Markov & Distribution generation States.

- **N:** new file appearance.
- **D:** file deletion.
- **U:** Unmodified file.
- **M:** Modified file.
- **Combinations** of this letters are transactions between snapshots, for example: MD stands for Modified-Deleted file transaction.

All this states and transitions have an associated probability, and it depends on file system type used [27]. In Table 4.2 we can see probabilities used in these experiments.

All transitions that involve a file modification have an special treatment because of chunk nature. A file is split in chunks and when you modify a file it is needed to know in which part of file the change is done. At this point it is important to

Dataset	N	NM	NU	ND	MU	MD	MM	UM	UD	UU	DN	D
kernels	5	32	65	3	49	3	48	17	3	80	1	3
Home	4	2	78	20	54	10	36	0.14	0.35	99.51	6	0.5

Table 4.2: File systems Probabilities Markov & Distribution.

differentiate between static and dynamic chunking. A modification in one bit on static chunking is equivalent to change all chunks associated to this file while on dynamic chunking the number of chunks affected by this change may vary from one to all the file chunks. As shown in [27] there are three main kind of changes that are to be considered:

- **B:** Beginning of file.
- **E:** End of file.
- **M:** Middle of file.

On Table 4.3 you can see probabilities of this three kind of modifications and combinations between them.

Dataset	B	E	M	BE	BM	ME	BEM
kernels	52	8	7	14	5	3	11
Home	38	3	8	10	11	1	29

Table 4.3: Probabilities modification pattern.

Once all changes have been registered, the following simulations are launched: $1change/2minutes$, $1change/1minute$ and $2change/1minute$ along a whole day, with different bundling windows sizes (W): 30 seconds, 60 seconds and 120 seconds.

This experiment is repeated N times where N is calculated as the number of W that fit in a whole day. Selecting different files changes from all dataset and comparing chunks of same file each time the experiment is ran. It is tried to show amount of data that is accumulated between changes from a hypothetical file system. Static and dynamic chunking is used in all experiments. In Figures 4.2 and 4.3 it is shown the different amount of data bundles are generated in different bundling window sizes combined with different chunking methods.

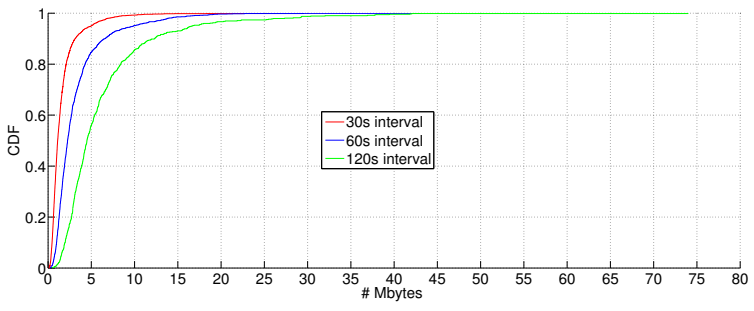


Figure 4.2: CDF Static chunking on file system.

In summary, with this experiment we have tried to study how different bundling windows can affect to the amount of data to be transferred when various files on a file system are affected.

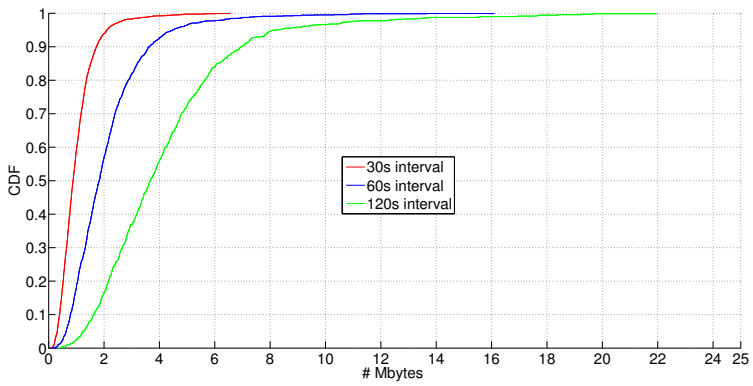


Figure 4.3: CDF Dynamic chunking on file system.

4.1.2 Log appending

In this experiment we examine the behavior of a personal cloud storage in the presence of short updates to a unique file.

