Department of Computer Engineering and Mathematics
Architecture and Telematic Services Research Group

M.Sc. Thesis Dissertation

Resorting to the Cloud for improving performance on F2F storage: FriendBox as an example

by

Adrián Moreno Martínez

Thesis Advisors:
Dr. Pedro García López
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 2012

Abstract

Personal storage is a mainstream service used by millions of users. Among the existing alternatives, Friend-to-Friend (F2F) systems are nowadays an interesting research topic aimed to leverage a secure and private off-site storage service.

However, the specific characteristics of F2F storage systems (reduced node degree, correlated availabilities) represent a serious obstacle to their performance. Actually, it is extremely difficult for a F2F system to guarantee an acceptable storage service quality in terms of transference times and data availability to end users. In this landscape, we propose resorting to the Cloud for improving the storage service of a F2F system.

We present `FriendBox`: a hybrid F2F personal storage system. `FriendBox` is the first F2F system that efficiently combines resources of trusted friends with Cloud storage for improving the service quality achievable by pure F2F systems.

We evaluated `FriendBox` through a real deployment in our university campus. We demonstrated that `FriendBox` achieves high transfer performance and flexible user-defined data availability guarantees. Furthermore, we analyzed the costs of `FriendBox` demonstrating its economic feasibility.

Acknowledgments

This thesis is the result of long 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 advisors Dr. Pedro García López and Dr. Marc Sánchez Artigas for their support and guidance during the development of this thesis. I would also like to thank them for giving me the freedom that I had regarding to many aspects of this work, contributing to my personal development.

Second, I would like to specially thank my colleague and fellow student at the AST research group Raúl Gracia Tinedo for his enormous help and being there in the tough moments. Without him, this thesis would have not been possible.

I also want to thank all the people that have worked with me at the AST group during this period. Guillermo Guerrero, Cristian Cotes, Aleix Ramirez and Edgar Zamora, thank you for your support and friendship.

For providing the infrastructure needed for the experiments, I want to thank David Manye. He reserved laboratories for us to be able to perform all experiments to validate this thesis.

Last but not least, a warm thank you to my family and my girlfriend for their love and their moral support. Thank you very much for always being there for me, Jordina Palau, Domi Martínez, Juan Moreno and Raúl Moreno.

Thank you very much!

Adrián.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Over the last few years, Cloud storage systems have increasingly gained importance and are currently a paramount research topic. Cloud storage providers like Amazon S3, Google Storage or Rackspace offer their storage services on a pay-as-you-go basis, allowing customers to consume the precise amount of resources they need. Such a novel business model presents the raw storage service as a utility, which enables 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 their 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.

Cloud services normally consist of a large number of servers linked together and hosted in data centers. Generally including backup power supplies, redundant data connection, environmental controls and security devices, witch results in high Quality of Service (QoS) in terms of data availability, that is, the probability that data is accessible by users in any situation. For instance, data availability can be achieved by placing different copies across the system, which is the simplest data redundancy technique. Therefore, if a node fails, data can still be retrieved from another source.

Although Cloud systems can meet the requirements of mass storage and data availability, some users are reluctant to move their data to the Cloud due to the large amount of control ceded to Cloud vendors, which includes the lack of data privacy [28] among other issues. Even so, some people are willing to trade the lack of trust towards the service provider in exchange for a more pleasant user experience. But if we go a little further in terms of very private data or business critical information, this trust completely disappears.

As an alternative, Friend-to-Friend (F2F) systems have been proposed recently to provide a private and secure personal storage service. In F2F systems, users do not rely neither in unknown peers nor in a centralized server, instead they benefit from social links to build a trusted storage system, enabling users to retain the control of their data. Consequently, users have the choice of storing their private information

only on nodes they trust. Moreover, the social component of F2F systems alleviates many undesirable problems present in large scale distributed systems —e.g. security, trust, incentives.

1.1 Problem statement

The main challenge that F2F systems experience is the poor QoS compared to pure Cloud storage services, both in terms of data availability and transfer times. That is because F2F systems implicitly suffer from two main drawbacks.

- **High availability correlations.** Indeed, measurements of online social networks have shown that friends present significant correlation in their activity patterns [18,33]. This implies that it is probable to find all friends of a user simultaneously offline —e.g. during night hours— which makes it impossible to maintain high data availability even when placing one replica at each friend. Moreover, such correlated patterns have an equally negative impact on transfer times. If a user were to upload one chunk of data to each of his friends, and most of them were offline due to correlation, the user would have to wait for those friends to come back online before completing all data transfers.

- **Extremely small friendsets.** Users in a F2F storage system are likely to have a reduced number of trustable friends. To wit, over 63% of Facebook users have less than 100 friends [44], and what is even worse, most of their interactions occur only across a reduced subset of their social links. Concretely, 20% of their friends account for 70% of all interactions [44]. In this line, other content distribution sites with social components —e.g. YouTube, Flickr, Friendster— present even lower connectivity among users.

Therefore, while Cloud storage services can almost completely guarantee data availability, achieving a similar level of data availability on F2F systems may require a high amount of data redundancy to mask the intermittent participation of friends.

1.2 Our proposal

In this landscape, we propose a radically different approach: resorting to Cloud storage services for improving the performance of F2F storage systems.

We propose `FriendBox`: a hybrid F2F personal storage system. `FriendBox` is the first F2F system that efficiently combines the resources of trusted friends with Cloud storage for improving storage service quality while preserving privacy. Using a coding scheme data is split into meaningless pieces and scattered through friends and the Cloud. In order to recover the original file, the owner must gather a set of these pieces and reconstruct the file. `FriendBox` also provides a flexible and user-defined Cloud usage: users are able to decide where to store their data, which can be completely at friends, only in the Cloud or both. Hence, users can set the level

of QoS they want to obtain according to two parameters: *the targeted level of data availability* and *the fraction of data to be permanently stored in the Cloud*. At one end lies a user who wants a high QoS, for which the amount of data to be stored at the Cloud is high. At the other end is a user who wants high control over his data, for which he keeps most of his data stored at friends, but at the cost of a lower storage service quality.

The solution devised in this thesis achieves the following goals:

- **Achieve better quality of service than pure F2F systems.** Due to the nature of pure F2F systems they inherit two characteristics that prevent them from providing an adequate service quality: *reduced node degree* and *availability correlations*. In `FriendBox`, we aim to find a solution for these problems and provide higher levels of data availability and transfer times.
- **User control of data.** One of the hot topics of Cloud computing refers to users' privacy of their personal information. User control of data is lost the same instant a user cedes their personal data to a company. On the contrary, in F2F systems users' data is split throughout a set of friends. Therefore, data privacy enforcement mechanism must be incorporated to ensure there a legit use of the system by all participants —i.e. there is no unauthorized access to other people's data.
- **User-defined cost vs. QoS trade-off.** The use of Cloud storage services like Amazon S3 imply storage and bandwidth costs to be faced by users. As our system will make use of Cloud services we aim to provide a mechanism to regulate Cloud usage in function of their monetary consumption. This way users will be sure that they will not pay more than expected. On the contrary, the fact of limiting Cloud usage will probably decrease data availability and, therefore, service quality.
- **Vendor Lock-in avoidance.** Since in Cloud storage services data is trusted to a single company users are dependent on their service and fate. For instance, if the company bankrupts or their data center is affected by a virus a user is likely to lose its data. We propose avoiding vendor lock-in problems by storing user data in different entities —i.e. throughout a set friends and the Cloud— so users are not tied up with a single entity. Eluding this major problem results in a more suitable and beneficial storage solution for end users.

1.3 Contributions of this Thesis

- **Analysis of the state-of-the-art.** Prior to designing our solution, we study the current approaches concerning Cloud and F2F systems which is reported further in this manuscript.
- **Design and implementation of `FriendBox`, a hybrid F2F personal storage application.** The main contribution of this thesis is `FriendBox`, an

application that combines the resources of trusted friends with Cloud storage systems to improve F2F service quality in terms of data availability and transfer times.

- **Evaluation of `FriendBox`.** In order to assess the main contribution of this thesis, we describe the environment as well as the key parameters and perform an experiment to shed some light on the feasibility of `FriendBox`. Eventually, we evaluate aspects such as transfer times, data availability or the monetary cost of taking advantage of the Cloud.

1.4 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 and F2F systems, analyzing different techniques and mechanisms regarding data privacy, redundancy, availability and privacy.

Chapter 3 focuses on the design and implementation of `FriendBox`, detailing the different steps that have led us to this particular solution. It also explains the motivation behind the software and introduces the `FriendBox` architecture. Furthermore, it illustrates the implications of the architecture, discussing data redundancy and placement techniques, as well as the services used. All in all, it is intended to provide a broader view of `FriendBox`.

In Chapter 4 we describe the environment used to perform our experiments as well as key parameters like the fraction of data to store in the Cloud or the parallel connections. In this context, we perform a feasibility study of `FriendBox` and evaluate key aspects such as transfer times, data availability or the monetary cost of taking advantage of the Cloud.

We conclude this thesis in Chapter 5, where we draw some conclusions and enumerate the achieved goals through our study. Finally, we propose future work which can be performed on the topic of this Master Thesis.

1.5 Publications

- Raúl Gracia Tinedo, Marc Sánchez Artigas, **Adrián Moreno Martínez**, Pedro García López. *"FriendBox: A Hybrid F2F Personal Storage Application"*. In the 5th IEEE International Conference on Cloud Computing. June 24th-29th, 2012, Honolulu, Hawaii, USA. Acceptance rate: 17%.

- **Adrián Moreno Martínez**, Raúl Gracia Tinedo, Marc Sánchez Artigas, Pedro García López. *"FriendBox: A Cloudified F2F Storage Application"*. In the 12th IEEE International Conference on Peer-to-Peer Computing. September 3rd-5th, 2012, Tarragona, Spain.

Chapter 2

Background and Related Work

In this section we aim to introduce the concept of storage system and its importance in the design of today's Internet applications. Moreover, we provide a comprehensive overview of current models of storage systems and some examples to better understand how they work. Subsequently, we compare the different characteristics of the mentioned systems. Finally, we also make allusion to related research and projects from the areas of personal storage and Friend-to-Friend systems.

2.1 Local storage systems

In a computer environment, storage is the place where data is held in an electromagnetic or optical form for access by a computer processor. Storage can be divided, in general terms, into two categories:

- *Primary storage*, which holds data in memory —i.e. random access memory or RAM— and other built-in devices such as the processor's cache.
- *Secondary storage*, which holds data on hard disks drives (HDD), tapes, flash memory and other devices requiring input/output operations.

Primary storage is much faster to access than secondary storage because of the proximity of the storage to the processor or because of the nature of the storage devices. On the other hand, secondary storage can hold much more data and for a longer time than primary storage.

Hard disks are designed to provide data persistently. This means that data always preserves the previous version of itself when it is modified. It outlives the process that created it. Without this capability, data would be lost every time the storage device loses power.

Persistent storage has been present in computers from a long time ago. Nowadays, every electronic device such as laptop computers or mobile phones come with some kind of persistent storage. But all of them suffer from a major problem: data is stored locally, and in consequence, several accidents such as hardware errors, theft or user misuse can lead to data losses. The traditional solution to overcome these deficiencies was to replicate the information into other external devices such as HDD. But these backup solution are difficult to manage for domestic users, so they generally opt for store it only on their local disk and face the risk of losing their data.

In order to partly deal with this problem, solutions like RAID aggregate independent disk drives to provide storage functions and reliability through redundancy. There exists different levels depending on the number of drives available and the level of fault tolerance required. The main advantage of RAID is that faulty disks can be easily replaced without losing any data.

However, RAID presents a set of deficiencies, in which we highlight their scalability limitations, allowing only configurations of a few hard disk drives. This limitation was the reason that led to the design of *distributed storage systems*, as means to provide a solution for scaling storage systems and meet ever-growing storage demands.

2.1.1 Network-attached storage

Network-attached storage (NAS) is a device specifically built to store and serve data. NAS devices are connected to a network and provide a central point allowing access to heterogeneous clients. In the last years, NAS devices have gained popularity as a convenient method for sharing and backing up data among multiple computers.

Unlike ordinary computers created to be of general purpose, NAS devices are specialized by their hardware and software. Depending on the model, multiple hard disk drives can be attached and function in a RAID configuration to provide data redundancy.

However, NAS devices present a set of drawbacks that limit their possibilities:

- *Network congestion.* Heavy use of NAS may cause congestion on the shared network and affect other users. Therefore, NAS solutions are not suitable for data transfer intensive applications.

- *Single point of failure.* The entire system depends on NAS reliability, if it fails, the system will not be accessible. Therefore, NAS cannot offer any storage service guarantees.

- *Limited upload bandwidth.* NAS provides fast access from the same local network, but retrieving large files from an external location can be an arduous task depending on the upload bandwidth at the NAS end.

2.2 Distributed storage systems

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.

2.2.1 Pure Peer-to-Peer storage systems

Peer-to-Peer (P2P) have increased popularity during in the past years. As seen in Figure 2.1 P2P systems aggregate resources from many nodes and makes them available among nodes, resulting in a decentralized management. This leads to a network that is easier and faster to setup and keep running because there is no need of system administrators to ensure efficiency and stability. As data is spread throughout the system, a failure of a node may not result in data loss because the rest of nodes may provide or reproduced the required data.

