
Universitat Rovira i Virgili
Department of Computer Engineering and Mathematics

# On Personal Storage Systems: Architecture and Design Considerations

A thesis submitted for the degree of

*PhilosophiæDoctor (PhD)*

Presented by:
Raúl Gracia Tinedo

Advised by:
Dr. Pedro García López
Dr. Marc Sánchez Artigas

Tarragona, 2015

Raúl Gracia Tinedo

# On Personal Storage Systems: Architecture and Design Considerations

Ph.D. Dissertation

*Advised by* Dr. Pedro García López and Dr. Marc Sánchez Artigas

Department of Computer Engineering and Mathematics

UNIVERSITAT
ROVIRA I VIRGILI

Tarragona, 2015

UNIVERSITAT ROVIRA I VIRGILI

Departament d'Enginyeria [DEIM] Informàtica i Matemàtiques

FAIG CONSTAR que aquest treball, titulat "On Personal Storage Systems: Architecture and Design Considerations", que presenta Raúl Gracia Tinedo per a l'obtenció del títol de Doctor, ha estat realitzat sota la meva direcció al Departament d'Enginyeria Informàtica i Matemàtiques d'aquesta universitat i que compleix els requeriments per poder optar a la Menció Internacional.

HAGO CONSTAR que el presente trabajo, titulado "On Personal Storage Systems: Architecture and Design Considerations", que presenta Raúl Gracia Tinedo para la obtención del título de Doctor, ha sido realizado bajo mi dirección en el Departamento de Ingeniería Informática y Matemáticas de esta universidad y que cumple los requisitos para poder optar a la Mención Internacional.

I STATE that the present study, entitled "On Personal Storage Systems: Architecture and Design Considerations", presented by Raúl Gracia Tinedo for the award of the degree of Doctor, has been carried out under my supervision at the Department of Computer Engineering and Mathematics of this university, and that it fulfills all the requirements to be eligible for the International Doctorate Award.

Tarragona, 15 de Juny/15 de Junio/June 15, 2015

El/s director/s de la tesi doctoral
El/los director/es de la tesis doctoral
Doctoral Thesis Supervisor/s

Dr. Pedro García López

Dr. Marc Sánchez Artigas

# Abstract

Nowadays, end-users require higher amounts of reliable and available on-line space to store their personal information (e.g., documents, pictures). This motivates researchers to devise and evaluate novel *personal storage systems* in order to cope with the growing storage demands of users. In this dissertation, we focus our efforts to study two emerging personal storage architectures: *Personal Clouds* and *social storage systems*. As one can easily infer, both architectures are radically different and pursue distinct goals.

On the one hand, Personal Clouds such as Dropbox or SugarSync, are *centralized* on-line cloud services for personal information that enable users to store, synchronize and share data from a variety of devices and operating systems. On the other hand, a social storage service is built upon a *decentralized* system that leverages preexisting trust or social relationships between users to enable mutually beneficial resource sharing.

According to these storage architectures, this thesis contributes in two general challenges.

Our first challenge is to *understand the operation and performance of Personal Clouds*. Under this ambitious challenge, (i) we first contribute by unveiling the internal structure of a global-scale Personal Cloud, namely UbuntuOne (U1), by describing its architecture, metadata service and storage interactions. Moreover, (ii) we provide a back-end analysis of U1 that includes the study of the storage workload, the user behavior and the performance of the U1 metadata store. We also suggest improvements to U1 that can also benefit similar Personal Cloud systems in terms of storage optimizations, user behavior detection and security.

Apart from the internal facets of Personal Clouds, users and applications may interact with these services externally. In this sense, we also contribute by (iii) measuring and characterizing the transfer performance (e.g., speed, variability) of Personal Cloud REST API services. Furthermore, we realized that these API services may be a vector of abuse of Personal Clouds free accounts, which motivated us to study this vulnerability and propose several countermeasures.

Overall, our contributions under this challenge provide a holistic view of the behavior of Personal Clouds that extends the state-of-the-art knowledge on these systems.

Our second challenge is to *explore the Quality of Service (QoS) of social storage systems*. To undertake this challenge, we noticed that social storage systems are highly affected by availability correlations and very small groups to store data. This particular scenario poses new research questions that remain unsolved for providing an adequate storage QoS to users.

In this sense, our first contribution is (iv) to analyze the QoS of social storage systems in terms of data availability, transfer performance and load balancing. Moreover, (v) we evaluate the suitability of common approaches for estimating data availability when users are correlated, showing that these techniques are severely biased and how this impacts on the data redundancy calculation. In consequence, we propose a history-based method to calculate data availability tailored to heterogeneous and correlated availabilities.

Given the performance limitations inherent to many social storage scenarios, (vi) we design a hybrid architecture to enhance the QoS achieved by the system that combines user resources and cloud storage to let users infer the right balance between control and performance. In the experimental evaluation of this architecture, we specially focus on the role that the social topology plays in the system's performance.

Therefore, we contribute new insights on the performance of social storage systems as well as alternative architectural designs. Our contributions may help to increase the feasibility and performance of these systems, which is fundamental to their eventual adoption by end-users.

*Keywords*: Personal Clouds, Performance Analysis, Distributed Storage, Friend-to-Friend Systems, Social Clouds.

# Resum

Actualment, els usuaris necessiten grans quantitats d'espai d'emmagatzematge remot per guardar la seva informació personal (p.e., documents, fotografies). Això motiva als investigadors a fer recerca i crear nous sistemes d'emmagatzematge d'informació personal per cobrir les creixents necessitats dels usuaris. En aquesta dissertació, estudiarem dues arquitectures emergents de sistemes d'emmagatzematge d'informació personal: els *Núvols Personals* i els *sistemes d'emmagatzematge social*. Com ara veurem, ambdues arquitectures són radicalment diferents i persegueixen objectius dispars.

D'una banda, els Núvols Personals, com Dropbox i SugarSync, són sistemes centralitzats d'emmagatzematge al núvol per informació personal que permeten als usuaris guardar, sincronitzar i compartir informació des d'una gran varietat de dispositius i sistemes operatius. D'altra banda, un sistema d'emmagatzematge social té una arquitectura descentralitzada i es beneficia dels lligams socials o de confiança existents entre usuaris per permetre la compartició de recursos al sistema. Donades aquestes arquitectures d'emmagatzematge, aquesta tesi contribueix en dos reptes generals.