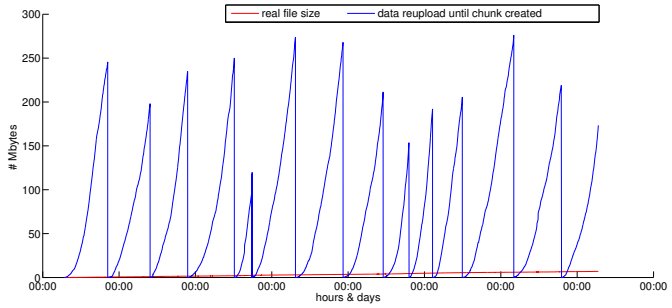
This experiment was created by analyzing a real apache HTTP server Log file. Each line of the log file contains a timestamp which let us know when an event occurs and the amount of bytes written to the log file are the bytes appended to the file.

For this experiment we recreate , with timestamps aid, a lifecycle of this log file on a personal cloud and applying static and dynamic techniques of chunking we perform a behavior study.

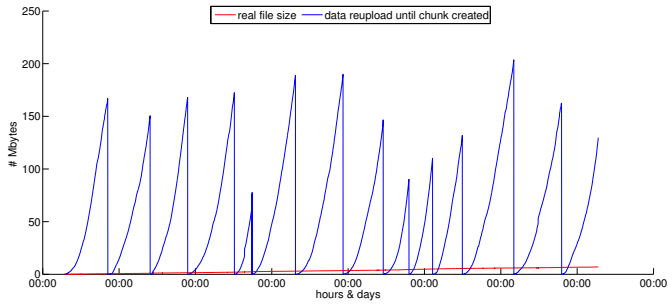
The file used to do experiments is a log file with timestamps between 6am on 29-dec-2013 and 6am on 05-gen-2014. This file begins with size $0bytes$ and the last day is a $7192KBs$ file.

We simulate how a $512KB$ chunk is created with different time window sizes 4.1. The shorter time between synchronization implies more data replicated to the cloud server for each chunk. This can be seen in Figure 4.4.

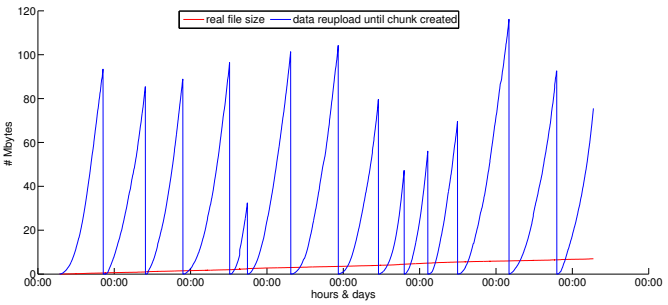
With this experiment we try to explore the consequences of appending short updates of few bytes each to a single file.



(a) Bundling 30s



(b) Bundling 60s



(c) Bundling 120s

Figure 4.4: Bundling chunk creation

4.1.3 Collaborative Editing

In this experiment we simulate a scenario where multiple users are collaboratively editing a document stored in a Personal Cloud Service Sync folder.

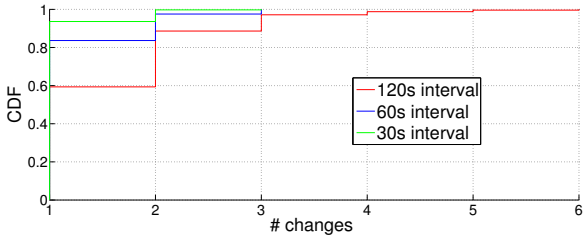
For this experiment we try to simulate the real behavior of a file in collaborative environment, so we search for a document edited simultaneously by different users.

We have found a great repository on wikipedia as it has a lot of files (wikipedia pages) that are edited simultaneously for a lot of users. Also wikipedia API provides information of all revisions a file has suffered along its life.

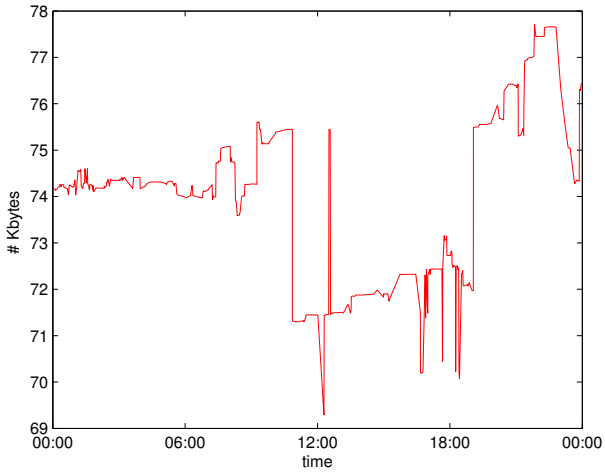
We have connected to Wikipedia API to download a whole day activity in two specific files with high number of revisions: Kosovo English wikipedia page on the independence day proclamation, Michael Jackson English wikipedia page on the day of his death.