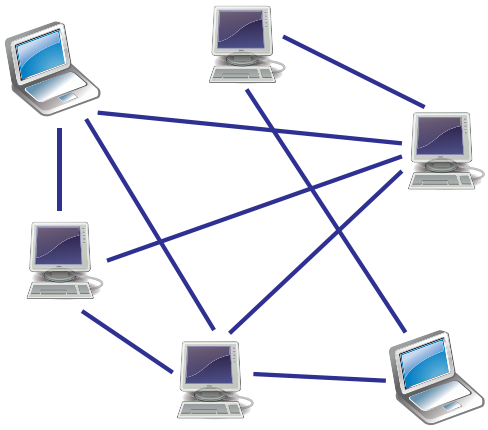

Figure 2.1: Example of a peer-to-peer system

In the same sense, when referring to storage systems, P2P is composed by storage nodes —i.e. peers— with equal privileges and liabilities. Each participant brings its own disk space to the system and makes it available to all nodes, so peers are both suppliers and consumers of resources without the need of a central server. Therefore, a large collaborative and distributed storage system is built.

However, besides the drawbacks of a distributed storage system, P2P needs to face another obstacle: as it is probable that storage nodes formed by personal computers and laptops, they have to handle constant arrivals, departures and failures of nodes. Thus, they have to design a reliable service over an unstable storage infrastructure. Furthermore, as P2P systems build an overlay network on top of the physical network topology, it is highly likely that neighbors in the P2P network are, in fact, nodes located in very distant places.

P2P storage systems have received a large amount of attention by researchers during the last years. Among all works performed we have selected the following two:

- *OceanStore* [23] was one of the first works to design a prototype of a P2P storage system. Basically, nodes in OceanStore are organized in a distributed hash table (DHT) overlay. When an object is inserted in the system it receives an unique ID and the object is sent to the node responsible for this ID. The responsible node for this ID stores one replica of the object and sends erasure encoded blocks to other nodes in the same DHT. Data repairs are performed

when node failures are detected. OceanStore produced a large number of contributions in the field of P2P storage systems.

- *PAST* [12] is as a large-scale persistent storage system for immutable data built on top of a DHT. PAST achieves data availability by simple object replication, and ensures durability by forcing each storage block to send heartbeat messages —i.e. periodic messages to identify if the originator continues to be available— to a node responsible to monitor data availability. However, PAST puts emphasis on guaranteeing load balancing among all participant nodes by storing replicas to nodes situated across the DHT's key-space.

2.2.2 Friend-to-Friend storage systems

Friend-to-Friend (F2F) systems emerged in the last years to overcome the limitations of P2P storage systems [22, 23]. Among these limitations P2P storage systems, the instability and heterogeneity of peers is an important issue that hinders the provision of an appropriate service quality [4]. Furthermore, despite important efforts [7], the existence of selfish behaviors (free-riding) and the lack of trust among participants make end-users reluctant to adopt P2P storage systems.

F2F systems can be considered as a specialized P2P network in which participants only make connections with people they trust. As an example, in Figure 2.2 we can observe a set of users linked to each other upon social network relationships. These real-life social relationships are exploited to guarantee a dependable system —e.g., a friend of mine will not erase my data—. Thus, data access is predominantly confined within a small social neighborhood. As friends are probably located close to each other and behave similarly, their connectivity patters will probably be correlated in time.

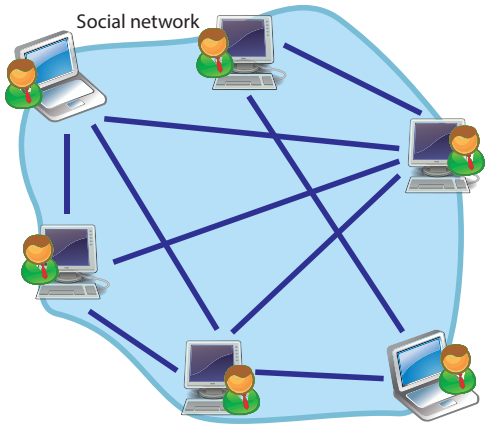

Figure 2.2: Example of a friend-to-friend system built upon social network relationships

However, F2F are unable to provide an acceptable level of data availability and transfer times due to they inherited characteristics: reduced node degree and availability correlations.

The first problem refers to the typically small number of available trusted friends to which store data. For instance, over 63% of Facebook users have less than 100 friends [44]. In this line, other content distribution sites with social components —e.g., YouTube, Flickr, Friendster— present even lower connectivity among users. The bad news are, however, that for most users the majority of interactions occur only across a small subset of their social links ——e.g., 20% of their friends account for 70% of all interactions [44]. This poses an important drawback: in a F2F system, users present extremely small friend-sets.

Second, real measurements on user behavior from online social networks suggest that friends are significantly correlated in their connectivity patterns [18,33]. This implies that is rather probable to find all friends of a user simultaneously offline, particularly during night hours, which makes it impossible to maintain high data availability even when placing one replica at each friend. Consequently, availability correlations degrade the storage service experienced by users in a F2F system.

Following we comment a F2F storage system approach called Friendstore.

- *Friendstore* [42] is a F2F backup system in which users only store their files in a subset of trusted peer nodes —i.e., their friends or colleagues— to discourage users from practicing free-riding. Users decide the amount of storage to be assigned to each node by real-world negotiations, called *storage contracts*. The amount of data each node can reliably store is calculated is based upon its upload bandwidth and the available disk space.

2.2.3 Peer-assisted storage systems

Although peer-to-peer (P2P) file sharing systems have received significant research attention, there exist a number of important differences between P2P file sharing and peer-assisted online storage systems. Peer-assisted online storage systems use peers' upload bandwidth in a complementary fashion in order to improve performance and file availability. Such systems typically employ dedicated servers to improve service quality. In Figure 2.3 we observe a typical hybrid architecture for peer-assisted systems: storage resources contributed by peers coexist with resources from dedicated servers.

P2P file sharing systems do not use servers to store actual file content, as all files are exchanged among users. As a result, they have no guarantees on file availability and files being downloaded may become unavailable at any time when all *seeds* —i.e. peers with a complete copy of the file— leave the system. Due to these differences, the design principles and objectives of peer-assisted online storage systems differ substantially from existing BitTorrent-like protocols.

The two main focuses of peer-assisted systems are *data placement* and *bandwidth allocation*. On one hand, data placement amounts to the problem of selecting the remote location that will store data objects. In many P2P storage systems, data fragments are randomly placed on remote peers using a DHT-based mechanism. In this layout, there is one more entity to consider —i.e., the centralized server—, and a

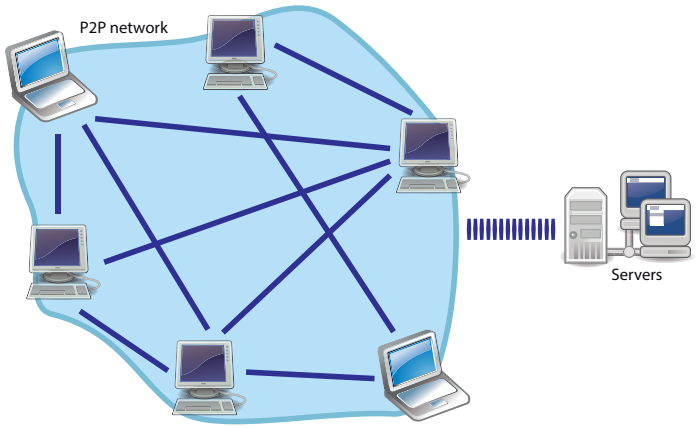

Figure 2.3: Example of a peer-assisted system

proper placement policy is crucial to allow users to recover their files at any time. On the other hand, since dedicated servers probably incur costs in terms of bandwidth and storage, to keep the economic cost at a minimum, a proper bandwidth allocation policy should use the server only in temporary and mandatory occasions.

In the recent years many researchers have proposed interesting peer-assisted solutions. Following, we discuss what we considered the most important ones:

- L. Toka [41] proposes an interesting peer-assisted architecture for back-up purposes. This work design an architecture considering three main points: availability, bandwidth allocation and data placement. Regarding data availability, authors just mention the limitations that unstable peers impose to the system in order to achieve a certain availability threshold. Bandwidth allocation refers to the algorithms employed to upload and download data to/from others, taking into account the data center as a particular and expensive resource. Finally, authors explore a couple of data placement policies in order to explore their impact on the system. This interesting study employs two metrics: transference times —i.e., restore and backup— and cost —i.e., the use of Cloud resources.

- R. Sweha [37] focuses on a peer-assisted content distribution system. Their main objective is to minimize the minimum distribution time metric, which actually means the time for a packet to reach all the members in the distribution group. The main reason is that this system is designed for bulk-synchronous applications, where a packet must be transmitted to every interested member before the next packet. The main contribution of this work is demonstrating that a set of *angels* —i.e., helper nodes in BitTorrent— powered by the Cloud reduces the minimum distribution time of a content.

2.2.4 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.

- *Geopraphic 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. Cloud storage systems typically implement RESTful APIs.

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** [32] (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** [34] 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.

- **Rackspace Cloud Files** [16] is product offered by Rackspace. It is powered by OpenStack, an open source, collaborative software project launched with contributions from Rackspace and the NASA. Rackspace's Cloud Files benefits from Akamai's content delivery network[1] (CDN) to deliver content rapidly through the globe. Akamai's CDN caches content at different locations around the world in order to save users time as the requested content is received from within the region instead of coming from the origin server. Pricing is based on the storage and outgoing bandwidth used. Unlike Amazon and Google, Rackspace does not charge for the incoming bandwidth nor the number of requests.

- **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

[1] A content delivery network (CDN) is a large distributed system of servers which allows faster, more efficient delivery of media files. Maintaining copies of data throughout the globe.

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.

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** [11] 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[2] 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.

- In the same line of Dropbox, **Box.com** [5] adopts a freemium model providing a free account of $5GB$ and paid accounts from $25GB$ up. As strengths, we highlight the feature of online edition of documents and an API to allow third-party applications to interact with their storage system. As weaknesses, Box.com free accounts have a file size limit of only $100MB$, which limits and may be not suitable for some users. Furthermore, it does not provide a desktop application to synchronize your local files to domestic users.

- Google has recently entered on the personal storage by introducing **Google Drive** [10], which basically is an upgrade of the former Google Docs. Google Drive works in much the same way as the previously mentioned services.

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

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.

2.2.5 Key concepts

- **Data availability.** The probability that a data continues to be accessible in situations ranging from normal through disastrous. Generally, increasing data availability is achieved through redundancy involving where the data is stored and how it can be reached. Data availability can be measured in terms of how often a data object is accessible —e.g. one Cloud vendor can promise $99,999\%$ availability.

- **Data placement.** In a distributed setting, where several nodes are involved in the system and data has to be scattered among them, there is a need to efficiently manage the amount of data that each node is responsible for. Therefore, the data placement process is in charge of managing storage resources and distributing data among nodes. Using an appropriate data placement strategy is crucial to keep the system well-balanced and avoid bottlenecks when users access to popular content at the same time.

- **Data redundancy.** In order to achieve high levels of data availability over a data object, the simplest strategy is to replicate this object and assign it to several nodes. Replication allows users to be able to retrieve the object in case there is at least one node keeping the object. Consequently, the storage space needed is the number of nodes storing the object times the size of the data. Therefore, providing a high availability may imply using numerous nodes, and that can become an issue in terms of disk space, transfer times and economic costs. Thus, an appropriate policy is of vital importance to achieve a proper performance.

- **Data durability.** Data durability is the probability of losing a data object after being stored for some time. To maintain data durability, the storage system should repair the objects lost when a node fails. This way the system remains in a consistent state. For instance, given a durability of $99,99\%$ of objects during a year, users can except losing a single object out of $10,000$. Of course, that is an expected average and does not guarantee the loss of $0,01\%$ of objects.

- **Data maintenance.** The data maintenance process continually monitors storage nodes and data integrity. When changes in nodes availability are detected —i.e. new nodes are added to the system or some nodes fail and become unresponsive— data has to be reallocated in order to ensure data

durability and keep resource consumption across nodes to a minimum. This feature also enables other important features like scalability to be guaranteed.

- **Scalability.** A system is said to be scalable when it is able to accommodate a significant growth in its amount of work and continue its normal functioning. Besides, the system's performance improves after adding hardware, proportionally to the capacity added. It is typically cheaper to add new storage nodes to a system than upgrade the hardware to each node in order to achieve an improvement in the system's performance.

- **Data location.** Depending on the storage system, data can be located in a wide variety of devices and infrastructures, from a local hard disk drive to a stack of racks in a data center. Since our data can be sensible or business critical it is of considerable importance knowing where is placed our data when using an storage system.

- **Retrieval time.** It is known as retrieval time the time elapsed from the instant a user gives the order to download a certain data object to the instant this user finally obtains it. Depending on the storage system, a data object can be stored in one or several storage nodes. Thus, the fact of retrieving an object may imply contacting many nodes. Users expect the time required to download data to be as short as possible. There are some factors that can alter retrieval times such as the physical location of the storage nodes, bandwidth and network congestion, and temporary unavailabilities.

2.2.6 Erasure codes

Although data replication is the simplest redundancy scheme, there exists other methods to achieve data redundancy such as using erasure codes. This codes are intended to reduce the storage and communication costs compared to replication in exchange for higher computational cost for coding and decoding.

The exponential growth in network bandwidth, storage capacity and computational power in today's personal computers has inspired designers of storage systems to propose achieving high availability and long-term durability by using erasure coding techniques. Although traditional replication increases data availability and durability, it introduces important challenges to system designers. The number of replicas must be increased to achieve high availability for wide-scale systems and, therefore, bandwidth and storage requirements increase as well.

As depicted in Figure 2.4, an erasure code splits a data object into k blocks and encodes them into n redundant blocks, where $n \geq k$. Calling $r = \frac{k}{n} < 1$ the rate of encoding. A rate r code increases the storage cost by a factor of $\frac{1}{r}$. These redundant blocks are such that any k out of the n blocks are sufficient to reconstruct the original data. For example, a $r = \frac{1}{3}$ encoding on a block divides the block into $k = 8$ fragments and encodes the original k fragments into $n = 24$ fragments. Which increases the storage cost by a factor of *three*.