Com a primer repte, ens proposem entendre l'operació i rendiment d'un Núvol Personal. Dins d'aquest ambiciós desafiament, les nostres contribucions són: (i) contribuïm desvelant l'operació interna d'un Núvol Personal d'escala global, anomenat UbuntuOne (U1), incloenthi la seva arquitectura, el seu servei de metadades i les interaccions d'emmagatzematge de dades. A més, (ii) proporcionem una anàlisi de la part de servidor d'U1 on estudiem la càrrega del sistema, el comportament dels usuaris i el rendiment del seu servei de metadades. També suggerim tota una sèrie de millores potencials al sistema, en termes d'optimització de procés de dades, seguretat i detecció de comportament d'usuari que poden beneficiar sistemes similars.

A més dels aspectes interns dels Núvols Personals, els usuaris i les aplicacions poden interactuar amb aquests serveis externament. En aquest sentit, en aquesta tesi també contribuïm (iii) mesurant i analitzant la qualitat de servei (p.e., velocitat, variabilitat) de les transferències sobre les REST APIs oferides pels Núvols Personals. A més, durant aquest estudi, ens hem

adonat que aquestes interfícies poden ser objecte d'abús quan són utilitzades sobre els comptes gratuïts que normalment ofereixen aquests serveis. Això ha motivat l'estudi d'aquesta vulnerabilitat, així com de potencials contramesures.

Sobretot, les nostres contribucions en aquest repte proporcionen una visió holística del comportament i la naturalesa dels Núvols Personals que va més enllà de la literatura actual en aquest camp.

El segon repte d'aquesta tesi consisteix a explorar la qualitat de servei que els sistemes d'emmagatzematge social poden aconseguir. Per portar a terme aquest estudi, primer vam entendre que a aquests sistemes els hi afecten greument les correlacions en les hores de disponibilitat dels usuaris així com la reduïda mida dels grups d'amics on es guarden les dades. Llavors, les característiques particulars d'aquests sistemes requereixen especial atenció per entendre la qualitat de servei que poden oferir.

En aquest sentit, la nostra primera contribució és (iv) analitzar la qualitat de servei que els sistemes d'emmagatzematge social poden proporcionar en termes de disponibilitat de dades, velocitat de transferència i balanceig de la càrrega. A més, (v) analitzem la idoneïtat d'aplicar tècniques de manegament de dades provinents de sistemes de gran escala, com ara les utilitzades per al càlcul de la disponibilitat de les dades. Una de les nostres observacions és que els mètodes tradicionals no són adequats per les particularitats dels sistemes d'emmagatzematge social. Per tant, proposem un mètode per calcular la disponibilitat de les dades que es basa en el comportament previ dels usuaris, el qual es demostra una tècnica millor en aquest context.

Donades les limitacions de rendiment dels sistemes d'emmagatzematge social purament descentralitzats, (vi) dissenyem una arquitectura híbrida que combina recursos del núvol i dels usuaris. Aquesta arquitectura té com a objectiu millorar la qualitat de servei del sistema i deixa als usuaris decidir la quantitat de recursos utilitzats del núvol, o en altres paraules, és una decisió entre control de les seves dades i rendiment. A l'avaluació experimental d'aquesta arquitectura posem especial èmfasi en el rol que la topologia social té al rendiment del sistema.

Per tant, aportem noves perspectives sobre el rendiment dels sistemes d'emmagatzematge social, així com dissenys d'arquitectures alternatives. Les nostres contribucions poden ajudar a millorar la viabilitat d'aquests sistemes, el qual és fonamental per a la seva eventual adopció pels usuaris finals.

*Paraules clau*: Núvols Personals, Anàlisi de Rendiment, Emmagatzematge Distribuït, Sistemes d'Amic-a-Amic, Núvols Socials.

# Acknowledgements

This dissertation has been written during my time at the "Arquitectures i Serveis Telemàtics (AST)" research group at Universitat Rovira i Virgili.

In first place, I would like to thank my advisors Dr. Pedro García López and Dr. Marc Sánchez Artigas their help and (not only professional) advise through my formation as a researcher. From the beginning of my Ph.D., they let me freely discuss and explore alternatives on a defined research topic, and this rapidly translated into full liberty to propose, develop and believe in my own research ideas. Without their teamwork as advisors and their open-minded perspective of what is doing research, I would not have been able of completing this thesis; and no doubt about it, they reinforced my desire of working as a researcher in the future. Thank you very much Pedro and Marc for our stimulating professional and personal relationship during these years.

Secondly, I would like to thank Dr. Dalit Naor (IBM Haifa Labs) and Dr. Sivan Toledo (Tel-Aviv University) for accepting me as an intern in their respective groups. The months I spent in Israel were not only fruitful from a professional perspective, but also very enriching from a personal viewpoint. Thanks Dalit and Sivan (and the rest of people there) for your support during my internship.

In third place, I also need to say "thank you very much guys!!" to all the people in the AST research group. You are one of the main reasons because I do not feel bad when Monday comes! It is hard to express how I enjoyed our long, and often chaotic, discussions about any aspect of life (I lost count of how many times we saved the world!). Specially, I thank Cristian Cotes and Adrián Moreno, for the so common hilarious moments that we enjoyed together in the lab. I finished my thesis without going crazy because of you guys!

Last but not least, my deepest appreciation goes to my family and friends who make my life something that I feel lucky and honored to enjoy. I would like to specially thank my parents Genoveva and Eduardo, my brothers Sandra and Eduardo, my uncle Juan, and Noe, Marc, Juan Héctor, Ana Belén, Eva and Eloi for their love and support.

*I appreciate you all with all my heart.*

Raúl Gracia Tinedo

Vila-Seca, April 2015

*It is hard to fail,*
*but it is worse never to have tried to succeed.*

Theodore Roosevelt

*You are as young as your self-confidence,*
*as old as your fears;*
*as young as your hope;*
*as old as your despair.*

Samuel Ullman

*The true sign of intelligence is*
*not knowledge but imagination.*

Albert Einstein

# Contents

# List of Figures

# List of Tables

# 1
# Motivation and Challenges

In a recent report, the International Data Corporation (IDC) stated that "a majority of the information in the digital universe, 68% in 2012, is created and consumed by end-users; watching digital TV, interacting with social media, sharing images and videos between devices and around the Internet, and so on" [1]. A significant fraction of this information can be classified as *personal data* and constitutes a fundamental part of the ever-growing digital lives of users.

Naturally, most of the data that users generate should be preserved or *stored* along time. Thus, devising novel *personal storage systems* to cope with the storage necessities of users has long been a relevant research line [2, 3, 4, 5, 6]. In this thesis, we focus on two particular personal storage architectures:

- **Personal Clouds**: A Personal Cloud is an online cloud service for personal information that enables users to store, synchronize and share data from a variety of devices and operating systems (OSes). Moreover, Personal Clouds are a platform to deploy third-party applications that provide value-added services on users' data.
- **Socially-oriented distributed storage systems**: This kind of storage systems —namely *social storage systems* for brevity— leverages preexisting trust relationships between users to enable mutually beneficial resource sharing. The main difference between social and traditional decentralized storage systems lies in the social component, which facilitates long term cooperation with lower privacy and security requirements.