- **Kosovo page**[1] has 136 users that do 387 modifications along the day. In Figure 4.5 it is shown how the file evolves during the whole day and the number of changes that match in the same synchronization window.

[1] independence declaration on 18/02/2008



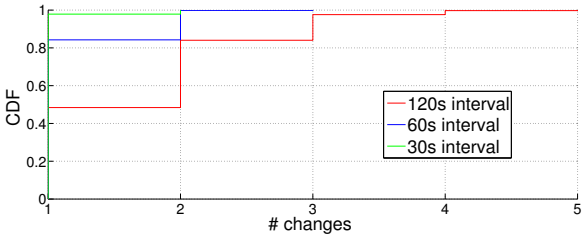
(a) updates matching sync window



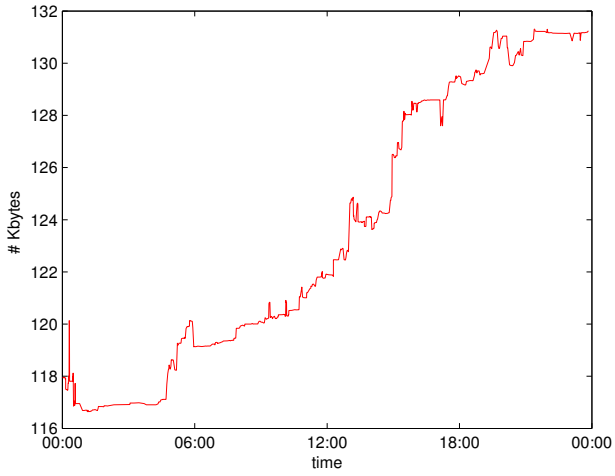
(b) file during day

Figure 4.5: Kosovo Wikipedia Page

- **Michael Jackson page**[2], has 317 users that do 642 modifications along the day. In Figure 4.6 it is shown how the file evolves during the whole day and the number of changes that match in the same synchronization window.



(a) updates matching sync window



(b) file during day

Figure 4.6: Micheasl J. Wikipedia Page

Once we have all the information downloaded, we have parsed files to obtain the user who has done the modification, the amount of data (bytes) modified and the timestamp when modification was done.

Studying this data we have realized that there exist two major types of edit users, the one who only do a modification in along the day and the former who remains connected for a period of time and does successive modifications within hours or even minutes. At this point we define "*user session*" as the period of time between first and last edit done by the same user.

For each window time W we extract the number of devices that have an active session and are going to be synchronized, then we apply the formulas in [22] and [5] to obtain the Gain and Offload data on each W.

In summary, with this experiment we reproduce the behavior of a file modified by multiple devices/users along different user session durations.

[2] one day after his death on 26/06/2009

4.2 Experiment Results

The results of the experiments in this thesis will be explained in separate sections, one for each experiment.

4.2.1 File System Dataset experiment results.

This experiment shows that using dynamic chunking in front of static chunking could save a 33% data traffic on 60 seconds and 120 seconds time windows and a 40% on 30 seconds windows.

The amount of data saved using bundling in combination with static chunking goes approximately from 43% to 49%. Using dynamic this value goes from 36% to 48%.

Both dynamic and static chunking suffer a big impact by increasing τ value bundling on 30 seconds windows. That impact is reduced as the f window time increases. As shown in tables 4.4 and 4.5, when using 120 seconds windows, choosing between static or dynamic chunking , does not affect the relation between amount of data offloaded and improvement of communication.

From the experiment results we can draw two main conclusions:

- First, P2P protocol is strongly recommended to offload cloud servers in this environment.
- Last, Using bundling technique is recommended since it offers a save of at least 50% on traffic communication regardless the chunking method that we use..