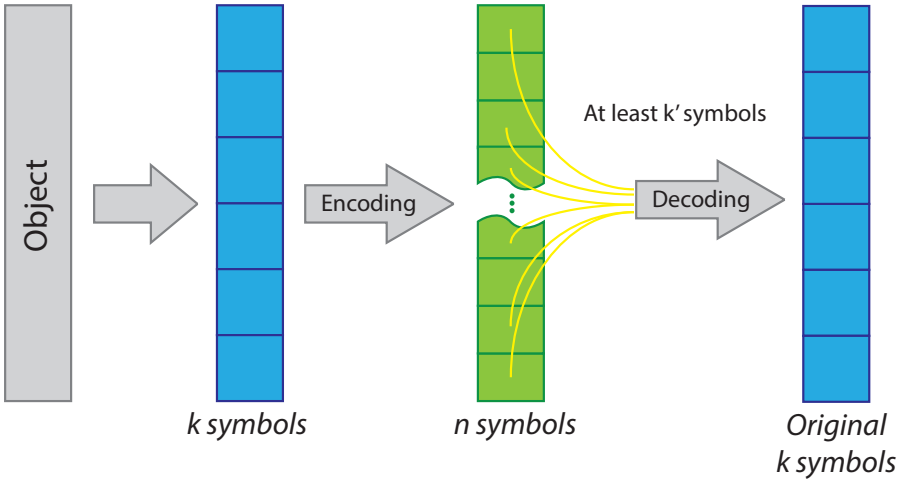

Figure 2.4: Generic erasure code scheme

Imagine a storage system composed by 1,000 nodes, ten percent of which are currently down. First, we analyze data availability over an object using a replication technique. Simply storing two replicas of the object provides 99% availability, because the probability of these two objects being in an offline node is $0.1^2 = 0.01$. Now, if we use an erasure code rate of $r = \frac{1}{2}$, which means that we are using twice the size of the original object like in the replication example, and encode the object into 18 fragments. We obtain a data availability of 99.9999999%, because the probability that more than nine blocks fall into offline nodes is $0.1^{10} = 0.0000000001$. Therefore, using the same amount of storage and bandwidth erasure codes can obtain better performance in terms of data availability and durability. Obtaining the same performance using replication would cause too much overhead. In Table 2.1 we can observe how replication and erasure codes techniques differ in some important terms.

Redundancy technique	Replicated r times	Erasure coded (k of n)
Faults tolerated	$r-1$	$n-k$
Probability of failure	f^r	$\sum_{j=1}^{k} \binom{n}{n-k+j} f^{n-k+j}(1-f)^{k-j}$
Storage efficiency	$1/r$	k/n
Access	find any one replica	find any k blocks

Table 2.1: Replication and erasure codes comparison

An implementation of erasure codes used in real-world systems are *Reed-Solomon codes* [31], which were created by Irving S. Reed and Gustave Solomon in 1960 as a forward error correction (FEC) method for sending data over an unreliable channel which may fail to transfer, or erase, some data. In fact, they have been around for decades. Reed Solomon codes are used in a wide variety of commercial applications such as *CDs*, *DVDs*, and in data transmission technologies like DVB and WiMAX. Recently, some enterprises offering storage services such as Cleversafe use Reed-

Solomon math to calculate the Error-Correcting codes (ECC). Reed-Solomon codes that do not specify a block size or a specific number of check symbols, instead these variables can be set to different values depending on the transmission medium or system characteristics.

2.3 Comparing storage systems

Before comparing the storage systems mentioned in the previous section, we will provide a definition of the main aspects we want to analyze. Some of them have already been introduced, but a brief reminder will be provided next.

Each storage system implement a distinctive **architecture**. The architecture of a system is composed by the number of nodes participating, type of nodes —e.g. domestic computers, dedicated servers—, their connection or the network topology. The architecture of a system is of vital importance and will determine their performance.

Another important point when analyzing a storage system is its **scalability**. A system is said to be scalable when it is able to accommodate a significant growth in its amount of work and continue its normal functioning. Furthermore, the system's performance improves after adding new nodes proportionally to the resources added.

Since some storage systems incur expenses for the use of bandwidth, storage or other services they apply a **pricing policy**. A proper pricing policy is essential to attract new users. For instance, a usage-based pricing policy charges the customer in function to the consumed resources. Another commonly extended pricing policy in the information technology is the *freemium* model, which consists in offering a basic product or service free of charge and other premium services that include advance features at a certain cost.

A key feature of any storage system is its **storage capacity**. Storage capacity refers to how much data is provided to the user. For instance, in a personal computer storage capacity can be increased by adding another internal hard drive. In a distributed storage system, the storage capacity available for users depend on the storage capacity contributed by nodes and, sometimes, on the user's collaboration in the system.

Users in a storage system must be able to retrieve their file at any time, Hence, the probability that a data continues to be accessible in situations ranging from normal through disastrous is called **data availability**. Generally, increasing data availability is achieved through redundancy involving where the data is stored and how it can be reached. Data availability can be measured in terms of how often a data object is accessible —e.g. one vendor can promise $99,999\%$ availability during one year.

In order to achieve high levels of data availability over a data object, the simplest **data redundancy** strategy is to replicate this object and assign it to several nodes. Replication allows users to be able to retrieve the object in case there is at least one node keeping the object. Consequently, the storage space needed is the number

of nodes storing the object times the size of the data. Therefore, providing a high availability may imply using numerous nodes, and that can become an issue in terms of disk space, transfer times and economic costs. Thus, an appropriate policy is crucial to achieve a proper performance.

Depending on the storage system, data can be located in a wide variety of devices and infrastructures, from a local hard disk drive to a stack of racks in a data center. Since our data can be sensible or business critical it is of considerable importance being aware of **data location** when using an storage system.

In order to access to data from an external software, storage system must implement an application programming interface (API). Providing a **public API** allows developers to integrate their application on top of the storage system. When used in the web environment, an API is typically defined as a set of HTTP request messages and XML or JSON response messages.

Since user data can be stored in alien nodes in which users do not have access, **file encryption** is essential to preserve their privacy and prevent unauthorized use or reproduction of data. When referring to data encryption in a storage system, there exists two techniques: client-side and server-side encryption. In the first technique data is encrypted before it is transmitted to a server. Usually, encryption is performed with a key that is unknown to the server. Therefore, the service provider is unable to decrypt the data object. In server-side encryption, users delegate their file protection to the service provider. Hence, encryption keys reside on the server machine. If it is allowed by the storage system, both techniques can work simultaneously.

In a distributed system, where several storage nodes are involved and data is divided among them, there is a need to efficiently manage the amount of data that each node is responsible for. Thus, a **load balancing** policy helps distribute workload across nodes to achieve optimal resource utilization and avoid bottlenecks when users access to popular content at the same time.

Some storage services provide tools to keep local and remote files identical to each other. This is called **file synchronization**. If any change occurs in the local file in gets copied to one or more remote locations, and vice versa. In order to support synchronization some type of software must be installed in the client end to monitor and compare local and remote data and keep them consistent.

When storing files sometimes users need to restore previous versions of files that have been modified. Therefore, it is needed to keep a historic of files. There exists different methods to implement **file versioning** in a storage system. Some simply save every change that is been made in the file, others save a copy after a number of changes or take periodic snapshots. File versioning can be implemented either on client or server side.

After providing a definition of the main characteristics of a storage system, we classify them in the Table 2.2 in order to analyze and compare features of the storage systems previously mentioned.

Storage system	NAS	P2P	F2F	Peer-assisted	Cloud	Personal Cloud
Architecture	Device connected to a local network	Overlay network at application layer on top of a physical network formed by nodes	P2P architecture where all peers are trusted friends among them	P2P architecture with assistance of a dedicated central server	Data center architecture composed by specialized devices	Run their added-value services on top of Cloud provider's services
Scalability	No. New HDD's must be purchased	Yes. New nodes can be easily incorporated to the system	Limited by the number of friends	Limited by server's scalability	Yes. According to Cloud providers	Yes. According to Cloud providers
Pricing	One-time payment for acquiring the device	None	None	Depending on server's policy	Billed on a usage basis	Typically Freemium model
Storage capacity	Depending on HDD capacity	Low	Very low	Medium	Unlimited	Variable
Data redundancy	RAID possible depending on the number of HDD	Storing data at nodes	Storing data at nodes	Storing data at nodes and server	Copies stored on different nodes in the same data center and in other data centers	Inherited from Cloud providers
Data location	In-situ. Stored in the device	Typically personal computers	Typically personal computers	Typically personal computers and a dedicated server	Specialized hardware in data centers	Specialized hardware in data centers
Data availability	Depending on local infrastructure (hardware, connection)	Medium (data redundancy trade-off)	Low (small peer set, availability patterns)	High (due to server assistance)	High (claimed by Cloud vendors)	High (inherited from Cloud providers)
Public API	Yes. Depending on the software installed in the device	No	No	No	Yes	Typically yes
File encryption	Yes. Depending on the software installed in the device	Client-side	Client-side	Client-side	Client-side and server-side	Client-side and server-side
Load balancing	No	Load is split among peers (e.g. DHT's consistent hashing)	Load is split among friends (e.g. placement policies)	Load is split among peers and server	Implemented in data centers	Implemented in data centers
Synchronization	Possible (depending on software)	No	No	No	No	Yes
Versioning	Generally no	No	No	No	Client-side possible	Client-side and server-side

Table 2.2: Comparison of storage solutions

2.4 Related work

This section is intended to survey existing F2F and peer-assisted systems and discuss the major differences between these systems and our approach intended to improve quality of service of pure F2F systems.

2.4.1 Peer-to-peer

P2P philosophy expect users to contribute to the system by sharing their resources in return for using other peers' resources. But, in real-world scenarios an important fraction of peers are reluctant to share resources [1]. Thus, P2P systems' main objectives —i.e cooperation and resource contribution between peers— gets altered leading to a situation called *free-riding*. Hence, we could roughly define a *free rider* as a peer that uses P2P network services but does not contribute to the system.

In [7], authors claim that users are unlikely to consume resources in proportion to their contribution unless they are enforced by some mechanism. *Samsara* intends to force honesty among peers by setting an agreement. Each peer that requests storage must reserve the same capacity for the requester peer. After an exchange, each partner checks the other to ensure faithfulness. In case the peer acted unfaithfully the system applies a probabilistic punishment.

Glacier [21] is a hybrid storage system that combines replication and erasure codes to provide a P2P backup system. It uses a primary storage layer that holds full replicas of each stored object and serves them to processes and users aiming to access the content. The secondary storage layer holds encoded blocks of the data stored in the primary layer to increase availability at a lower cost. With this 2-layer infrastructure, authors aim to guarantee efficient access to stored data and low storage overheads.

In [2], authors provide file availability and reliability through randomized replicated storage. Farsite is a distributed file system designed by Microsoft with the aim to integrate idle storage resources of their desktop PCs within the company. Data availability is guaranteed by replication although erasure codes are mentioned by the authors as a possible alternative.

However, despite these important efforts, the existence of *selfish behaviors* and the *lack of trust* among participants make end users reluctant to adopt P2P storage systems.

2.4.2 Friend-to-Friend

F2F systems emerged in the last years to overcome the limitations of P2P storage systems [22, 23]. Among the limitations of P2P storage systems, the *instability and heterogeneity* of peers is an important issue that hinders the provision of an appropriate service quality [4].

Storing data reliably in traditional P2P systems in terms of availability is a difficult task because nodes frequently fail and eventually data objects are lost and become unrecoverable. Thus, storing large amounts of data durably results in an

expensive or even impossible job. But, since users' bandwidth is unlikely to be increased in the near future, node failure rate has to be reduced to increase data durability and prevent data loss. In this sense, in [24], authors state that storing data reliably in P2P systems using nodes that have little or no incentives to remain in the system is an arduous task. Authors propose using neighbors based on real social relationships instead of randomly chosen peers. This would provide incentives for peers to cooperate and, at the same time, improve the stability of existing systems and reduce data maintenance.

Similarly, in [42] authors state that there is a need for users to seek off-site redundancy solutions to store their important data so it can be guaranteed in case of natural disaster or user misuses. However, P2P storage systems' present two main drawbacks: the previously mentioned availability problem and the possible denial of restore services when needed. Due to these reasons, authors present FriendStore, a cooperative backup system where each node use a subset of peers in which they trust —i.e. their friends and colleagues—. FriendStore aims to solve both the availability and denial-of-service problems thanks to trusted relationships. Users enter storage contracts only limited with their friends via real-world negotiations. Contracts are reliable because social relationships are at stake, they do not believe that friends-of-friends can enforce such contracts reliably. They validated their work developing a cooperative backup system.

A recent work in this field is [33]. Their empirical study of availability in F2F storage systems present a major drawback regarding data availability: if no friends are online, then data stored in the system will not be accessible. Whereas backup systems may be feasible using friends, storage systems require data availability that cannot be ensured with friends' resources. They explore the trade-offs between redundancy (how many copies of a data have to be scattered among nodes), data placement (where to store these data) and data availability (the probability of finding accessible data). They also show that the problem of obtaining maximal availability while minimizing redundancy is NP-complete. Furthermore, they perform a study based on real social network traces and examine the data redundancy needed to achieve a certain level of data availability. Concluding that nodes with as few as ten friends can obtain acceptable data availability ratios.