As one can easily infer, both architectures are radically different and pursue distinct goals. On the one hand, Personal Clouds are *centralized* systems designed to provide a rich and massive high-quality storage service to end-users. However, due to their proprietary nature, very little is known about their internal operation, infrastructure and supported workload.

On the other hand, a *decentralized* alternative such as a social storage system gives users the control over their personal information and resources, which engenders trust and cooperation. Unfortunately, despite the potential benefits that the social component brings to distributed storage systems, not enough attention has been paid to explore their Quality of Service (QoS).

In this thesis, we aim to explore these architectures from an empirical perspective.

## 1.1 Personal Clouds: Analyzing Global-Scale Storage Services

In one decade, an entire ecosystem of cloud storage services has emerged to satisfy the need for storage of end-users. To illustrate this, Gartner Inc. forecasts a growth of 36% in the volume of digital information that users will store in the cloud from 2011 to 2016 [7]. In other words, users will store more than a third of their data in the cloud by 2016.

One of the factors that motivates this "exodus to the cloud" lies deeply in the widespread necessity for *online and ubiquitous* storage that users currently exhibit. This is exacerbated by the fact that users need their personal information to be accessed by a multitude of devices and OSes. Apparently, we are witnessing a paradigm shift from the *personal computer* to the *Personal Cloud* to cope with the nowadays' requirements of users.

Over the last years, the concept of Personal Cloud has been materialized by several successful commercial offerings. Services like Dropbox, Box or SugarSync provide online storage, file synchronization, sharing, as well as accessibility from a variety of mobile devices and the Web. Furthermore, Personal Clouds are also becoming a popular platform to deploy external applications, such as photo viewers or document editors, that give added value to the personal storage service itself. According to the market reports, these services are meeting well users' needs; for instance, Dropbox's user population grew from 100 to 200 million only in 2013 [8].

Unfortunately, very little is known about what happens *behind the scenes* in a Personal Cloud. Typically, the implementation of these services is proprietary and it is difficult to fully understand their back-end operation from external vantage points. Besides, despite their broad adoption, many practitioners desire to understand the QoS of Personal Clouds to choose a particular vendor or to benefit from storage diversity [9, 10]; to wit, there is little or no public information about the control policies that Personal Clouds may enforce, or about the factors impacting on their performance.

At the time of this writing, it is known that Dropbox decouples the management of file contents (*data*) and their logical representation (*metadata*) [11]. Thus, Dropbox only owns the metadata service, which processes requests that affect the virtual organization of files in user volumes. The actual contents of files are stored separately in a third party cloud provider (Amazon S3). However, this is the boundary of current knowledge; more research is required to understand the internal operation of Dropbox-like services, in terms of infrastructure, metadata organization or QoS characterization, to name a few.

The opaque operation of Personal Clouds may have consequences. That is, the lack of knowledge around the internal operation of Personal Clouds may limit research advances in

this field. Moreover, developers integrating third-party applications in Personal Cloud platforms may have difficulties to understand the performance implications of choosing one vendor or another. As a result, we believe that it is essential to understand in depth how these services operate due to their relevance and scale. This leads to the following challenge:

**Challenge 1: Understanding the operation and performance of Personal Clouds**.

## 1.2 Social Storage Systems: A True Decentralized Alternative?

Many users are reluctant to move all their data to centralized storage services due to the large amount of control ceded to the service provider, and the lack of trust that users may feel in such situation [12, 13]. In fact, this motivated the research on decentralized approaches for personal storage [14]. Broadly speaking, the existing *decentralized* personal storage systems can be classified into *P2P* and *social storage* systems.

Essentially, P2P systems provide a unified and self-organizing management of the sparse disk space of users to storage, replicate and maintain data. Thus, users participating in a P2P storage system contribute part of their local storage resources in exchange of available and durable online storage space [3, 15, 16, 17]. However, P2P storage systems have been studied in depth over the last decade but their adoption by end-users has been lower than expected.

Among the limitations P2P storage systems, the *instability and heterogeneity* of peers is an important issue that hinders the provision of an appropriate service quality [18]. Even worse, despite important efforts [19], the existence of *selfish behaviors* (e.g., free-riding) and the *lack of trust* among participants make end-users reluctant to adopt P2P storage systems to store their personal information.

Social storage systems originally emerged to overcome many of the limitations of P2P storage systems. These systems rely on the synergy between social networks and storage systems: users store their data in a set of social or real-world friends [20, 21, 22, 23]. Thus, data is neither stored in a centralized server nor in unknown peers, enabling users to retain the control of their data. Moreover, the social component of social storage systems alleviates many undesirable problems present in large-scale distributed systems such as security, trust, and incentives.

However, a social storage system also carries important deficiencies because its operational feasibility is based on the premise that participants are socially motivated and subject to the personal repercussions outside the functional scope of the system. This is primarily due to the existing level of trust that already exists between members. Although a social storage system

is built upon social incentives, peer pressure, etc., the discontinuous participation of social contacts, or even the abandon of the system, is intrinsic to the nature of social relationships.

In terms of storage, the intermittent participation of users impacts on the achievable storage QoS. That is, the unavailability of users limits the amount of time a file can be downloaded from the system (i.e., *data availability*) and how fast a file transfer can be performed (i.e., *transfer times*). Moreover, if some users are significantly more available than others, the workload supported across them may be unequally distributed (i.e., *load balancing*).

Surprisingly, despite the potential benefits that the social component may provide to distributed storage systems, very little attention has been paid to explore the storage QoS of these systems. In this thesis, by storage QoS we refer to the data availability, transfer performance and load balancing levels that a social storage system can provide to end-users. We believe that understanding the potential limitations of these systems is a necessary first step towards devising new techniques to improve their performance. This leads to the following challenge:

**Challenge 2: Explore the QoS of social storage systems.**

## 1.3 Research Questions and Contributions of This Thesis

In what follows, we aim to relate specific research questions with the aforementioned general research challenges. These unanswered questions are the motivation for the main contributions of this thesis.

*Challenge 1: Understanding the operation and performance of Personal Clouds*. Several research questions arise under this ambitious challenge. Since we mentioned that Dropbox decouples the management of file data and metadata, natural research questions in this sense are, for instance, *how does a Personal Cloud internally manage the metadata of clients?* And, *which is the extent of the required metadata infrastructure?* Despite the relevance of these questions, they are still unanswered in the literature.