	$Window = 120secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	240,9350	240,9350	239,9298
Upload Total (inMBs)	492,0917	492,0917	492,0917
Overall Offload %	48,96	48,96	48,76

	$Window = 60secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	113,5929	113,5929	108,5909
Upload Total (inMBs)	239,1006	239,1006	239,1006
Overall Offload %	47,51	47,51	45,42

	$Window = 30secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	55,6143	55,6143	46,1376
Upload Total (inMBs)	129,1033	129,1033	129,1033
Overall Offload %	43,08	43,08	35,74

Table 4.4: Offload on Static Chunking on File System (5 devices synced).

	$Window = 120secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	158,0073	158,0073	156,7351
Upload Total (inMBs)	326,2837	326,2837	326,2837
Overall Offload %	48,43	48,43	48,04

	$Window = 60secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	73,8110	73,8110	67,2710
Upload Total (inMBs)	160,4154	160,4154	160,4154
Overall Offload %	46,01	46,01	41,94

	$Window = 30secs$		
Gain Constraint τ	**-0.2**	**0**	**0.2**
Offloaded Data (inMBs)	28,9831	28,9831	17,1789
Upload Total (inMBs)	79,2444	79,2444	79,2444
Overall Offload %	36,57	36,57	21,68

Table 4.5: Offload on Dynamic Chunking on File System (5 devices synced).

4.2.2 Log appending experiment results.

These results are analyzed from two different perspectives: the bundling one , and the chunking one.

From bundling perspective, the 30 seconds window data implies that client chunks are updated more often, so the server realizes that the file has changed synchronizing the same data than before with only a few bytes appended. As repeated data is sent to server until chunk maximum size is reached, the repeated data re-upload grows up dramatically along the time. That is shown in Figures 4.7 and 4.8 , where it can be compared the amount of data size re-sent at the $7th$ day.

From chunking point of view, in Figure 4.7 can be obrserved that with a 30 seconds window and static chunking mechanism applied, the data uploaded was $2,95GBs$, for 60 seconds window was $2,04GBs$, and for 120 seconds window was $1,11GBs$.

In Figure 4.8 with dynamic chunking mechanism applied in a 30 seconds window, the data uploaded was $178,8MBs$, for 60 seconds window was $119,6MBs$, and for 120 seconds window was $62,95MBs$.

It must be considered that the real file size by the $7th$ day was was $7MB$. So the amount of data that is sent and repeated is quite big in comparison with the file size.

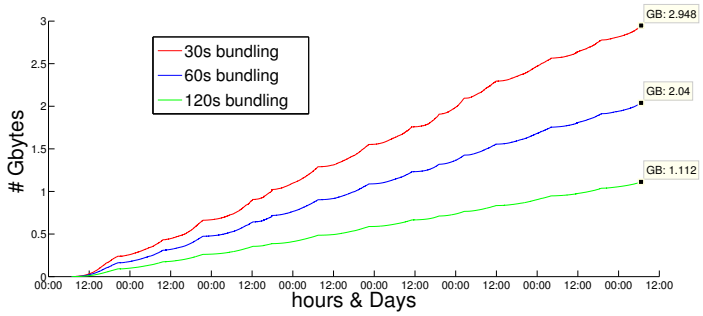


Figure 4.7: Data re-sent with Static chunking (512KB).

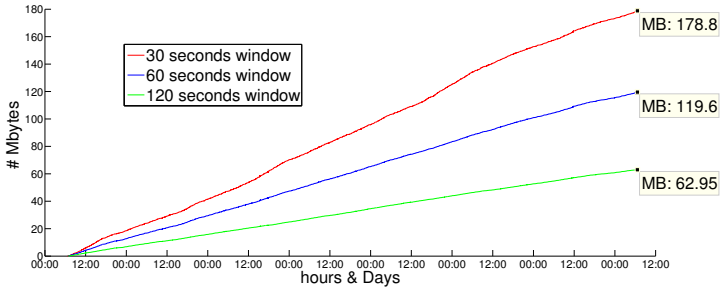


Figure 4.8: Data re-sent with Dynamic chunking (8KB-32KB).

Dynamic chunking on this experiment has a big impact on the amount of data re-sent to the cloud server. The quantity of data bundled on each W is so small that does not produce positive effects on the Gain. The number of devices to be synced must be very big to obtain positive values of Gain. For this reason, dynamic chunking is deprecated in this experiment, because even bundling on different W the amount of data does not reach a minimum necessary level to switch from HTTP to BitTorrent communication protocol.

When static chunking is used, a positive result is obtained with few devices, and this improvement tends to be better when the number of devices increase.