Social networking has become an everyday part of many peoples live –i.e. Facebook currently has more than 900 million active users as of March 2012[3]. Social networks provide an interesting platform to facilitate communication and sharing among users. And beyond that, there exists a multitude of integrated applications and even some web pages and organization make use of Facebook's credential instead of their own credentials to grant access to their systems. In [6], authors propose using inherent trust relationships between friends in a Social Network as a foundation for resource sharing. They present a Facebook application that aims to create a Social Cloud, enabling friends to share resources within the context of a Social network. Their prototype application is a marketplace where friends trade their

[3] http://newsroom.fb.com/content/default.aspx?NewsAreaId=22

resources using auctions and bidding mechanisms mediated by contracts. However, as stated previously, the availability of these resources cannot be guaranteed by the sole usage of friends.

In [9], authors present a Privacy-Preserving online Social Network called Safebook that leverages the trusted relationships of friends. Whereas in the previous works the major motivation for using friends was the increased stability, in this case the motivation is privacy and security. They suggest an approach to tackle privacy protection by adopting a decentralized architecture relying on the cooperation among a number of independent parties that essentially are the users constituting the on-line social network application. As previous works, Safebook also exploits trust relationships of users that are part of a social network. But in this case, for building privacy-preserving mechanisms. Again, the major limitation of this approach is that it relies solely on friends so that the system cannot ensure proper levels of data availability.

2.4.3 Peer-assisted storage

Another related line of research is peer-assisted [36] storage systems that combine Cloud and peer resources. When focusing on recent data backup solutions we cannot avoid thinking about Cloud storage systems —e.g. Dropbox, Google Drive— which, among other features, transparently sync your local files with remote ones residing at a data center. Despite their undeniable success, these applications have some negative points. The backed up data is trusted to a single company, bringing problems of data privacy and risks of data loss. Furthermore, since this company may be using a proprietary software, users are dependent on this vendor and unable to use other vendors without substantial costs, causing *vendor lock-in*.

In [41], authors present a hybrid architecture where resources contributed by peers (spare bandwidth, storage space) are complemented with temporal usage of Cloud storage services. They mainly focus on *data placement* and *bandwidth allocation* and study their impact on performance measured by the time required to complete a backup and a restore operation and the end users' costs. They demonstrated that, by using adequate bandwidth allocation policies in which storage space at a Cloud provider is only used temporarily, a peer-assisted backup application can achieve performance comparable to traditional client-server architectures but at a fraction of their costs.

As online storage systems become increasingly popular, server bandwidth costs have become prohibitively expensive, as files are hosted in either content distribution networks or dedicated large data centers. This bandwidth costs from server-based architectures have made it necessary for all online storage systems that remain free of charge to impose certain restrictions, including download bandwidth quotas per day, file size limitations, as well as maximum file online available time. In this line, in [35] authors present a large-scale online storage system with peer-based assistance and semi-persistent file availability that was developed to reduce server bandwidth costs. Their measurement study demonstrated the feasibility of combining peers

with Cloud resources and obtained good results while dramatically reducing Cloud costs. They claim that FS2You has quickly become one of the most popular online storage systems in China.

To conclude, `FriendBox` is, to the best of our knowledge, the first hybrid F2F solution that efficiently combines resources from trusted friends and Cloud services to provide a flexible, trusted and private personal storage service.

Chapter 3

FriendBox: A Hybrid F2F Personal Storage Application

This chapter introduces the file backup software `FriendBox`*. It describes the motivation behind the software, its design goals as well as its architecture. The main purpose of this chapter is to create a basic understanding of the system structure and process flows.*

3.1 Motivation

In the last few years the Cloud Computing market has grown tremendously and new services are emerging steadily. One of these services is Cloud storage, which has rapidly developed to meet the increasing demand for people to handle their digital lives, including photos, media files, and work documents. The number of Cloud offerings is large and growing, ranging from Dropbox and the likes, to IaaS providers like Amazon.

Although Cloud systems can meet the requirements of mass storage, some users are reluctant to move data to the Cloud due to the large amount of control ceded to Cloud vendors, which includes the lack of data privacy [28] among other issues.

In this sense, social storage, also known as Friend-to-Friend (F2F) storage, has emerged as an attractive alternative to Cloud storage systems, as it places data control in the hands of users by which it significantly improves privacy and security [6, 42]. F2F storage also includes commercial storage applications like CrashPlan [8], which allow to back up data to friends.

In pure F2F storage systems, users have the choice to store their private information only on nodes they trust, and as such, they are decentralized in nature. However, their decentralized operation poses unique challenges that are actually hindering their adoption. The main one is the poor QoS offered to users compared with Cloud storage services, both in terms of data availability and transfer times. While Cloud storage services can offer multiple nines of data availability, a similar level of data availability may require a high amount of redundancy to be added to friends to mask their discontinuous participation.

And even, it may be impossible to ensure it due to availability correlations. Indeed, measurements of online social networks have shown that friends present significant correlation in their activity patterns [18, 33]. This implies that it is not improbable to find all friends of a user simultaneously offline, for instance, during

night hours, which makes it impossible to maintain high data availability even when placing one replica at each friend.

Moreover, such correlated patterns have an equally negative impact on transfer times. If a user were to upload one chunk of data to each of his friends, and most of them were offline due to correlation, the user would have to wait for those friends to come back online before completing all the data transfers. These negative effects are even aggravated when the number of trustable friends is reduced, which is very common in social storage applications. To wit, over 63% of Facebook users have less than 100 friends [44], and what is even worse, most of their interactions occur only across a reduced subset of their social links. Concretely, 20% of their friends account for 70% of all interactions [44].

Although this thesis strongly relies on `FriendBox`, it does not focus on the application itself, but rather on optimizing service quality of F2F systems by making use of Cloud storage solutions. This thesis uses `FriendBox` as a test bed for the experiments and aims to optimize data placement and bandwidth consumption in a real-world scenario. For these experiments, `FriendBox` provides an optimal environment and offers the potential for field tests in future research.

3.2 Characteristics

In this section we describe the required functionalities of `FriendBox` in depth. Many of these requirement can also be found in related software. But, as far as we know, `FriendBox` is the only one to fulfill all of them at the same time.

- **File backup.** The key purpose of this application is to backup certain files stored locally into your friends and any kind of Cloud storage service. Hence, as a backup system, a user must be able to store and retrieve files at any time. This requirement is satisfied thanks to the use of Cloud storage services. Even if all your friends are off-line you will be able to retrieve your files. The amount of both transferred and remotely stored data should be reduced to a minimum.

- **Cloud independence.** Currently, `FriendBox` uses Amazon S3 as default Cloud storage service. We also provide a simple application that implements basic data operations (PUT, GET, DELETE) to test it with your own storage. In addition, new solutions can be easily implemented by fulfilling a simple interface. Examples for possible storage types include FTP, WebDAV or personal Cloud solutions like Dropbox.

- **Privacy.** Unlike other storage solutions, data is not trusted to a single company nor to unknown peers. Instead, data is split into pieces and scattered across trusted friends and the Cloud. These pieces are meaningless to storage nodes since they provide no relevant information about the original file or its owner. Furthermore, many pieces are required to reconstruct the original file, preventing illicit acts from malicious users.

- **Ease-of-use.** Despite the complexity behind the system, both Facebook's and desktop user front-ends are designed to provide a pleasant experience by implementing an intuitive interface for end-users. While it should not stand in the way of new functionalities, ease-of-use is a crucial element for the acceptance of the software.

- **Multi-platform.** Since desktop application is implemented in Erlang and Java, few changes were needed to adapt `FriendBox` to function either in Windows and Linux. It only requires the installation of the respective virtual machines from user side.

- **Openness.** The system is designed in a modular basis, so it can easily extend its functionalities by adding or replacing new modules. This technique allows upgrading certain aspects of the system without interfering in other parts. Moreover, the application is made public so users can freely download it.

3.3 Design choices

To improve the performance of F2F storage, we propose `FriendBox`, a hybrid architecture that utilizes the higher availability of Cloud storage services. As pure F2F systems, `FriendBox` nodes use their social links to set up symmetric storage relationships among them. Furthermore, each node is intended to use his preferred Cloud storage service as a storage node. Our hybrid model does not restrict the number of Cloud providers, thereby avoiding the vendor lock-in problem.

Cloud services improve the storage QoS of nodes in a twofold manner:

1. When friends exhibit poor availabilities, Cloud storage is used to store a fraction of the data to assure the targeted data availability.

2. As a temporary buffer to store blocks assigned to offline friends until they become online again. The idea of using the Cloud as a temporary buffer is to shorten the transfer times by letting `FriendBox` users to upload blocks to the Cloud instead of waiting for the disconnected friends to come back online again.

If the Cloud is used as a temporary buffer, then friends themselves are responsible for downloading the missing blocks from the Cloud. Eventually, the data owner removes the extra blocks from the Cloud upon a valid notification.

Because `FriendBox` is distributed in nature, we developed the system core in Erlang, a programming language devised to build massively scalable soft real-time systems with requirements on high availability. Some of its uses can be found in telecommunications, banking, e-commerce, computer telephony and instant messaging. Erlang's runtime system has built-in support for concurrency, distribution and fault tolerance.

Erlang is, in essence, a functional programming language, but it puts a large emphasis on concurrency and reliability. To be able to support hundreds of tasks running at the same time, Erlang uses the actor model, where each actor is a separate process in the virtual machine. Additionally, inter-process communication is provided in a asynchronous message passing fashion, instead of shared variables. That is, individual processes send messages among them to coordinate their execution, avoiding traditional locking synchronization mechanisms.

Communication between `Friendbox` clients and the application state server is made through REST API calls. We have opted for REST instead of other methods because of its simplicity against other technologies like SOAP in terms of organizing system resources. Furthermore, it runs over the standard HTTP protocol, which facilitates its inclusion and bypasses any intermediate firewall. Another feature that fits in our design is that REST calls are stateless, i.e., each request is treated as an independent transaction that is unrelated to any previous requests.

Traditionally, F2F systems were based on real-contact relationships. Instead, our system benefits from the current success of social networks to provision social relationships. Concretely, we have opted for Facebook since its the most spread social network with more than 900 million monthly active users as of June 2012 [14]. Moreover, it permits integration of third-party application into their social network, which serves as a entry point to our storage system.

As our system design requires from a Cloud service for assistance, we opted for implementing compatibility with Amazon S3 instead of other Cloud providers due to its acceptability across the community and was being used by many successful applications such as Dropbox or Tumblr. Nevertheless, thanks to our modular design, compatibility with other Cloud providers can be easily incorporated.

3.4 Architecture

`FriendBox` is split into three components with different responsibilities: Social Front-end, Storage Client and Application State. Each component accomplishes a particular functionality and interacts with the rest of components in the system.

The overall structure of `FriendBox` is as follows: the entry point of the system is the Social Front-end, which is a Facebook web site that serves the purposes of social relationships provisioning, access control and system information. In addition, the Storage Client, which is a desktop application, can be considered as the core of the system. In general terms, users are able to upload and retrieve files to the system. And last but not least, the two previous components communicate with the Application State server, that acts as system back-end and is in charge of data management and keeps the system in a consistent state.

In Figure 3.1 we illustrate how a user maintains storage links with some of his friends in a social network. Moreover, this user is able to store a fraction of his data in a Cloud storage service. The state information of a user's data is stored in the `FriendBox` Application State. Finally, users manage their storage relationships and

check the state of their storage service in the FriendBox Social Front-end.

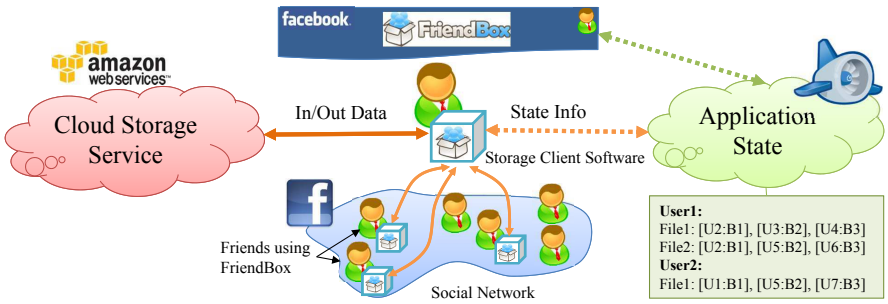

Figure 3.1: Overview of FriendBox's architecture.

3.4.1 FriendBox Social Front-end

The FriendBox Social Front-end provides three main services: social relationships provisioning, access control and information service via a web site.

FriendBox has been coupled with a social network with the main objective to provision social relationships. We have opted for Facebook since it is nowadays the most popular social network. Our Facebook website is the entry point of FriendBox: only those users which are members of the social network are capable of accessing to our F2F system. This fact simplifies the task of user management and access control by partially delegating them to the social network, which are known to be complex tasks in massive distributed applications.

The FriendBox Social Front-end provides a very intuitive GUI to expose the most relevant service information to the user. Concretely, users are able to obtain the next information:

- *Distribution of user's data within the system.* Since data is spread across friends', we considered appropriate that users knew whose friends are contributing to store their data and the degree of collaboration.
- *Accountability of others users' data.* Similarly, we considered interesting the fact of being aware of whose friends are you helping by storing their data and in which degree.
- *Monthly Cloud resources consumption.* Besides friends, data is also stored in the Cloud. Therefore, there is a chart displaying the monthly resources consumption in terms of bandwidth and storage.
- *Location of friends.* In case user's friends disclose their location, the user will be able to pinpoint them in an interactive map.
- *Listing of friends participation.* There is also a list enumerating the friends that are using the application and those that are already collaborating with the user.

- *Invitation system.* To attract new users to the system we implemented an invitation system that allows users to suggest the application to his Facebook friends. Once their friends incorporate to the system, the user can send them requests to start collaborating.

- *Listing of backed up files.* Users are able to control what files are backed up in the system at any time.

- *Configuration of personal and system parameters.* Through the control panel, users can inspect and update their personal and system information such as passwords and Cloud costs.