Another interesting set of questions emerges if we want to understand the storage service of a Personal Cloud itself. That is, one can easily formulate questions of great interest like *which is the nature of the workload supported by a Personal Cloud? How users behave in this kind of services?* Or, *is there a relationship between these factors and the performance of the metadata service?* Moreover, it would be desirable to enable the research community to also address these challenges by making real workload traces of Personal Clouds publicly available [24].

On the other hand, very little is known about the transfer QoS of Personal Clouds. There is no public information about the control policies that Personal Clouds may enforce, as well as the factors impacting on their service performance. There are a variety of aspects that are relevant not only to end-users, but also to developers integrating third-party applications in a Personal Cloud platform: *does the geographic location impact of a user on the transfer QoS? Does the service exhibit variability along time?* And, *are Personal Clouds services reliable?*

Our contributions under *Challenge 1* are the following:

- **Our first contribution** is to unveil the internal structure of a global-scale Personal Cloud, namely UbuntuOne (U1), by describing its: *architecture*, core components involved in the U1 *metadata service* hosted in the datacenter of Canonical, as well as the *interactions* of U1 with Amazon S3 storage service, to which U1 outsources data storage.

- **Our second contribution** is to provide an extensive analysis of the back-end activity of U1 for one month, by means of tracing the metadata servers. Our analysis includes the study of the *storage workload*, the *user behavior* and the performance of the U1 *metadata store*. Moreover, based on our analysis, we also suggest improvements to U1 that can also benefit similar Personal Cloud systems.

- **Our third contribution** is to actively measure Personal Clouds through their REST APIs for *characterizing their QoS*, such as transfer speed, variability and failure rate. Our measurement is the first to deeply analyze many facets of these services and reveals new insights, such as important performance differences among providers, the existence of transfer speed daily patterns or sudden service breakdowns. Moreover, we demonstrate that combining *open APIs and free accounts* may lead to abuse by malicious parties. We also propose countermeasures to limit the impact of abusive applications in this scenario.

*Challenge 2: Explore the QoS of social storage systems.* First of all, this challenge calls for a deep understanding of the specific characteristics that govern the performance of social storage systems. Basically, we refer to fundamental questions such as "*are social storage systems a particular case of a P2P system?*" "*Are there differential factors that impact on their QoS?*" In fact, understanding the nature of these systems is the first step towards their analysis.

Following this line of reasoning, there are important issues that remain unexplored related to the feasibility of these systems in terms of storage QoS: *Can social storage systems provide*

*adequate data availability to end-users? And short transfer times?* And, *is the storage service unequally supported among participants?* These questions should be explored to discern the potential adoption of social storage systems as a practical alternative to cloud services.

As one can infer, the storage QoS also depends on the data management decisions implemented by a particular social storage system (e.g., *data placement*, *data redundancy management*). From the point of view of designers and practitioners, a natural approach to develop a social storage application could be to borrow data management techniques from large-scale distributed storage systems. Therefore, a critical question in this regard is, *"Are the traditional large-scale data management mechanisms suitable in a social storage scenario?"*

Perhaps, in a pessimistic scenario, a possible conclusion could be that it is difficult to achieve an adequate storage QoS in this setting —similar to what has been observed in large-scale systems [18]. In this case, *how can we improve the performance of a social storage system? Does improving the storage QoS have repercussions on the trust and privacy properties of the system?*

Our contributions under *Challenge 2* are the following:

- **Our fourth contribution** is to analyze the QoS of social storage systems in terms of *data availability*, *transfer performance* and *load balancing*. Conversely to P2P systems, social storage is highly affected by *availability correlations* and very *small groups*, which pose new challenges that remain unsolved to achieve an adequate storage QoS.

- **Our fifth contribution** is to understand the role of data management techniques in social storage system. Concretely, we evaluate the suitability of common approaches of *estimating data availability* when users are correlated. We demonstrate that these techniques are severely biased and this impacts on the *data redundancy calculation*. We propose a *history-based method* to calculate data availability tailored to heterogeneous and correlated availabilities. We also contribute by analyzing the impact of *data placement* and *transfer scheduling* policies in these systems.

- **Our sixth contribution** is to design a *hybrid architecture* to enhance the QoS achieved by a social storage system that combines user resources and cloud storage to let users infer the right balance between user control and QoS. This architecture is able to deliver such a balance thanks to the development of a new suite of data management algorithms. We also present an empirical evaluation of our architecture to study important operational aspects, such as the impact of the *social topology* on the storage QoS.

## 1.4 Outline of This Dissertation

This thesis is organized in two parts. According to the presented challenges, Part I contains the analysis and measurement of Personal Clouds, and Part II explores the QoS of social storage systems. In the following, we provide a summary of the thesis chapters:

**Chapter 2: Background**. This Chapter provides definitions and concepts that are required throughout the thesis, as well as an overview of a set of important personal storage systems.

**Chapter 3: State-of-the-Art**. This Chapter discusses the current literature in personal storage systems and illustrates the key differences of this thesis with previous works.

### Part I: Analysis of Personal Clouds

**Chapter 4: Dissecting a Personal Cloud Back-end**. In this Chapter we describe the internal operation of the UbuntuOne Personal Cloud, including the analysis of the storage workload, user behavior and metadata performance of this system.

**Chapter 5: Actively Measuring Personal Clouds: Analysis and Abuse**. This Chapter presents our measurement analysis of Personal Clouds REST APIs, jointly with the characterization of their QoS in various aspects. We also illustrate the potential abuse of these APIs and possible countermeasures.

### Part II: Exploring QoS in Social Storage Systems

**Chapter 6: Understanding QoS in Friend-to-Friend Storage Systems**. This Chapter provides an analysis of the storage QoS of purely distributed storage systems, as well as the impact of traditional large-scale data management techniques in this setting. We also devise cloud-assisted or hybrid architectures to improve the overall service performance.

**Chapter 7: Empirical Analysis of Social Cloud Storage**. Based on a battery of real experiments, this Chapter evaluates various aspects that impact on the storage QoS of a social cloud for storage, paying special attention to the role of the social topology.

**Chapter 8: Conclusions and Future Directions**. This Chapter presents the conclusions that ensue from this work and a variety of possible future research lines.

## 1.5 Selected Publications

This thesis is based on the following publications:

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas and Pedro García López. *FriendBox: A Hybrid F2F Personal Storage Application*. In Proceedings of 5th IEEE International Conference on Cloud Computing (CLOUD'12). Honolulu, USA, June 24-29 2012, Pages 131-138 [25].

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas and Pedro García López. *F2Box: Cloudifying F2F Storage Systems with High Availability Correlation*. In Proceedings of 5th IEEE International Conference on Cloud Computing (CLOUD'12). Honolulu, USA, June 24-29 2012, Pages 123-130 [26].