The amount of offloaded data percentage remains stable regardless the bundling window used. Even synchronizing 10 devices and relaxing τ we cannot see a communication improvement. A slight improvement can be seen when using 15 or more devices. This is shown in Tables 4.7 and 4.8, in which it appears a new column to fit new positive values while τ changes.

In Tables 4.6 , 4.7 and 4.8 can be seen that growing up the number of synchronized devices significantly improves the benefits of applying the BitTorrent Protocol, so amount of data synchronized is greater and percentage ratio of offloaded data too. That is, the communication system can scale well on a large group of synced devices with a relative small amount of short and fast updates. The amount of data to be synced on each individual modification is small to apply BitTorrent protocol, but applying a bundling window and static chunking this amount of data reach a level in which switching HTTP by BitTorrent is suitable for communication purposes.

	$Window = 120secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	25,6057	25,6057	-	-
Upload Total (inMBs)	88,9556	88,9556	-	-
Overall Offload %	28,78	28,78	-	-

	$Window = 60secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	46,9352	46,9352	-	-
Upload Total (inMBs)	162,6506	162,6506	-	-
Overall Offload %	28,86	28,86	-	-

	$Window = 30secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	67,4347	67,4347	-	-
Upload Total (inMBs)	235,0898	235,0898	-	-
Overall Offload %	28,68	28,68	-	-

Table 4.6: Offload on Static Chunking Log file (10 devices synced).

	$Window = 120secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	60,5653	60,5653	57,9811	-
Upload Total (inMBs)	133,4333	133,4333	133,4333	-
Overall Offload %	45,39	45,39	43,45	-

	$Window = 60secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	110,0735	110,0735	105,8914	-
Upload Total (inMBs)	243,9759	243,9759	243,9759	-
Overall Offload %	45,12	45,12	43,40	-

	$Window = 30secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	157,4474	157,4474	151,7737	-
Upload Total (inMBs)	352,6347	352,6347	352,6347	-
Overall Offload %	44,65	44,65	43,04	

Table 4.7: Offload on Static Chunking Log file (15 devices synced).

	$Window = 120secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	89,7263	89,7263	89,7263	10,8236
Upload Total (inMBs)	177,9111	177,9111	177,9111	177,9111
Overall Offload %	50,43	50,43	50,43	6,08

	$Window = 60secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	163,0718	163,0718	163,0718	20,1313
Upload Total (inMBs)	325,3012	325,3012	325,3012	325,3012
Overall Offload %	50,13	50,13	50,13	6,19

	$Window = 30secs$			
Gain Constraint τ	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	233,2555	233,2555	233,2555	-
Upload Total (inMBs)	470,1796	470,1796	470,1796	-
Overall Offload %	49,61	49,61	49,61	-

Table 4.8: Offload on Static Chunking Log file (20 devices synced).

4.2.3 Collaborative Editing experiment results.

In this experiment it is important to note that techniques like bundling mechanisms on different W do not affect the offloaded data percentage and this relation remains under a 1% for a given W and chunking technique. This can be observed in Tables 4.9, 4.10, 4.11 and 4.12.

The greater the amount of data to be synced, the greater positive impact the bundling techniques bring in. Savings can reach approximately a 24% on Kosovo file and 57% on Michael Jackson file depending on the W applied with static chunking.

If τ is between -1 and 0, the relation of offloaded data remains stable regardless the window size used. Then, with a positive value of τ the percentage decays abruptly, especially if static chunking is used. With dynamic chunking the percentage variantion can not be appreciated since the gain is not as high as in static chunking.

In Figures 4.9, 4.10, 4.11 and 4.12 can be seen the Gain and the Offload calculated for each W.

	$Window = 120secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	185,6701	185,6701	185,6701	185,6701	29,4339	16,3803
Upload Total (inMBs)	755,7457	755,7457	755,7457	755,7457	755,7457	755,7457
Overall Offload %	24,57	24,57	24,57	24,57	3,89	2,17
	$Window = 60secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	247,8590	247,8590	247,8590	247,8590	34,5451	16,3803
Upload Total (inMBs)	986,3450	986,3450	986,3450	986,3450	986,3450	986,3450
Overall Offload %	25,13	25,13	25,13	25,13	3,50	1,66
	$Window = 30secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	268,2375	268,2375	268,2375	268,2375	39,1951	16,3803
Upload Total (inMBs)	1084,0085	1084,0085	1084,0085	1084,0085	1084,0085	1084,0085
Overall Offload %	24,74	24,74	24,74	24,74	3,62	1,51

Table 4.9: Offload on Static Chunking Kosovo file varying Gain Constraint.