3.4.2 FriendBox Storage Client

Users must download the `FriendBox` Storage Client in order to connect to the system. This software enables users to perform basic data operations, such as storing and retrieving files from the system (which can be in friends and/or in the Cloud). Moreover, this software acts both as a client and a server in that our friends can store their information in the storage space we contribute to the system.

To cope with friends' instability, data redundancy must be added to the system to guarantee an acceptable level of data availability. To this end, a core functionality of the `FriendBox` Storage Client is the generation of data redundancy before inserting a file into the system. We armed this client with various redundancy mechanisms as we describe in Section 3.6.1.

One of the main novelties of `FriendBox` is to resort to Cloud storage services for improving the service quality achieved by pure decentralized F2F systems. The `FriendBox` Storage Client enables users to decide the distribution of data between their friends and the Cloud. This leaves to the user the control of his data and its placement within the system. In this line, the architecture of `FriendBox` supports simultaneous connections with several Cloud providers. This attractive feature opens the door for the development of new strategies to enforce privacy and avoid data lock-in.

In Figure 3.1 we observe how a user distributes his data among friends and the chosen Cloud storage service.

3.4.3 FriendBox Application State

Essentially, the `FriendBox` Application State maintains up to date the data management information about users' files. This information expresses which friends store which files and the network address of each friend. This information is needed to perform friend-to-friend storage operations. The maintenance process of this information is carried out by `FriendBox` Storage Clients installed at participants. Clients communicate with the `FriendBox` Application State and update their state information via a Representational State Transfer (REST) API. In addition, this application is the back-end of the `FriendBox` Social Front-end.

Storing the application state of `FriendBox` in the Cloud provides service ubiquity, that is, a user is able to install the `FriendBox` Storage Client in any machine and obtain the distribution of their files and the overall state of the application from this Cloud application. To the best of our knowledge, no other F2F system provides this feature.

The role of the `FriendBox` Application State is illustrated in Figure 3.1. In this figure, we show how a user communicates with the `FriendBox` Application State to transfer state information. In this example, a user sends a message informing that a new file has been stored in the system and containing which friends are responsible for data blocks from that file. As can be seen, the `FriendBox` Application State stores this information using mappings that relate data blocks with the friends who are responsible for them.

As all operations must be validated against the `FriendBox` Application State, it gathers anonymous information about users' activity in order to provide real availability patterns which can be used in future, for instance, to suggest the optimal time to backup or restore files for a user. Furthermore, we will release these availability patters to the community to allow testing novel applications with real user data.

3.5 Advanced aspects

In this section, we provide a technical description about the problems that put the feasibility of F2F systems at risk: *reduced friend-sets* and *availability correlations*. Moreover, we overview the Cloud-based mechanisms introduced in `FriendBox` to mitigate these problems.

3.5.1 Understanding the problems of F2F systems

In `FriendBox`, we assume that friends alternate between *online* and *offline* states, with their online sessions being correlated over time. Correlation can be understood as the high probability that given an online user, his friends are also online, which corresponds well with the strong diurnal pattern empirically observed in social networks like *Facebook* [18].

More formally, for a node v, we denote by $\mathcal{F}_v$ the set of friends at which v wants to store data. We assume this set is built up by leveraging upon real-trust between users in a social network. Since our focus is on domestic users, we assume that node n has limited download and upload bandwidth, denoted by d_v and u_v, respectively. We also limit the number of parallel connections to P_d and to P_u for downloads and uploads, respectively. The storage capacity at node v is denoted by s_v.

To capture availability correlations, we distinguish between availability correlation for online sessions and correlations for offline sessions. As we will see next, this separation provisions us with valuable information about the impact of correlations on scheduling decisions that otherwise would remain hidden. This issue has not been well addressed in the literature. As the number friends is small and they can

be temporarily offline, F2F storage systems provide data availability by means of redundancy.

Data Availability. To assure a given level of availability, `FriendBox` makes use of *Erasure Codes* (ECs), which has been proven to be more efficient in terms of redundancy than replication [25, 43]. As explained in Section 2.2.6, an EC scheme splits an input file into k fragments of $1/k$th the size of the original file. Then, these k fragments are encoded into n redundant blocks k, $k \leq n$, which are stored at different nodes to mask failures. The *data redundancy* required to store a file is thus $\frac{n}{k}$. The original file can then be recovered by collecting any subset of k blocks out of the total n.

Ideally, the amount of redundancy used should be chosen to optimize transfer times considering that the target level of data availability is met. The reason is that the higher the amount of redundancy, the higher the quantity of data to be transferred to friends, and hence, the longer the scheduling times. Traditionally, given the number of fragments for a file k and the target level of data availability A, the number of encoded blocks to upload n and hence, the redundancy rate $\frac{n}{k}$, has been determined as follows:

$$n = \min \left\{ x \in \mathbb{N} \left| \left(\sum_{i=k}^{x} \binom{x}{i} \bar{a}^i (1-\bar{a})^{x-i} \right) \geq A \right. \right\}, \tag{3.1}$$

where $\bar{a}$ is the average host availability. (3.1) simply accounts for all the possible combinations of finding k blocks (or more) out of n, times the probability that this happens.

Two important observations must be discussed here about (3.1). The first is that this equation *assumes that each fragment is stored at a distinct machine*, because otherwise the failure of a single host would imply the loss of multiple fragments, thereby leading to an underestimation of the real data availability given by (3.1). This assumption is not realistic in our case. Due to the reduced number of friends (typically, between 5 and 20), it is very likely that a friend gets assigned more than one fragment.

Second, (3.1) *assumes that hosts are not correlated*, which is by far not true in F2F systems. As before, this implies that (3.1) can highly underestimate the real data availability. Considering a diurnal pattern in host availability, it is almost impossible for a reduced number of storage nodes, irrespective of the amount of redundancy used, to provide high data availability, as all of them will be probably offline during night hours [33].

This leads to the following problem:

Problem 1. Offering a high level of data availability requires redundancy to be added to friends to mask their discontinuous and correlated participation. If the amount of redundancy is too high, a F2F system could become impractical.

Data Transfer Scheduling. Concretely, when we refer to a *transfer*, we mean the connection with a remote node that causes the transfer of a single block of data to it. Clearly, a transfer may be interrupted if the remote node becomes offline during this process. This takes an amount of time, namely, *block transfer time*

(BTT). For this reason, we refer to a *schedule* as the set of transfers concerning the same data object. Furthermore, we refer to as *scheduling policy*, the algorithm that decides the order according to which transfers must occur over time in order to minimize the time to complete a given schedule. We refer to the time to complete a schedule simply as the *time to schedule* (TTS) [39].

Neither the formalization of the data transfer scheduling problem nor the existing scheduling policies take into account availability correlations [39]. This implies that the TTS may grow significantly if, for instance, the least available friend was scheduled last when all friends follow a diurnal pattern. This fact is mirrored in the example of Fig. 3.2, where $n = |\mathcal{F}| = 3$ and the set of potential schedules is S_1 (exactly one encoded fragment to each friend). Because friends present discontinuous participation, we have depicted in gray the time slots where each node is online. This scenario clearly highlights the importance of a good schedule when storage friends are not available at all times.

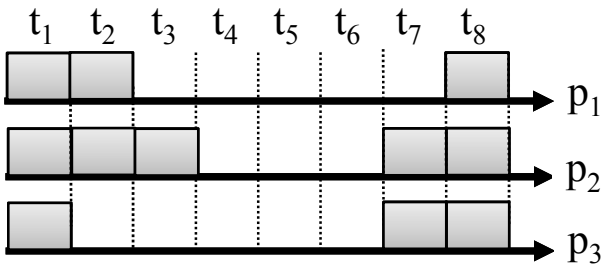

Figure 3.2: Example of availability correlation.

With an optimal schedule, the owner would send a fragment to p_3 in the first time slot, then another to p_1 in time slot t_2, and finally one to p_2 in time slot t_3, concluding the schedule. On the contrary, if the first block is sent to p_2, then the second block to p_1, the owner will have to wait for p_3 to come online again in time slot t_8 to complete the upload. If the availability pattern of p_3 had been considered, this schedule would have been discarded.

Problem 2. In a F2F system, finding the *optimal scheduling plan* is important to provide an efficient storage service in terms of transfer times, which is not trivial. However, the role the *reduced number of friends* and *the availability correlations among them* is even more important in a F2F scenario. That leads us to the question: Are friends enough to achieve a good QoS?

3.5.2 Improving a F2F storage service with a Cloud backend

Indeed, as we describe next, a Cloud back-end is *fundamental* to solve the problems illustrated in the previous section.

Improving data availability. As discussed in the previous section, availability correlation in conjunction with a small friend set makes it hard to maintain a high data availability during the 24 hours of the day. For this reason, we advocate for

ensuring a high data availability during the day hours where friends are mostly online, what we call as *daily data availability*, measured in δ hours. Clearly, as a consequence of availability patterns, the required δ hours of data availability will be mostly concentrated during the online periods of nodes. To this end, we propose a novel hybrid redundancy scheme where a fraction of the data is permanently stored in the Cloud and the rest is supported by friends. To adjust the amount of redundancy required at friends, our algorithm makes use of their availability histories (Section 3.5.3).

First, `FriendBox` defines the fraction of a file to be stored in the Cloud as F_C. A high F_C will take more advantage of the superior availability of the Cloud, thereby reducing the amount of redundancy to be assigned to the friends to support the rest of the file: $1 - F_C$. However, a lower F_C will reduce the monetary costs of storing a file into the system. In this way, `FriendBox` allows users to trade monetary cost for data availability in order to meet their particular storage needs. We are not aware of *any other F2F system that benefits from the Cloud to provide a differentiated storage service per user.*

Note that for a $F_C < 1$ there will be less than k blocks permanently stored in the Cloud. This preserves the distinctive *data privacy* feature of F2F systems, since the Cloud vendor *cannot reconstruct the original file* by any means.

Reducing scheduling times. We resort to the Cloud for reducing scheduling times. In case of uploads, the Cloud acts as a temporal buffer to store those blocks belonging to unavailable nodes during the scheduling process.

To this end, we implemented a *bandwidth maximizing friend-to-Cloud policy*. With this policy a user seeks to reduce transfer times as much as possible by fully utilizing its own bandwidth. If a node responsible for a block is not online, this policy automatically redirects that block to the Cloud service. This way, the node will not have to wait until all blocks are received by their recipients, leading to an important reduction of scheduling times when nodes are unavailable, in particular during the night hours. In any case, the extra blocks pushed to the Cloud are downloaded and deleted afterwards by the nodes to whom they were assigned in first time.

For downloads, we prioritize block downloads from friends since they have no cost for the requester node. If the user cannot gather more than k blocks from available friends, the remaining ones are downloaded from the Cloud. Reducing monetary cost for users.

3.5.3 Monitoring

Currently, there exist no information about *workloads and user behavior in F2F systems*. Furthermore, to the best of our knowledge, related storage systems like Personal Clouds —e.g. Dropbox, SugarSync— have not revealed information about the dynamics of their clients since it is considered strategic information.

We believe that further advances in the field of personal storage need from real-world information coming from deployed systems. To this end, the `FriendBox`

Application State transparently gathers information about, for instance, availability of users and system workloads (storage, bandwidth). Without compromising the privacy of users, we will provide real traces to the research community in order to devise novel mechanisms for improving personal storage systems.

3.6 Implementation

3.6.1 Storage Client: Blending Friend & Cloud Storage

The heart of the `FriendBox` Storage Client is implemented in Erlang [29], a functional programming language conceived to build massive distributed systems. Erlang was created by Ericsson to write highly scalable mission critical telecom server products. Due to the decentralized nature of our system we considered Erlang an appropriate solution. Furthermore, Erlang is used in a large number of leading applications and services that require high levels of concurrency such as Apache CouchDB, GitHub or WhatsApp Messenger. Among others, the main characteristics that have made these services and our own system to opt for Erlang instead of other languages are the following:

1. *Concurrent processes.* Concurrency is one of the main strength of this language. It can spawn a large number of processes running concurrently in Erlang's Virtual Machine (VM) —some users claim being successfully completed a benchmark with 20 million processes [38]—. The VM automatically handles executing processes across multiple processors.

2. *Message based.* Erlang processes communicate using an asynchronous message passing system. Therefore, processes do not get blocked and there is no shared memory among them. Every process has a queue of messages that have been sent by other processes and yet not treated.

3. *Variables that do not vary.* Erlang follows a single-assignment policy for variables, so once a variable is assign a value it can no longer change its value. This is a mechanism to avoid shared states and read/write contention and locks. This way processes are much more scalable and issues like collisions, deadlocks or resource starvation are circumvented.

4. *Pattern matching.* Erlang uses pattern matching to bind variables to values.

5. *Hot code swapping.* Erlang's OTP framework provides a mechanism to achieve live code replacing. This way server applications can be upgraded without stopping their service.

6. *Distributed computing.* Communication to processes created on remote nodes is transparent —i.e. Once the machines are physically allocated by an IP address you can spawn processes on remote VMs and call remote processes as if they were local—. Thus, a cluster of nodes in different locations can be

easily build and exchanging messages among them is nearly identical to local messaging between processes.

Over 4,000 lines of code implement the basic functionality of this software. This includes block level PUT/GET/DELETE storage operations, management of parallel transfer connections, logging, etc.

The software architecture —based on the Erlang OTP Framework [20]— permits to *easily extend* its functionality. For example, currently the software includes two ways of providing data redundancy: *replication* and *Reed-Solomon erasure codes* (see section 2.2.6) (developed in C). However, other advanced data redundancy mechanisms and coding schemes can be introduced as independent modules in `FriendBox` without effort.