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas and Pedro García López. *Analysis of Data Availability in F2F Storage Systems: When Correlations Matter*. In Proceedings of 12th IEEE International Conference on Peer-to-Peer Computing (P2P'12). Tarragona, Spain, September 3-5 2012, Pages 225-236 [27].

- Adrián Moreno-Martínez, **Raúl Gracia Tinedo**, Marc Sánchez Artigas and Pedro García López. *FRIENDBOX: A Cloudified F2F Storage Application* (demo paper). In Proceedings of 12th IEEE International Conference on Peer-to-Peer Computing (P2P'12). Tarragona, Spain, September 3-5 2012, Pages 75-76 [28].

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas, Adrián Moreno-Martínez, Cristian Cotes and Pedro García López. *Actively Measuring Personal Cloud Storage*. In Proceedings of 6th IEEE International Conference on Cloud Computing (CLOUD'13). Santa Clara, USA, June 27-July 2 2013, Pages 301-308 [29].

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas and Pedro García López. *Cloud-as-a-Gift: Effectively Exploiting Personal Cloud Free Accounts via REST APIs*. In Proceedings of 6th IEEE International Conference on Cloud Computing (CLOUD'13). Santa Clara, USA, June 27-July 2 2013, Pages 621-628 [30].

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas, Aleix Ramírez, Adrián Moreno-Martínez, Xavier León and Pedro García López. *Giving form to social cloud storage through experimentation: Issues and insights*. Elsevier Future Generation Computer Systems. Volume 40, November 2014, Pages 1-16 [31].

- **Raúl Gracia Tinedo**, Yongchao Tian, Josep Sampé, Hamza Harkous, John Lenton, Pedro García López, Marc Sánchez Artigas, Marko Vukolic. *Dissecting UbuntuOne: Autopsy of a Global-scale Personal Cloud Back-end*. Submitted for publication.

Other publications completed during the thesis period are the following:

- **Raúl Gracia Tinedo**, Marc Sánchez Artigas, Pedro García López. *eWave: Leveraging Energy-awareness for In-line Deduplication Clusters*. 7th ACM International Systems and Storage Conference (SYSTOR'14). Haifa, Israel, June 10-12 2014, Pages 1-11 [32].

- **Raúl Gracia Tinedo**, Danny Harnik, Dalit Naor, Dmitry Sotnikov, Sivan Toledo, Aviad Zuck. *SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks*. 13th USENIX Conference on File and Storage Technologies (FAST'15). Santa Clara, USA, February 16-19 2015 [33].

# 2

# Personal Storage Systems: Background and Definitions

In this Chapter, we aim at providing the necessary concepts and definitions to properly understand the rest of this thesis. We first give some background on the operation and architecture of Personal Clouds, taking Dropbox as a paradigmatic example, which is essential to introduce the reader for Chapters 4 and 5. Second, we illustrate the principles and concepts behind a decentralized storage system, paying particular attention to social storage systems. This background is necessary for Chapters 6 and 7. In both cases, we overview a variety of existing systems to provide the reader with a big picture of the personal storage arena.

## 2.1 Overview of a Personal Cloud System

As we mentioned in Chapter 1, services like Dropbox, Box or SugarSync currently provide end-users and enterprises with online storage, file synchronization, sharing, as well as accessibility from a variety of mobile devices and the Web. To clarify the concept of Personal Cloud, we provide our own definition as follows:

**Definition 1 (Personal Cloud)** *The Personal Cloud is an online cloud service for personal information that enables users to store, synchronize and share data from a variety of devices and OSes. Moreover, Personal Clouds are a platform to deploy third-party applications that provide value-added services on users' data.*

In this sense, a natural question might be: *how does a Pesonal Cloud work?* To gain better understanding on the operation of a Personal Cloud like Dropbox, we believe appropriate the parallelism with a traditional file system.

In the context of a file system, a collection of user files consists of two separate information layers: *metadata* and *data*. On the one hand, a file's metadata contains information about the physical location of the file contents (*structural metadata* or *inodes*) as well as a variety of attributes describing a file's content type or application (*descriptive metadata*). On the other hand, a file's data refers to the actual contents or extents indexed by the structural metadata.

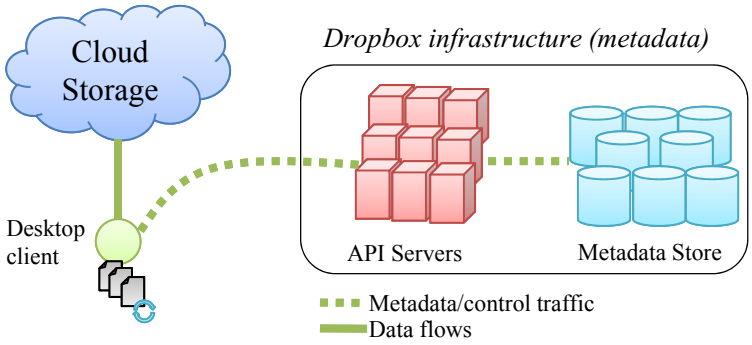



**Figure 2.1:** High-level architecture of Dropbox. We observe that Dropbox owns the metadata back-end whereas file contents are stored in Amazon S3.

Analogously, Personal Clouds like Dropbox, UbuntuOne (U1) or StackSync [34] are designed to separately manage the data and metadata of users files. We illustrate this next.

### 2.1.1 Architecture: The Case of Dropbox

Following the example of Dropbox, from an architectural viewpoint, this service exhibits a 3-tier architecture consisting of *clients*, *synchronization service* and *data store* [11] (see Fig. 2.1). As visible in Fig. 2.1, Dropbox decouples the management of file contents (*data*) and their logical representation (*metadata*). Thus, Dropbox only owns the infrastructure for the metadata service, which processes requests that affect the virtual organization of files in user volumes. The actual contents of file transfers are stored separately in a third party cloud provider (Amazon S3). One of the most important advantages of this architecture is that Dropbox can easily scale out the storage back-end thanks to the "pay-as-you-go" economies of cloud services [35], thus avoiding a heavy initial investment in storage resources.

In general, Personal Clouds provide users with 3 main types of access to their service: *Web access*, *desktop clients* and *REST (Representational State Transfer) APIs (Application Programming Interface)*. Perhaps, for users, the Web access is the most common and intuitive way of managing their data online. However, since a Personal Cloud Web access is less aligned with our research interests, it is not discussed further in this thesis.

In this sense, Personal Clouds enable other applications to interact with the service via REST APIs. These APIs make it possible for third-party applications to execute data management operations on files (`PUT`, `GET`, etc.) in user accounts. In fact, one can easily find similarities between these APIs (*files/accounts*) and the operation of object storage services (*ob-*

*jects/containers*) [36]. In the course of this thesis, we found that these API services are powerful abstractions that have not received enough attention from the research community. For this reason, we devote Chapter 5 to characterize and understand the operation of these APIs.