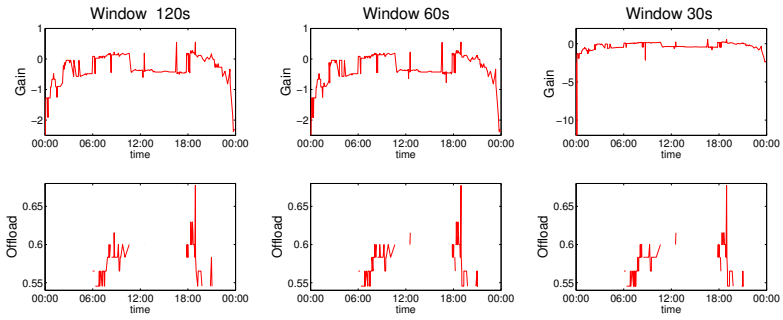


Figure 4.9: Gains & Offloads Static Chunking Kosovo file.

	$Window = 120secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	18,4964	18,4964	18,4964	18,4964	16,3803	16,3803
Upload Total (inMBs)	355,1809	355,1809	355,1809	355,1809	355,1809	355,1809
Overall Offload %	5,21	5,21	5,21	5,21	4,61	4,61

	$Window = 60secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	18,4964	18,4964	18,4964	18,4964	16,3803	16,3803
Upload Total (inMBs)	439,3491	439,3491	439,3491	439,3491	439,3491	439,3491
Overall Offload %	4,21	4,21	4,21	4,21	3,73	3,73

	$Window = 30secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	20,5437	20,5437	20,5437	20,5437	16,3803	16,3803
Upload Total (inMBs)	480,3158	480,3158	480,3158	480,3158	480,3158	480,3158
Overall Offload %	4,28	4,28	4,28	4,28	3,41	3,41

Table 4.10: Offload on Dynamic Chunking Kosovo file varying Gain Constraint.

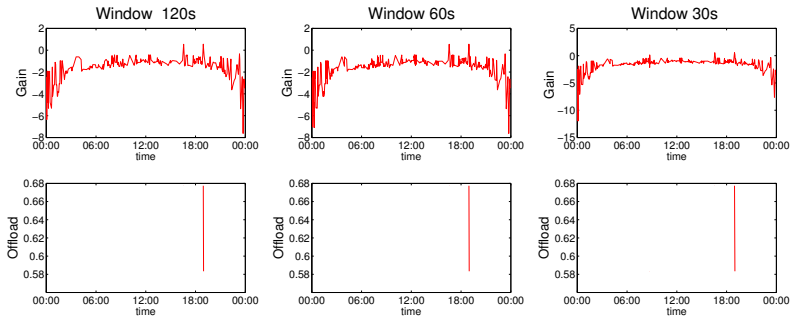


Figure 4.10: Gains & Offloads Dynamic Chunking Kosovo file.

	$Window = 120secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	1403,7939	1403,7939	1403,7939	1403,7939	1306,0388	139,4286
Upload Total (inMBs)	2444,2655	2444,2655	2444,2655	2444,2655	2444,2655	2444,2655
Overall Offload %	57,43	57,43	57,43	57,43	53,43	5,70

	$Window = 60secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	2037,1366	2037,1366	2037,1366	2037,1366	1899,2184	172,4507
Upload Total (inMBs)	3561,8627	3561,8627	3561,8627	3561,8627	3561,8627	3561,8627
Overall Offload %	57,19	57,19	57,19	57,19	53,32	4,84

	$Window = 30secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	2297,3560	2297,3560	2297,3560	2297,3560	2133,8456	191,3237
Upload Total (inMBs)	4026,2463	4026,2463	4026,2463	4026,2463	4026,2463	4026,2463
Overall Offload %	57,06	57,06	57,06	57,06	53,00	4,75

Table 4.11: Offload on Static Chunking Michael J. file varying Gain Constraint.