We also have incorporated a set of mechanisms to improve `FriendBox` performance. These mechanisms include parallel upload and download transference, exponential backoff retry technique and different redundancy techniques. Besides traditional replication, we include an implementation of Reed-Solomon erasure codes (Section 2.2.6) to optimize the transferred data.

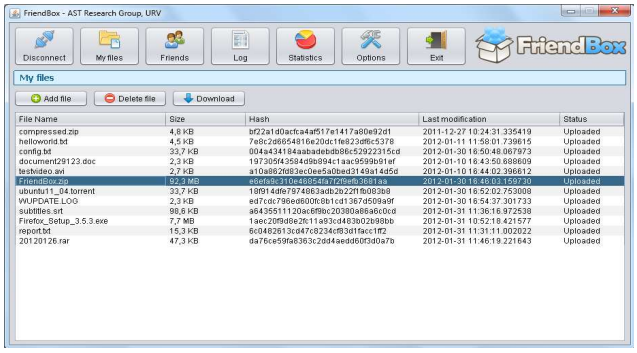

Figure 3.3: View of the `FriendBox` Storage Client.

In the current version of `FriendBox`, the storage client provides a REST API to store data in Amazon S3[1]. Users can configure the fraction (F_C) of their files to be stored in the Cloud. In this sense, to introduce more flexibility on selecting a Cloud provider we are developing APIs for other Cloud storage services such as Google Storage or OpenStack and Personal Cloud services like Dropbox and Box.com.

Finally, users interact with the storage client via a user-friendly Java Swing graphical user interface (GUI). This GUI has been implemented in Java and connected to the storage client through *Jinterface*, a package for communicating Java programs with Erlang processes. In Figure 3.3 we show the storage client GUI. The GUI enables users to perform several actions: data management operations (add, retrieve and delete files), inspecting which friends are currently participating in the storage system or modifying configuration settings (connection ports, Cloud accounts, etc).

[1] http://aws.amazon.com/

3.6.2 Application State powered by Google App Engine

As explained before, the `FriendBox` Application State goal is to maintain a consistent state of the system. Thus, users can retrieve their files distribution from any computer in order to recover them. Besides, it is also responsible for hosting Facebook's web application. It has been developed with 3,000 lines of Python code, plus HTML and CSS to render the `FriendBox` Front-end in Facebook.

Instead of investing in new and expensive server machines to host our application state, we employed Google App Engine [13] (GAE), a PaaS (Section 2.2.4) for developing and hosting web applications in Google-managed data centers. GAE offers an elastic and scalable service platform[2]. As the number of requests increases, GAE automatically allocates more resources for the application to handle the additional demand. Another important aspect is that GAE remains free of charge until a certain level of consumed resources, this allows the deployment of novel applications without incurring in any initial charge.

However, GAE apply some restrictions that must be taken into account before start development phase. Most significant restrictions and characteristics that may affect our system are listed as follows:

- GAE only allows applications written in Python and Java. Furthermore, only arbitrary modules (pure-Python) and classes (The JRE Class White List [26]) are permitted.
- Developers have read-only access to file system on GAE. Nevertheless, there exist workarounds that implement a virtual file system upon GAE DataStore by using the Low-level API to store files [17].
- GAE can only execute code called from an HTTP request. No other protocol is available for developers.
- Unlike traditional databases, GAE DataStore is built on top of Google's BigTable, a scalable non-relational database. Moreover, GAE uses a SQL-like syntax called GQL to interact with data stored in their DataStore.

We had to design our system by avoiding not to be hampered by any of the mentioned limitations.

Both Java and Python would have served well to build the application state server. But since Python is a dynamically-typed language and its philosophy emphasizes in code readability, programming in Python is more agile and results in faster development and easier code maintenance.

For desktop clients to communicate with the application state we had to adopt a solution that operated in the HTTP protocol due to GAE limitations. Therefore, we opted for using a Representational State Transfer (REST) API.

[2]Facebook applications are not hosted within the Facebook environment. Since we enable users to access their information from Facebook, we adopted a PaaS hosting instead deploying the application in our own servers.

REST is an architectural style defined by Dr. Roy Fielding's in his PhD thesis dissertation [15]. REST is a client-server protocol that uses HTTP methods (GET, POST, PUT, DELETE) to communicate. Each HTTP message contains all necessary information to comprehend the query. Responses typically are formatted in HTML, XML or JSON, although it may be an image, plain text or any other content. Unlike other design models such as SOAP, REST does not require specific error messages, instead it uses HTTP status codes [27]. When the same HTTP error code maps to several application errors, an additional header with a simple string helps to remove ambiguity and be human-readable.

Services that are implemented using HTTP and following REST principles are called RESTful services.

In `FriendBox` we implemented a RESTful service to enable communication between desktop clients and GAE's application state server.

```
GET http://astfriendbox.appspot.com/api/files
Authorization: a0571c5b9493187adb5bd07ad0faf279a86251df
```

Listing 3.1: Example of REST API call that returns the metadata of files owned by a user

In Listing 3.1 we illustrate an example of a real REST call implemented in `FriendBox` Application State. All API calls require of an *authorizing token* that is passed as a HTTP header, this token is assigned to a client after validating in the system with its credentials. Tokens expire after a 20 minutes of inactivity. Nevertheless, while `FriendBox` desktop client is running it transparently keeps the session alive. For security reasons, each token is related to an IP address in order to avoid session hijacking by third parties.

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=UTF-8
Date: Wed, 28 Mar 2012 19:29:00 GMT

<?xml version="1.0" encoding="utf-8"?>
  <api>
     <count>2</count>
     <file>
        <filekey>ag9zfmVtcGVlcmktY</filekey>
        <hash>5068fb4d88175563ae7d541d87afba8b086ddbf1</hash>
        <filename>sample_file.txt</filename>
        <size>9723</size>
        <date>2011-12-27 10:24:31.335419</date>
     </file>
     <file>
        <filekey>bfdgtmVtcGVgfh5</filekey>
        <hash>9ad264a3ae9d40a62a3cb941cdabc68a041a01e7</hash>
        <filename>test.zip</filename>
        <size>632547</size>
        <date>2012-01-11 11:58:01.739615</date>
```