On the other hand, Personal Cloud desktop clients are very popular among users since they provide automatic synchronization of user files across several devices. To achieve this, desktop clients and the server-side infrastructure communicate via a *storage protocol*. In the case of Dropbox this protocol is proprietary. However, Drago et al. [11] inferred the messages exchanged between clients and servers. Similarly to the UbuntuOne protocol described in Chapter 4, the storage protocol of Dropbox is based on TCP and offers an API consisting of the *data management* and *metadata operations* that can be executed by a client. Metadata operations are those operations that do not involve transfers to/from the data store (i.e., Amazon S3), such as listing or deleting files, and are entirely managed by the synchronization service. On the contrary, uploads and downloads are, for instance, typical examples of data management operations.

From a research perspective, both desktop clients and REST APIs are the most interesting ways of accessing Personal Cloud services. In the next subsections, we illustrate technical aspects of both desktop clients and REST APIs, respectively.

### 2.1.2 Data Reduction Techniques in Desktop Clients

In general, it is important to the economic feasibility of Personal Clouds to reduce the expense on data outsourcing as much as possible. For this reason, desktop clients of most vendors include a number of *data reduction techniques*, that is, data management techniques intended to minimize the amount of data actually stored or transferred to the system.

Recent research works on Personal Clouds unveiled the techniques that are being currently applied by major vendors [24, 34, 37, 38]. In this section, however, our objective is to introduce the basic concepts and definitions related to these techniques:

**Chunking**: To ease the management of large data transfers, most desktop clients split files into *chunks* of smaller size, of the order of a few MBs. As one can infer, a desktop client in a domestic network has limited bandwidth and may experience failures while transferring large files. Thus, transferring files at the chunk granularity is an effective mechanism for providing resumable transfers in the presence of failures. As we will see, splitting files is also a intermediate step to enable data reduction techniques at the chunk granularity.

**Compression**: Compression is, perhaps, the most intuitive data reduction technique. It consists of reducing data redundancies within data streams, such as the repetition of sequences (*back-pointers*, *dictionaries* [39, 40]) or the skewed frequency distribution of bytes (*entropy coding* [41]). Algorithms like `zlib` [42] or `lz4` are commonly found in a variety of storage systems to improve performance and increase storage capacity [43, 44, 45]. In the case of desktop clients, data compression reduces the amount of data transferred and stored in the data store.

**Deduplication**: Data deduplication is a technique intended to avoid storing repeated content in a system [46, 47, 48]. Very succinctly, in a deduplication system, pieces of data are indexed and identified by its content (e.g., SHA-1 hash of a file) at either the chunk or file granularities. Upon the arrival of a new store operation, the index is checked to ensure that the new content to be stored does not exist already in the system. In the affirmative case, a logical link relates the new object with the existing content, avoiding thus storing again repeated data. Personal Clouds desktop clients that apply deduplication at the network level [11, 24]. That is, they avoid many file uploads detecting those ones that are already stored in the system; to this end, for instance, U1 desktop clients send the hash of a new file prior to the actual upload.

**Synchronization deferment**: It is common to find files in a Personal Cloud that are susceptible of experiencing successive modifications due to the user activity, such as text documents or source code files. Under this active update pattern, a desktop client may produce an intense network overhead if every change persisted in the file is uploaded to the cloud. To prevent this situation, many desktop clients wait for a certain period of time from the last file update before starting the synchronization process [24, 37].

**Delta updates**: When a file changes, a desktop client should reflect this change in the server-side. In a simple desktop client implementation, this can be done by uploading the whole file again to the server. However, as one can infer, this may induce high network overhead under successive changes on moderate or large files. To reduce this burden, Personal Clouds like Dropbox implement *delta updates*. That is, the desktop client detects the differences between the current and the immediate previous version and only uploads the fraction of the file that actually changed. In the case of Dropbox desktop clients, delta updates are provided by the `libsync` library [49] that, in turn, is based on the well-known `rsync` algorithm [50].

**File bundling**: In many cases users synchronize files at directory granularity with their desktop clients —i.e., synchronize a directory that may contain several files inside it. In this situation, desktop clients may be forced to perform a synchronization process for every file

contained in that directory. This can be inefficient, if we consider the synchronization protocol overhead (metadata, communication) compared with very small files. Therefore, some Personal Clouds enable desktop clients to transfer multiple small files as a single object to efficiently handle these situations.

Clearly, as previous works pointed out [11, 24, 37, 38], various vendors integrate different combinations of data reduction techniques in their respective desktop clients. A main design choice that Personal Clouds should face is whether to apply data reduction techniques at the *file or chunk levels*; for instance, Dropbox deduplicates data at the chunk level, whereas U1 implements file-level deduplication —as we will show in Chapter 4.

Furthermore, different combinations of these techniques may have disparate effects on both the design complexity of the system and the experienced savings. That is, implementing both data compression and deduplication at chunk-level may be complex as stated by Li et al. [24]. This may justify implementing data deduplication only at file level, if the potential savings are similar to the ones obtained by applying chunk-level deduplication. In our view, exploring these trade-offs is a potential vein of research related to this thesis.

### 2.1.3 Understanding Personal Cloud REST APIs

In addition to desktop clients, most Personal Clouds provide open REST APIs to make it possible for developers to create novel applications which use the information stored in user accounts. In this section, we will describe the functioning of these APIs and the procedure needed to register an application to enable its access to user storage. We will describe the complete process for Dropbox at the time of this writing (see Figure 2.2).

**Registering our application with Dropbox**. A Personal Cloud *application* is an authorized namespace within the Personal Cloud domains which enables REST API calls over user accounts. In Dropbox, these applications are either in production or development states. The former means that the application has been revised and approved by Dropbox, whereas the latter has limited features (development purposes). The Dropbox API incorporates OAuth [51] authorization mechanism to manage the credentials/tokens of applications and users granting access to these applications. Note that with a Dropbox application in development state, *a user is able to access up to* 5 *free storage accounts through the REST API*.