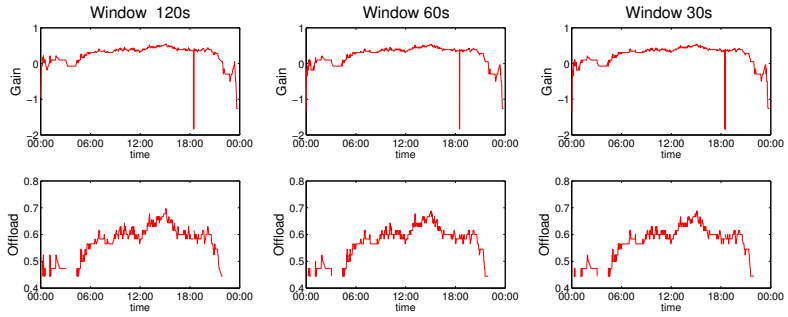


Figure 4.11: Gains & Offloads Static Chunking Michel J. file.

	$Window = 120secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	69,7128	69,7128	69,7128	69,7128	53,9592	5,9856
Upload Total (inMBs)	649,5081	649,5081	649,5081	649,5081	649,5081	649,5081
Overall Offload %	10,73	10,73	10,73	10,73	8,31	0,92

	$Window = 60secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	70,0314	70,0314	70,0314	70,0314	52,4913	5,9856
Upload Total (inMBs)	830,1583	830,1583	830,1583	830,1583	830,1583	830,1583
Overall Offload %	8,44	8,44	8,44	8,44	6,32	0,72

	$Window = 30secs$					
Gain Constraint τ	**-1**	**-0.5**	**-0.2**	**0**	**0.2**	**0.5**
Offloaded Data (inMBs)	76,4415	76,4415	76,4415	76,4415	58,3558	-
Upload Total (inMBs)	920,4727	920,4727	920,4727	920,4727	920,4727	-
Overall Offload %	8,30	8,30	8,30	8,30	6,34	-

Table 4.12: Offload on Dynamic Chunking Michael J. file varying Gain Constraint.

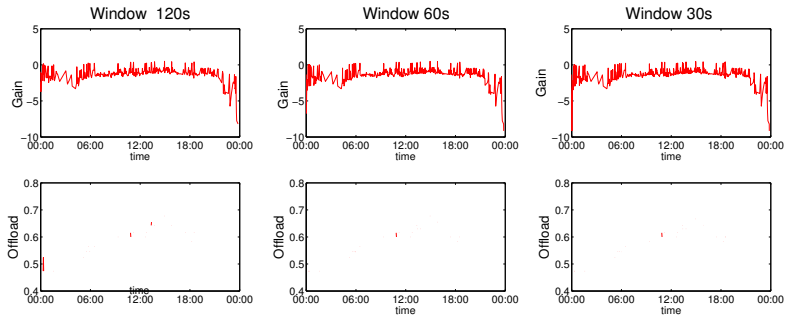


Figure 4.12: Gains & Offloads Dynamic Chunking Michel J. file.

Chapter 5

Conclusions

We have studied the effect of bundling and chunking techniques with the aim of taking advantage from the clients badwidth to reduce the cloud server workload.

To use BitTorrent the data to be synchronized must have a minimum size. Depending on the bundling technique the amount of data to be synchronized can be very big even when the files used in our experiments and the changes applied to them are small. These big bundles can justify the usage of BitTorrent.

It is important to note that techniques like bundling mechanisms on different time window sizes with the same chunking technique applied do not affect clearly offloaded data percentage. This relation remains stable in most of experiments done with small size files.

It is observed that the greater the amount the data to be synced the greater is the impact of bundling techniques. On static chunking this impact is greater than on dynamic because of the amount of data that must be transferred.

Dynamic chunking clearly reduces the amount of data re-sent to cloud servers in front of Static chunking, but implies a high use of CPU to create chunks on client side.

Static chunking is easy to compute on client side but sends a lot of repeated data depending on chunk size. This difference of amount of data repeated could be saved by communication protocol. A lot of data repeated to be sent implies to load the cloud servers with a huge amount of data, but if BitTorrent is used to offload communication process on the server side, this problem will be solved.

Bibliography

[1] Nitin Agrawal, William J Bolosky, John R Douceur, and Jacob R Lorch. A five-year study of file-system metadata. *ACM Transactions on Storage (TOS)*, 3(3):9, 2007.

[2] Liliana Ardissono, Anna Goy, Giovanna Petrone, and Marino Segnan. From service clouds to user-centric personal clouds. In *Cloud Computing, 2009. CLOUD'09. IEEE International Conference on*, pages 1–8. IEEE, 2009.

[3] Windows Azure. http://www.windowsazure.com/, Retrieved: August 2014.