```
    </file>
</api>
```

Listing 3.2: Example of response for API call in Listing 3.1

A response for the above API call is exemplified in Listing 3.2. Here we can observe that the call has returned a $200OK$ code, witch indicates that the request has been processed successfully. In the response body we find the list of files owned by the user. Afterwards, the application parses this information and shows it in a proper way.

In addition to the example showed above, `FriendBox` Application State implements a set of API calls to interact with stored resources. This mode the application is able to store and update the relational data structures that reflect the state of users and the location of their data.

On the other hand, for its internal management, this application uses the REST API that Facebook exposes to get a range of user data including friends and profile information. Such information is utilized, for instance, to manage the social relationships among users, i.e., `FriendBox` storage invitations.

3.6.3 Social Front-end in Facebook

The Social Front-end of `FriendBox` is integrated within the Facebook environment. This is possible since the `FriendBox` Application State follows the requirements and communication standards of Facebook. The web site uses Facebook Markup Language (FBML), which is a modification of HTML that includes new tags to interact with Facebook functionalities.

From a user perspective, the Social Front-end presents a useful an ubiquitous source of information about the state of a user's storage service. This application enables users to manage their storage relationships, analyze the distribution of their data and check the Cloud resource consumption.

At the top of Figure 3.4 we observe that the current user (Pedro García) has sent a storage invitation to Marc Sánchez which is in *pending state*. This exemplifies how users interact among them to manage their storage relationships.

Another interesting feature we introduced in this component is a chart that shows how a user's data is distributed among his friends and the Cloud. This chart illustrates where a user's data is stored and to whom a user is storing data.

As many commercial services, we benefit from this component to present to users the amount of consumed Cloud storage resources. In the charts included in the "Cloud Consumption" section, we depict the traffic and storage capacity that have been consumed over the last months. Moreover, actions such as changing the `FriendBox` Storage Client password or listing the files stored in the system are present in this application.

Other useful information that can be found in this Facebook application. For instance, a user is able to pinpoint the location of their storage friends, and thus,

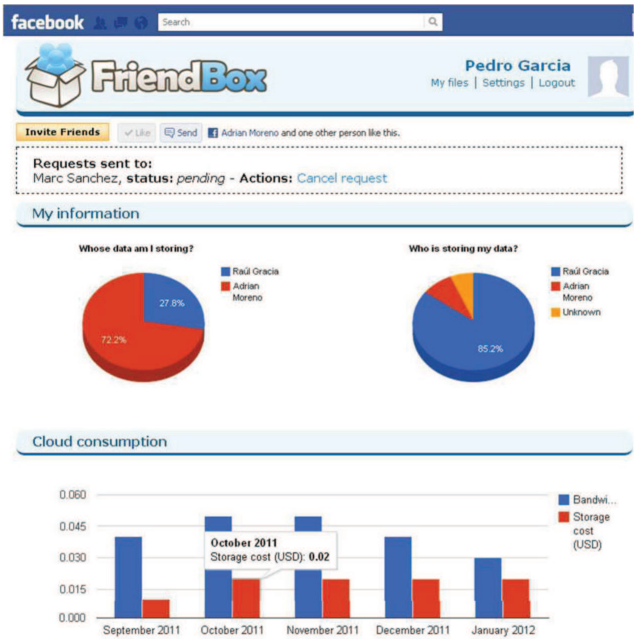

Figure 3.4: `FriendBox` Social Front-end in the Facebook environment.

know where his data is exactly placed. Of course, only if a user's friends agree to disclose their location to friends.

Chapter 4

Experiments

In this chapter we describe the simulation environment as well as parameters like the fraction of data to store in the Cloud, parallel connections and number of nodes participating. In this context, we perform an experiment to shed some light on the feasibility of `FriendBox` *and evaluate key aspects such as scheduling times, data availability or the monetary cost of taking advantage from the Cloud.*

4.1 Scenario and Setup

In this section we detail the simulation environment and the parameters that will influence in the final results. For this reason, fixing the value of this parameters is of considerable importance for the well-being of our experiment. These parameters are depicted in Table 4.1.

Parameter Description and Values	
Nodes in the system	10
Experiment duration	24 hours
Node storage capacity	10GB
Parallel upload/download connections	2, 2
Erasure codes original file fragments (k)	40
Cloud file fraction (F_C)	0.5
Object size (β)	200MB, 400MB, 600MB
Data redundancy (n/k)	2.5

Table 4.1: Parameter configuration in our experimental scenario.

We deployed a group of ten `FriendBox` users in the university campus, simulating that are friends among them in Facebook. To this end, we created a topology of friends based on a real measurement of Friendster [45]: a social networking service that allows users to contact other members, maintain those contacts, and share online content and media with those contacts.

The group topology within the campus is depicted in Figure 4.1. In our scenario, users perform and serve storage operations in the system. To be more concrete, there are two types of nodes depending on their mission in the experiment:

- *Requester nodes.* Nodes 1, 2 and 3 are addressed to continuously perform storage request to the rest of nodes.

- *Non-requester nodes.* These nodes emulate real users, exhibiting diurnal availability patterns.

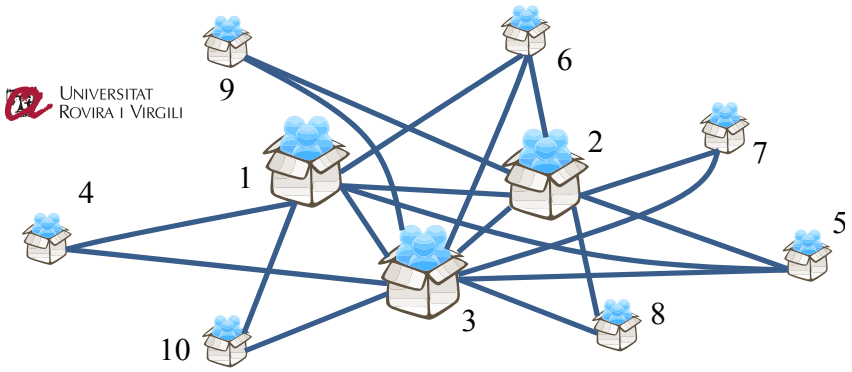

Figure 4.1: Network topology configured in our experiments.

Therefore, at the end of the experiment we will analyze the quality of service received by requester nodes in this deployment.

Nodes deployed in our campus emulate real users with respect to their availability. Concretely, we introduce in each non-requester node an availability trace extracted from a Skype measurement [19]. These traces dictate the online/offline behavior of nodes.

Since nodes are based on real-user behaviors —i.e., diurnal patterns— our experiments suffer from *availability correlations*. The availability of nodes during the experiment (24 hours) is depicted in Figure 4.2. In our experiments, requester nodes are continuously performing storage operations and therefore, they always appear available.

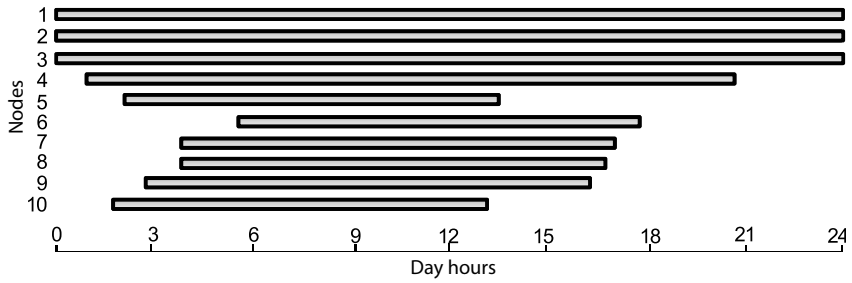

Figure 4.2: Node availabilities configured in our experiments.

The workload model of our experiments is simple. Requester nodes are alternatively performing file downloads and uploads during the whole experiment. These operations are performed over synthetic random files of β bytes.

We fixed the fraction to be stored in the Cloud as the half of the uploaded data ($F_C = 0.5$) and the data redundancy[1] rate to $n/k = 2.5$. With these parameters we will analyze the resulting data availability δ.

[1] Although our data availability algorithm (Section 3.5.2) has been validated via simulation, a *long term* experimental evaluation is left for future work.

Hardware and software configuration	
CPU	Intel Core2 Duo and AMD Athlon X2 processors
RAM	4GB DDR2
Operating system	Debian Linux
Network	Fast Ethernet 100Mbps
Network monitoring	Vnstat

Table 4.2: Hardware and software configuration in our experimental scenario.

Moreover, data transfers of requester nodes are concurrently executed along the experiment to observe the effects of network congestion.

We applied a *random scheduling policy* to schedule transfers among friends for both uploads and downloads [39]. That is, the order of block transfers is chosen completely at random. This scheduling is computationally simple and very common in existing systems. It works as follows: in each time slot, the user transfers one missing block to a storage node chosen uniformly at random among the available ones.

Furthermore, we assume that nodes that have been online in the past will continue to do so in a near future. Which is called *Least-Available First* policy. Therefore, it prioritizes transfers towards nodes that have been less available within a past time window of w time slots.

In addition, `FriendBox` introduces a *bandwidth maximizing friend-to-Cloud policy* (Section 3.5). To wit, if the block's destination friend is unavailable, that block is temporarily uploaded to the Cloud until the friend in charge of that block becomes online. This mechanism reduces the upload time to schedule but incurs an extra cost, as later on the extra blocks must be downloaded from the Cloud by their respective addressees.

As shown in Table 4.2, `FriendBox` clients are hosted in desktop computers (Intel Core2 Duo and AMD Athlon X2 processors) equipped with 4GB DDR2 RAM. The operating system employed is a Debian Linux distribution[2]. Users were connected via a 100Mbps switched Ethernet links. For gathering physical network information we used `vnstat`: a tool that keeps a log of network traffic for a selected interface. The rest of information presented in this chapter –e.g. scheduling times, data distribution– has been gathered by the `FriendBox` log system.

4.2 Measuring scheduling times

In this section we focus on analyzing the obtained times to schedule (TTS) among nodes. Figure 4.3 illustrates the performance of upload and download TTS along one day. Note that scheduling times are faster during the periods of higher node availability. This means that, as in our campus all nodes are connected by Fast Ethernet links, accessing to friends is significantly faster than resorting to the Cloud.

First, we observe that in case of uploads (Figure 4.3), requester nodes transfer 500MB of data ($\beta = 200$MB, $n/k = 2.5$). However, none of these upload schedules

[2]FriendBox works for other platforms such as Windows and Linux Ubuntu.

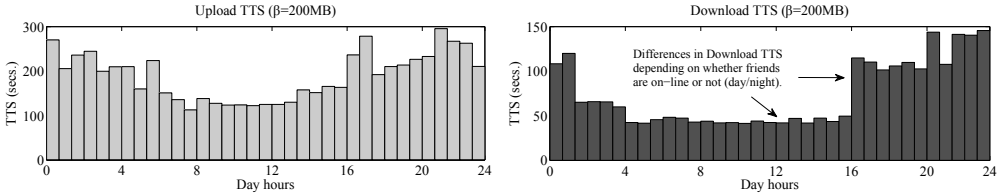

Figure 4.3: Upload and download TTS along one day.

takes more than 5 minutes to finalize. In case of uploading larger files, upload TTS are equally short: around 330 secs for $\beta = 400$MB and 474 secs for $\beta = 600$MB in average.

The average TTS is appreciably shorter for downloads than for uploads. The reason is that requester nodes only need to gather k blocks in order to reconstruct the original file. Further, since these k blocks are mostly collected from friends, transfers among campus nodes are faster than accessing to the Cloud, download TTS are short. Note that all the download schedules finished correctly.

One of the most interesting points of this section refers to the TTS performance depending on the moment of the day. We can observe that scheduling times (uploads and downloads) are significantly shorter in the central hours of the day than in the rest. Actually, this is the time range where the majority of friends are online. From this result, we conclude that in these kind of scenarios the availability of friends improves the TTS performance. This is specially evident in case of downloads, since in the best cases there is no need for accessing the Cloud.

It is specially important to observe that `FriendBox` is able to complete upload schedules irrespective of the availability of friends. Thus, even when a fraction of friends are unavailable, the bandwidth maximizing policy redirects blocks to the Cloud in order to finalize the upload. Note that this is not possible in a pure F2F system; a user should wait until their friends come back online to finalize the upload schedule.

In conclusion, `FriendBox` achieves fast data transfers among nodes and the Cloud. It also manages to upload files even when friends are unavailable, resulting in faster TTS and, therefore, improving quality of service. Finally, we observed the impact of availability correlations on the TTS.

4.3 Data availability

We also evaluate the data availability exhibited by requester nodes. In Figure 4.4 we compare the data availability obtained by `FriendBox` and a pure F2F system in our scenario.

As expected, for the same amount of data redundancy, users obtain a higher data availability with `FriendBox` than with a F2F system. For instance, `FriendBox` nodes 2 and 3 exhibit $\delta = 24$ hours of data availability per day. With a pure F2F approach, they exhibit $\delta = 15$ hours and $\delta = 20$ hours respectively. Thus, storing a fraction of a file into the Cloud alleviates the impact of offline friends on the

resulting data availability.

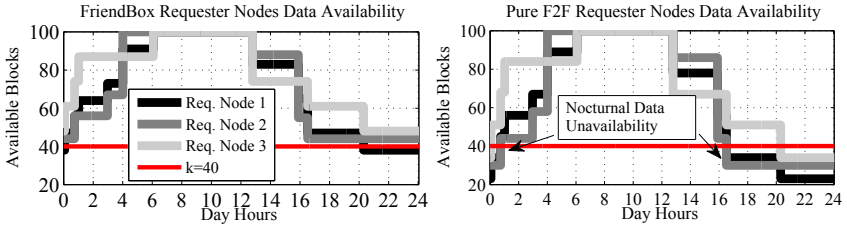

Figure 4.4: Data availability of `FriendBox` compared with a pure F2F system.

In this line, `FriendBox` is able to obtain equal or higher levels of data availability than a F2F system with less data redundancy. This is key for reducing scheduling times and making a F2F system scalable in terms of storage redundancy.

We observe that node 3 exhibits higher levels of data availability than other nodes. The reason is that node 3 has more friends for storing data than nodes 1 and 2. This gives us an important insight: *the size and behavior of a node's friend-set has important effects on data availability.* The more active friends a user has, the more probably will be retrieving data only from those friends.

In conclusion, `FriendBox` is flexible enough to satisfy the data availability requirements of users. Clearly, the amount of data redundancy, the role of the Cloud and the behavior of friends will dictate the service quality.

4.4 Analyzing transfer capacity

In this section, we address the data transfer capacity of `FriendBox`. To this aim, we illustrate in Figure 4.5 the distribution of block transfer times (BTT). Upload and download BTT distributions for friends and the Cloud are plotted separately.

First, the vast majority of block transfers among `FriendBox` nodes are faster than resorting to the Cloud. Again, this is mainly caused due to the high speed network available in the campus. Furthermore, we observe that the distribution of uploaded and downloaded blocks among nodes is quite similar. Obviously, in contrast to domestic network connections, the upload/download bandwidth asymmetry is not relevant within the campus.

Figure 4.5 suggests that uploading blocks to the Cloud is slightly faster than downloading blocks from it. In our opinion, two distinct factors might lead to this result: First, downloading data from the Cloud is more common than uploading data to it. This might induce restrictions on download bandwidth served to a single user. Second, the download process implemented in the `FriendBox` Amazon S3 API is different to the upload process. Whereas blocks are loaded in memory and uploaded directly to the Cloud, the download process receives small chunks of data which are progressively stored on disk. This incurs extra time when downloading blocks from the Cloud. We will address this issue in depth in future tests.

Finally, Figure 4.6 depicts the download bandwidth of requester node measured

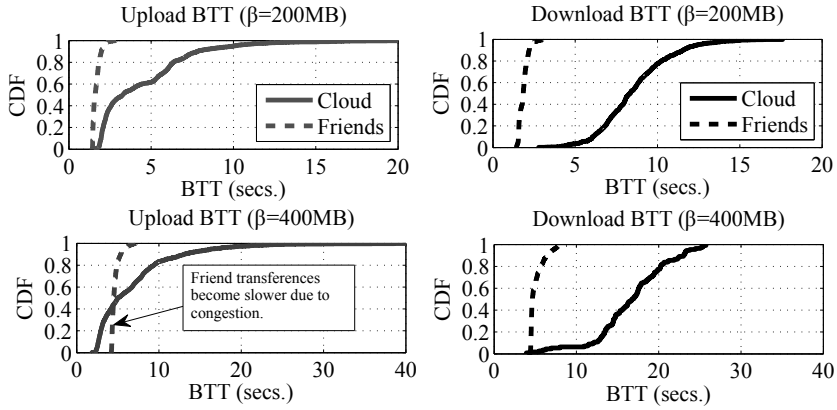

Figure 4.5: Block transfer time (BTT) distribution depending on whether blocks are transferred from the friends or the Cloud.

in two distinct moments: day and night. We observe that the diurnal download is faster than the nocturnal download. Basically, the bandwidth trace indicates that during the day *all* blocks are downloaded at high speed ($\approx$ 4 MB/s); that is, they are gathered from friends.

In case of a nocturnal download, only a fraction of blocks is downloaded from friends. These blocks are the first ones, which have a higher priority than blocks to be downloaded from the Cloud in our scheduling policy. Thus, the requester node finally must resort to the Cloud to finalize the download process. We observe (Figure 4.6) that the differences in transfer speed between the first part of the nocturnal download and the last part are significant. This result corroborates the results obtained in Section 4.2.

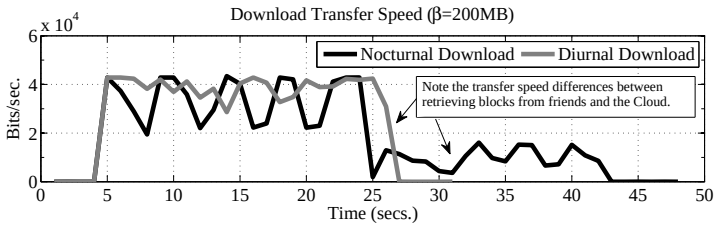

Figure 4.6: Differences between diurnal and nocturnal download transfers.

4.5 Monetary cost

In this section, we address the economic implications of our system to end-users. The monetary cost of `FriendBox` directly depends on the use of the Cloud. In our experiments we used Amazon S3 as a Cloud provider. As of July 2012, Amazon's S3 pricing policy states that they charge 0.125$ per GB/month of storage and 0.120$ per GB of outgoing traffic. Note that incoming traffic is free of charge. Request pricing is much less significant and have been ignored. To be concrete, pricing is