Dropbox provides 3 subdomains to support its API service: i) `dropbox.com` corresponds to the webpage, the place where users and developers perform manual interactions as explained later on; ii) `api.dropbox.com` is the subdomain against which applications perform

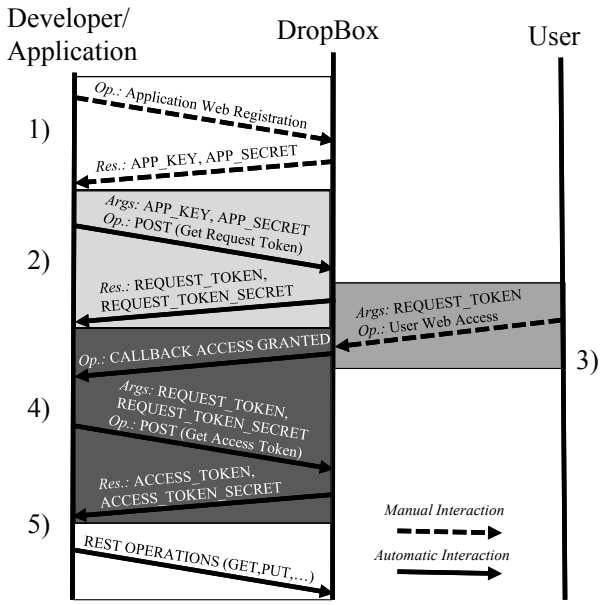



**Figure 2.2:** Registering an application in Dropbox to enable REST API file access to user accounts.

authentication and meta-data requests; and iii) `api-content.dropbox.com` is the subdomain where Dropbox handles API data management operations (`PUT`, `GET`). In the latter case, these operations are executed against Amazon S3, the storage back-end of Dropbox.

Now, we describe in general terms how to make an application operational in Dropbox. We denote the application to be registered as $A$, Dropbox as $DB$, and a user $U$ that permits the access to his storage space. The procedure is as follows.

First, a developer registers $A$ via $DB$'s webpage (`dropbox.com` subdomain), where $DB$ creates an *application token pair* that it will use to authenticate $A$. Second, $A$ asks for a request token to $DB$. Note that $A$ performs this step using $DB$'s API, and therefore, addressing a request via a HTTP `POST` message to the `api.dropbox.com` subdomain. As a result, $DB$ replies to $A$ with a *request token pair*. Thirdly, $U$ authorizes $A$ via $DB$'s webpage. Normally, $U$ is redirected to $DB$'s webpage by a link containing $A$'s *request token* as argument. With this information, $DB$ knows that user $U$ is giving access to $A$. In fourth place, once $U$ authorizes $A$, $DB$ automatically notifies $A$ about this event. $DB$ generates the *access token* for $A$, which grants access to $U$'s storage space. Next, $A$ performs an API call to $DB$ asking for the *access tokens*. Finally, $A$ performs storage operations against $U$'s account. The only requirement in each API call (`PUT`, `GET`) is to include the *access token* in the request.

We followed this procedure to enable user accounts to be accessed through REST APIs in order develop our analysis in Chapter 5.

### 2.1.4 Landscape of Personal Clouds

Next, we overview the current landscape of Personal Clouds[1]. Concretely, we aim at illustrating the available *deployment strategies* of a Personal Cloud as well as several *relevant functionalities* that big vendors provide nowadays. All in all, we believe that this section will help the reader to understand the variety, heterogeneity and extent of the Personal Cloud arena.

At this moment, we mainly referred to vendors that provide public services like Box, U1, Dropbox, etc. Nevertheless, a client may use a Personal Cloud system based on different *deployment strategies* depending on how the service *metadata and data* are located and managed:

- **Private or on-premise Personal Cloud**: In this deployment strategy, both data and metadata are stored on the client's infrastructure. In other words, a client administrates the Personal Cloud system that is deployed in his own storage infrastructure. Systems like `StackSync`[2] or `ownCloud`[3] allow private deployments on a client's infrastructure.

- **Public Personal Cloud**: Data and metadata are stored in a public storage provider such as Dropbox or Box. This is probably the most common case, where users delegate on a provider such as Dropbox the entire management of the service.

- **Hybrid Personal Cloud**: In this case, data is stored in a public cloud storage provider and metadata is kept inside the client's infrastructure. This allows clients to keep sensitive meta-information of their files under control, whereas raw data is encrypted and stored at a third-party storage service. A representative service of this category is `Egnyte`[4], that allows customers to work either with cloud or owned storage resources.

The non-public deployments are particularly important for organizations and companies, since many of them may be reluctant to outsource all their data to a public Personal Cloud. In this sense, Personal Clouds such as `Egnyte`, `ownCloud` or `StackSync` are currently providing on-premise or hybrid deployments, which are being increasingly adopted in the market [53].

Moreover, another interesting dimension that may serve to classify existing Personal Clouds lies on their functionalities. Concretely, we focus on the following operational areas: *storage*, *file synchronization*, *sharing*, *security* and *platform*. In what follows, we elaborate these features providing appropriate examples of real-world systems.

[1] This section in based on our work in Deliverable 2.1 of CloudSpaces project [52].
[2] `http://stacksync.org/`
[3] `http://www.owncloud.org/`
[4] `https://www.egnyte.com/`

**Storage**. Online storage is, perhaps, the most basic service that Personal Clouds. Often, to materialize this service, public vendors may opt to own the storage infrastructure (Google Drive, SkyDrive) or to outsource data storage (Dropbox, U1, SugarSync) to a third-party storage provider (e.g., Amazon S3, Carpathia Hosting).

Regarding storage, public Personal Clouds usually offer their services based on a *freemium* business model. In other words, a product is offered for free, but a premium product with advanced features is offered at a charge. Therefore, the storage quota offered for free accounts is an important feature that users consider. This aggressive market strategy has been widely adopted by providers such as Dropbox or Box.

Sometimes, Personal Clouds apply restrictions on the maximum file size, which can vary depending on whether the file is synced on the desktop application or uploaded through the web interface, or even the type of account (free/paid). For instance, Box enforces a file size limitation of 250MB for personal free accounts, whereas for business accounts this limit is 5GB.

To optimize storage and minimize the bandwidth consumed by clients, Personal Clouds may introduce data management techniques in desktop clients (see Section 2.1.2). Moreover, techniques like deduplication can be applied to different scopes (i.e., across all files in the system or only across a user's files), which presents a trade-off between storage efficiency and privacy [54]. Actually, the study of the implications of data management techniques in Personal Clouds is an active research path [24, 37].

**File Synchronization**. One of the key aspects of Personal Clouds is file synchronization (or *syncing*). We understand it as a two-way file synchronization, which means that a locally modified file is updated in each location this file is present. In addition, if a file is modified remotely, these changes will be automatically updated locally, with the purpose of keeping every copy of a file identical in all locations.

In this sense, some companies such as Cubby[1] or BTSync[2] also implement P2P file syncing, that is, the ability to keep two or more files identical in different locations without resorting to a central service. It allows companies to reduce the outgoing traffic from the data center, which translates in cost savings. It is also useful for users that want to store the same files on two or more computers avoiding the need to resort to the server.