[4] Rahma Chaabouni, Pedro Garcia-Lopez, Marc Sanchez-Artigas, Sandra Ferrer-Celma, and Carlos Cebrian. Boosting content delivery with bittorrent in online cloud storage services. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–2. IEEE, 2013.

[5] García-López P. Chaabouni R., Sànchez-Artigas M. Reducing costs in the personal cloud: Is bittorrent a better bet. 2014.

[6] BingChun Chang. A running time improvement for two thresholds two divisors algorithm. 2009.

[7] Mosharaf Chowdhury, Matei Zaharia, Justin Ma, Michael I Jordan, and Ion Stoica. Managing data transfers in computer clusters with orchestra. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 98–109. ACM, 2011.

[8] Bram Cohen. Incentives build robustness in bittorrent. In *Workshop on Economics of Peer-to-Peer systems*, volume 6, pages 68–72, 2003.

[9] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *Proceedings of the 2013 conference on Internet measurement conference*, pages 205–212. ACM, 2013.

[10] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *Proceedings of the 2012 ACM conference on Internet measurement conference*, pages 481–494. ACM, 2012.

[11] Google Drive. https://drive.google.com, Retrieved: August 2014.

[12] Dropbox. https://www.dropbox.com, Retrieved: August 2014.

[13] Kave Eshghi and Hsiu Khuern Tang. A framework for analyzing and improving content-based chunking algorithms. *Hewlett-Packard Labs Technical Report TR*, 30:2005, 2005.

[14] Gartner. http://www.gartner.com/technology/topics/cloud-computing.jsp, Retrieved: August 2014.

[15] Frank E Gillett, Christopher Mines, Ted Schadler, Michael Yamnitsky, Heidi Shey, Amelia Martland, and R Iqbals. The personal cloud: Transforming personal computing, mobile, and web markets. *Forrester Research, BT Futures Report*, 2011.

[16] Glauber Gonçalves, Idilio Drago, Ana Paula Couto da Silva, Alex Borges Vieira, and Jussara M Almeida. Modeling the dropbox client behavior. In *Proceedings of the International Conference on Communications, ICC*, volume 14, 2014.

[17] Adiseshu Hari, Ramesh Viswanathan, TV Lakshman, and YJ Chang. The personal cloud—design, architecture and matchmaking algorithms for resource management. In *Presented as part of the 2nd USENIX Workshop on Hot Topics in Management of Internet, Cloud, and Enterprise Networks and Services*. USENIX, 2012.

[18] Dropbox is now the data fabric typing together devices for 100M registered users who save 1B files a day. http://techcrunch.com/2012/11/13/dropbox-100-million, Retrieved: August 2014.

[19] Mikel Izal, Guillaume Urvoy-Keller, Ernst W Biersack, Pascal A Felber, Anwar Al Hamra, and Luis Garces-Erice. Dissecting bittorrent: Five months in a torrent's lifetime. In *Passive and Active Network Measurement*, pages 1–11. Springer, 2004.

[20] Shakir James and Patrick Crowley. Fast content distribution on datacenter networks. In *Proceedings of the 2011 ACM/IEEE Seventh Symposium on Architectures for Networking and Communications Systems*, pages 87–88. IEEE Computer Society, 2011.

[21] Shakir James and Patrick Crowley. Experimental analyses of data distribution on data center networks. In *Peer-to-Peer Computing (P2P), 2013 IEEE Thirteenth International Conference on*, pages 1–10. IEEE, 2013.

[22] Rakesh Kumar and Keith W Ross. Peer-assisted file distribution: The minimum distribution time. In *Hot Topics in Web Systems and Technologies, 2006. HOTWEB'06. 1st IEEE Workshop on*, pages 1–11. IEEE, 2006.

[23] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *Middleware 2013*, pages 307–327. Springer, 2013.

[24] Dongyu Qiu and Rayadurgam Srikant. Modeling and performance analysis of bittorrent-like peer-to-peer networks. In *ACM SIGCOMM Computer Communication Review*, volume 34, pages 367–378. ACM, 2004.

[25] Amazon S3. http://aws.amazon.com/s3/, Retrieved: August 2014.

[26] Google Cloud Storage. https://developers.google.com/storage/, Retrieved: August 2014.

[27] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *USENIX Annual Technical Conference*, pages 261–272, 2012.

[28] P. Windley. http://www.windley.com/archives/2012/04/from_personal_computers_to_personal_clouds.shtml, Retrieved: August 2014.