0.01$ per 1,000 PUT, COPY, POST, or LIST requests and 0.01$ per 10,000 GET requests. Realize that there is no charge for DELETE requests.

We compared the costs of `FriendBox` with the same service provided by Amazon S3. The costs depicted in Figure 4.7 capture the expenses to upload and download a file of β bytes, as well as keeping it stored for one month in the system. Note that `FriendBox` should also manage the associated data redundancy of a file.

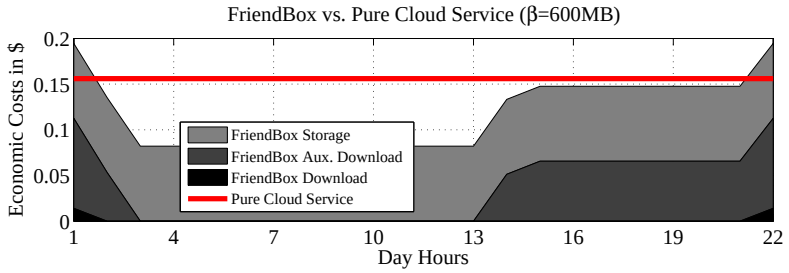

Figure 4.7: Monetary cost comparison of `FriendBox` vs. Amazon S3.

First, we infer that `FriendBox` monetary costs are generally lower than Amazon S3 (Figure 4.7). Regarding storage costs, `FriendBox` is configured with $F_C = 0.5$. This means that only the half of a file is stored in the Cloud, inducing important savings. Clearly, the storage costs of `FriendBox` are limited by the value of F_C, which is selected by the user.

`FriendBox` has a particular cost labeled in Figure 4.7 as *auxiliary download*. This cost comes from the mechanism implemented in `FriendBox` for accelerating uploads (Section 3.5.2): when a node is uploading a file, all the *redundant blocks* whose recipients are offline at the moment of scheduling are redirected to the Cloud. When these previously offline nodes become available, they download those blocks they are responsible for. This introduces an extra economic cost. Thus, depending on the availability of nodes, the cost for auxiliary download will vary. Note that in our scenario, this cost is only present in the moments of lowest node availability.

Actually, in our experimental scenario only 2 download schedules resorted to the Cloud to finalize. This suggests that `FriendBox` can reduce costs in download traffic compared with a Cloud service. However, this cost is dictated by the amount of redundant blocks placed in friends and the availability of friends at the download instant.

Next we discuss the impact of availability correlations upon the monetary costs of the `FriendBox` service. Clearly, performing storage operations during the moments of lower availability of friends –i.e. during the night– is significantly more expensive than during the day. This leads us to a clear conclusion: `FriendBox` should be aware of the availability of a user's friends to give advise about the resulting service costs.

We conclude that `FriendBox` can reduce the monetary cost of a pure Cloud solution, specially in the long term. These costs correspond to the permanent Cloud storage and the Cloud outgoing traffic of future file downloads. In addition, two parameters dictate the costs of `FriendBox` service: the fraction of data permanently

stored in the Cloud (F_C) and the amount of redundancy generated to guarantee data availability (n/k). Finally, the availability patterns of friends produces variations in `FriendBox` service costs along the day.

4.6 Erasure Codes Performance

In the final part of the evaluation, we focus on the computational erasure coding scheme used in `FriendBox`. In this particular experiment, the hardware employed is different to the previous tests (Intel Pentium 4 3GHz and 2GB RAM).

For each parameter configuration in Figure 4.8 we plotted the average execution time from 100 executions. At first glance, we infer from Figure 4.8 that coding is generally affordable for domestic hardware when the size of the file is small or medium. However, when the size of the file to manage is large, coding becomes quite expensive in terms of both CPU time and memory. Therefore, although coding can be used for managing large files sporadically, it seems not adequate for performing *frequent* operations on large files.

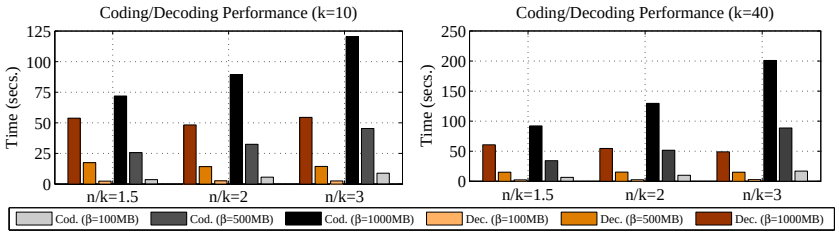

Figure 4.8: `FriendBox` Erasure Codes performance.

Moreover, in Table 4.3 we observe the detailed results of the experiment. The performance is measured in seconds and the standard deviation (σ). Examining the table we observe that performance varies depending on the number of blocks required for recovery (k), the original file size (β) and the degree of data redundancy (n/k). In this context, coding performance is directly affected by the increasing of data redundancy because more blocks have to be created. Otherwise, since decoding only requires k out of n blocks, its performance is not affected as we increment redundancy.

As expected, whereas the encoding process is impacted by β, k and n/k, the decoding process is mainly affected by the size of the file β.

We will study new ways to optimize the performance of the Erasure Codes scheme included in `FriendBox`.

	$n/k = 1.5$		$n/k = 2$		$n/k = 3$	
Encoding $k = 10$	*avg.*	σ	*avg.*	σ	*avg.*	σ
$\beta = 100$ MB	3.60	0.85	5.63	0.76	8.89	0.97
$\beta = 500$ MB	25.65	3.01	32.52	3.73	45.41	6.36
$\beta = 1000$ MB	71.93	5.45	89.22	6.70	120.43	8.56
	$n/k = 1.5$		$n/k = 2$		$n/k = 3$	
Decoding $k = 10$	*avg.*	σ	*avg.*	σ	*avg.*	σ
$\beta = 100$ MB	2.39	0.43	2.60	0.42	2.53	0.28
$\beta = 500$ MB	17.51	2.11	14.23	1.63	14.38	3.05
$\beta = 1000$ MB	53.81	4.26	48.30	3.57	54.45	4.32
	$n/k = 1.5$		$n/k = 2$		$n/k = 3$	
Encoding $k = 40$	*avg.*	σ	*avg.*	σ	*avg.*	σ
$\beta = 100$ MB	6.46	0.85	10.04	0.64	16.92	0.70
$\beta = 500$ MB	34.24	2.61	51.62	1.81	88.63	1.90
$\beta = 1000$ MB	92.14	3.89	129.56	4.01	201.01	4.76
	$n/k = 1.5$		$n/k = 2$		$n/k = 3$	
Decoding $k = 40$	*avg.*	σ	*avg.*	σ	*avg.*	σ
$\beta = 100$ MB	2.46	0.29	2.60	0.25	2.77	0.36
$\beta = 500$ MB	15.09	2.83	15.22	2.35	15.12	1.84
$\beta = 1000$ MB	60.74	3.30	54.66	4.01	48.97	3.31

Table 4.3: Coding/Decoding times (in seconds) and standard deviation.

Chapter 5

Conclusions

In this last chapter we conclude our work enumerating the main conclusions and achieved goals through our study. Finally, we propose future work which can be performed on the topic of this Master Thesis.

5.1 Conclusions

This thesis has been devoted to improve quality of service of friend-to-friend (F2F) storage systems.

In a spirit of contextualizing our work, we provided a solid background covering several topics. We thoroughly described different kinds of storage systems, providing examples to better understand how they work. After describing the main aspect of storage systems we provided a general comparison of them. Then, we thoroughly surveyed existing F2F and peer-assisted systems and discussed the major differences between them and our approach.

We have seen that pure F2F storage systems cannot provide a storage service comparable to Cloud offerings due to their reduced node degree and availability correlations. Research done on this area concludes that users in a F2F storage system are likely to have a reduced number of trustable friends. Moreover, friends present significant correlation in their activity patterns. This implies that it is probable to find the majority of friends simultaneously offline, particularly during night hours. Which makes it impossible to maintain high data availability.

The main contribution of this thesis is `FriendBox`: a system aimed to improve Quality of Service (QoS) offered by F2F storage systems. In `FriendBox` we resort to Cloud back-end in order to maintain high data availability during the 24 hours of the day. Files are coded using an erasure codes scheme to obtain data redundancy and scattered throughout friends and the Cloud. Any entity of the system is not trusted enough blocks so nobody except the owner is be able to reconstruct the original file. Furthermore, to reduce transfer times we take advantage of the Cloud so when a user backs up a file, the Cloud acts as a temporal buffer to store those blocks belonging to unavailable nodes during the scheduling process. Afterwards, these blocks are downloaded by the nodes to whom they were assigned in first time. Leading to an important reduction of transfer times. At download time, users frist try to obtain file blocks from their friends. But if the user cannot gather enough blocks from available friends, the remaining ones are downloaded from the Cloud.

Additionally, `FriendBox` transparently gathers information about availability of users and system workloads (storage, bandwidth). Without compromising the

privacy of users, we will provide real traces to the research community in order to devise novel mechanisms for improving personal storage systems.

Finally, we performed experiments in our university campus to shed some light on the feasibility of `FriendBox`. We demonstrated that `FriendBox` provides high data transfer performance and user-defined data availability guarantees. We also examined the impact of the correlated availabilities and analyzed the monetary costs of using Cloud services, showing its economic feasibility.

In our opinion, this thesis represents a step further towards the achievement of service quality in the context of F2F storage systems.

5.2 Directions for future work

Nowadays, `FriendBox` has released a first beta version which has been tested in the URV campus with dozens of users. However, `FriendBox` is an ambitious project that is currently growing fast towards promising stages. Next, we describe the future steps of `FriendBox`.

Our primary objective is to leverage a secure, flexible and efficient Internet-scale storage system with `FriendBox`; to this end, we recently introduced NAT traversal mechanisms and we are working on data repair mechanisms. Furthermore, we are also introducing cryptography techniques to ameliorate the impact of storage and communication attacks, enforcing user security and privacy. In addition, we will introduce a set of tools in order to study the efficacy in the long term data maintenance. This represents the first milestone of `FriendBox`, that will be reached in the next months.

Another issue is to investigate the development of advanced mechanisms to automatically produce the necessary amount of redundancy thanks to the transparent Cloud and friends monitoring. And, therefore, optimize data transferred and economic costs depending on user requirements.

Since `FriendBox` is based on trending technologies it offers us the possibility of continue research and developing. For instance, we are planning to integrate `FriendBox` into the RealCloud and CloudSpaces projects. The RealCloud project is aimed to develop a complete Cloud Computing environment using a Tier-3 data center. On another front, CloudSpaces will devise an open service platform providing privacy-aware data sharing as well as interoperability mechanisms among heterogeneous Clouds.

Bibliography

[1] Eytan Adar and Bernardo A. Huberman. Free riding on gnutella. *First Monday*, 5:2000, 2000.

[2] Atul Adya, William J. Bolosky, Miguel Castro, Gerald Cermak, Ronnie Chaiken, John R. Douceur, Jon Howell, Jacob R. Lorch, Marvin Theimer, and Roger P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper. Syst. Rev.*, 36:1–14, 2002.

[3] Windows Azure. http://www.windowsazure.com/, Retrieved: July 2012.

[4] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: Pick two. In *HotOS'03*, pages 1–6, 2003.

[5] Box.com. https://www.box.com/, Retrieved: July 2012.

[6] Kyle Chard, Simon Caton, Omer Rana, and Kris Bubendorfer. Social cloud: Cloud computing in social networks. In *IEEE CLOUD'10*, pages 99–106, 2010.

[7] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, pages 120–132, 2003.

[8] CrashPlan. http://www.crashplan.com, Retrieved: August 2012.

[9] L.A. Cutillo, R. Molva, and T. Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[10] Google Drive. https://drive.google.com, Retrieved: July 2012.

[11] Dropbox. https://www.dropbox.com, Retrieved: July 2012.

[12] Peter Druschel and Antony Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *In HotOS VIII*, pages 75–80, 2001.

[13] Google App Engine. https://developers.google.com/appengine/, Retrieved: July 2012.

[14] Facebook Newsroom Key facts. http://newsroom.fb.com/content/default.aspx?NewsAreaId=22, Retrieved: July 2012.

[15] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[16] Rackspace Cloud Files. http://www.rackspace.com/cloud/public/files/, Retrieved: July 2012.

[17] Gae filestore: Simple Virtual File System on Google App Engine DataStore. http://code.google.com/p/gae-filestore/, Retrieved: July 2012.

[18] Scott A. Golder, Dennis M. Wilkinson, and Bernardo A. Huberman. Rhythms of social interaction: Messaging within a massive online network. In *Communities and Technologies*, pages 41–66. 2007.

[19] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *IPTPS'06*, pages 1 – 6, 2006.

[20] OTP Design Principles User's Guide. http://www.erlang.org/doc/design_principles/users_guide.html, Retrieved: July 2012.

[21] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: Highly durable, decentralized storage despite massive correlated failures. In *IN PROC. OF NSDI*, 2005.

[22] Ramakrishna Kotla, Lorenzo Alvisi, and Mike Dahlin. Safestore: a durable and practical storage system. In *USENIX ATC*, pages 10:1–10:14, 2007.

[23] J. Kubiatowicz et. al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN*, 35:190–201, 2000.

[24] J. Li and F. Dabek. F2f: Reliable storage in open networks. In *IPTPS'06*, 2006.

[25] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure code replication revisited. In *IEEE P2P'04*, pages 90–97, 2004.

[26] The JRE Class White List. https://developers.google.com/appengine/docs/java/jrewhitelist, Retrieved: July 2012.

[27] Wikipedia List of HTTP status codes. http://en.wikipedia.org/wiki/List_of_HTTP_status_codes, Retrieved: July 2012.

[28] S. Pearson. Taking account of privacy when designing cloud computing services. In *Software Engineering Challenges of Cloud Computing*, pages 44 –52, 2009.

[29] Erlang programming language. http://www.erlang.org/, Retrieved: July 2012.

[30] FriendBox project web site. http://ast-deim.urv.cat/friendbox, Retrieved: July 2012.

[31] I. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[32] Amazon S3. http://aws.amazon.com/s3/, Retrieved: July 2012.

[33] Rajesh Sharma, Anwitaman Datta, Matteo Dell'Amico, and Pietro Michiardi. An empirical study of availability in friend-to-friend storage systems. In *IEEE P2P'11*, pages 348–351, 2011.

[34] Google Cloud Storage. https://developers.google.com/storage/, Retrieved: July 2012.

[35] Ye Sun, Fangming Liu, Bo Li, Baochun Li, and Xinyan Zhang. Fs2you: Peer-assisted semi-persistent online storage at a large scale. In *INFOCOM*, pages 873–881, 2009.

[36] R. Sweha, V. Ishakian, and A. Bestavros. Angels in the cloud: A peer-assisted bulk-synchronous content distribution service. In *IEEE CLOUD'11*, pages 97–104, 2011.

[37] Raymond Sweha, Vatche Ishakian, and Azer Bestavros. Angels in the cloud: A peer-assisted bulk-synchronous content distribution service. In *Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing*, CLOUD '11, pages 97–104, 2011.

[38] Stress testing erlang. https://groups.google.com/d/msg/comp.lang.functional/5kldn1QJ73c/T3py-yqmtzMJ, Retrieved: July 2012.

[39] L. Toka, M. Dell'Amico, and P. Michiardi. Data transfer scheduling for p2p storage. In *IEEE P2P'11*, pages 132–141, 2011.

[40] László Toka, Matteo Dell'Amico, and Pietro Michiardi. Online data backup: A peer-assisted approach. In *Peer-to-Peer Computing*, pages 1–10, 2010.

[41] Lazslo. Toka, Matteo Dell'Amico, and Pietro Michiardi. Online data backup: A peer-assisted approach. In *IEEE P2P'10*, pages 1–10, 2010.

[42] Dinh Nguyen Tran, Frank Chiang, and Jinyang Li. Friendstore: cooperative online backup using trusted nodes. In *SocialNets'08*, pages 37–42, 2008.

[43] Hakim Weatherspoon and John D. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *IPTPS'02*, pages 328–338, 2002.

[44] Christo Wilson, Bryce Boe, Alessandra Sala, Krishna P.N. Puttaswamy, and Ben Y. Zhao. User interactions in social networks and their implications. In *EuroSys'09*, pages 205–218, 2009.

[45] R. Zafarani and H. Liu. Social computing data repository at ASU, 2009.