Personal Clouds like Google Drive or Dropbox only sync files that are stored inside a specific folder created for that purpose. On the contrary, services like U1 or SugarSync, apart from

[1] `https://www.cubby.com`
[2] `https://www.getsync.com`

creating a default synchronized folder, enable a user to keep in sync multiple folders within his file system.

Another interesting feature is file versioning, which allows users to restore previous versions of a file after a number of changes. For those Personal Clouds implementing versioning, this can be done by limiting the version history to a maximum number of revisions to be kept in the system or for a specific period of time. For instance, Dropbox stores all versions of a file in the last 30 days —this may also vary depending on the account type.

**Sharing**. Sharing is an attractive feature that most of Personal Clouds provide, whether it is with users inside the service or with people outside the Personal Cloud. Internal sharing is usually offered as an integrated functionality in the user interface. Whereas public sharing is commonly offered as direct HTTP links that allow other users to access to files or folders. Anyway, the sharing infrastructure must provide users with mechanisms for managing access control of external users to their personal data.

Privacy-aware sharing is arousing interest in the Personal Cloud market. Currently, only SpiderOak is considered to implement a privacy-aware data sharing scheme. SpiderOak allows users to password protect all their Share Rooms[1] so that only the people they want to give access to their data can view or download their shared files. Each Share Room has its own private, secure URL so users can easily share them with only the people they want.

Real-time collaboration allows multiple users to edit a file at the same time. So users can see where in the document or file a particular editor is currently writing. Only Google Drive and SkyDrive let multiple users collaborate simultaneously on the same file from any computer. When someone makes changes to a document, the other person can see the changes in real-time and respond to the edits immediately.

**Security**. Personal Clouds must ensure that user data is not accessed by third-parties and only authenticated users are granted access. Some companies use standard authentication protocols such as OAuth [51] (Dropbox, Box), others opt for using their own mechanisms (SugarSync).

As a security measure, most vendors store user data encrypted. In general vendors provide *server-side encryption*, meaning that users delegate to the system the task of protecting their files and managing the cryptographic keys. As an alternative, vendors such as `SpiderOak`[2] and `Wuala`[3] implement *client-side encryption*, which allows users to encrypt their data before it

[1] `https://spideroak.com/engineering_matters`
[2] `https://spideroak.com`
[3] `https://www.wuala.com`

is transmitted to the server. So the user is responsible for managing the keys and the service provider is unable to decrypt his data, adding an extra layer of security.

Besides the fact of having the files secured when they are at rest in the server, it is also essential to assure their privacy when they are being transmitted to and from the server. To this end, these systems usually use HTTPS to communicate to their services either from the desktop application or other tools such as the REST APIs or the web interface.

As any other piece of software, the implementation of Personal Cloud systems are under licenses that grant rights and impose restrictions on the use of software. In general, most Personal Clouds are proprietary, so the source code cannot be freely accessed and reused. However, very few Personal Cloud implementations, such as `ownCloud`[1], `StackSync` and the desktop client of U1, are available for the general public and the end-user can further distribute or copy the software.

**Platforms**. Many Personal Clouds are currently providing raw storage services through web interfaces, acting like pseudo Infrastructure-as-a-Service (IaaS) providers. Thus, to access user data from an external application, Personal Clouds must implement an API, which 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, also known as REST API. These APIs are implemented by most public vendors, such as Dropbox, SugarSync, SkyDrive and Box.

An alternative way to allow external access to user data is through the WebDAV protocol. It provides a framework for users to create, change and move their documents. Most current operating systems (OSes) provide built-in support for WebDAV. This approach, however, has gauged less adoption, and only Cubby and ownCloud implement WebDAV data access.

Being able to access to users' stored data from a web browser is an essential functionality. Web interfaces typically allow users to manage their files (move, delete, upload, download, etc.) and access to extra tools such as generating public links. Additionally, most Personal Clouds are integrating their services across a multitude of OSes (Linux, Windows, Mac) and devices (Android, iOS) to reach large amounts of users.

[1] `https://owncloud.org/`

## 2.2 Definition of a Social Storage System

In this section, we provide a comprehensive overview of the principles and characteristics that give form to the social storage paradigm. Moreover, we introduce concepts related to our view of the storage QoS of a social storage system, as well as the most important data management techniques in this regard. We believe that this background is essential to understand Part II of this dissertation.

### 2.2.1 Principles: Decentralization and Social Component

In essence, a *social storage system* can be understood as a particular case of a *decentralized storage system*. For this reason, we believe adequate to provide the following definition:

**Definition 2 (Decentralized Storage System)** *A decentralized storage system integrates storage nodes into a single and uniform data storage service that applications and users can access through a communication network [55].*

In such a kind of systems users donate part of their local storage resources in exchange of a share in the on-line storage service [3, 5, 56, 57]. The system is thus in charge of transparently manage the data stored given the *amount and stability* of the resources that users contribute. In fact, the unavailability of user resources, namely *churn*, is one of the main impediments for the correct operation of decentralized storage systems [58, 59, 60].

Particularly, in a *social storage system*, the exchanges of resources among users are *bounded by their social relationships* [20, 61]. This means that, conversely to traditional large-scale systems, participating users are *socially motivated* to do so.

**Definition 3 (Social Tie or Friend)** *In a social storage system, two storage nodes are social ties or friends of each other if they establish a symmetric link between them to share storage resources based on trust, social or real-world relationships.*

Therefore, it is important to note here that a social storage system is not crowdsourcing [62] or volunteer computing [63] as the social relationships are generally symmetric. In other words, members are more or less seen as equals who provision resources to benefit from sharing, whereas crowdsourcing or volunteer systems operate in the master-worker model where work flows in one direction, which does not in itself constitute sharing. That is, the interactions that govern the resource contribution in a social storage system are similar to those found in the traditional P2P literature [3, 15, 16, 19].

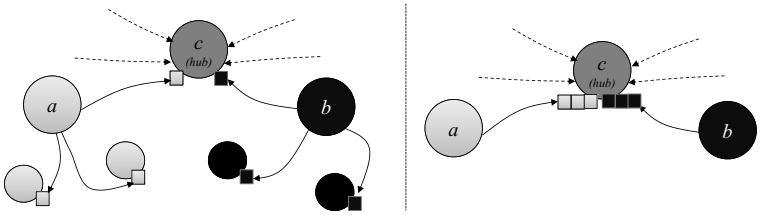



**Figure 2.3:** The contribution of each node to the system may depend on the social topology.

In a social storage service, the topology of the underlying friendship graph plays a central role in the contribution asymmetry and, consequently, in the operational system requirements.

**Definition 4 (Social Graph or Topology)** *The social graph or topology governs the interaction between pairs of users and determines the storage resources to be contributed by them.*

Although users with many friends have more chances of storing their data with higher availability, they may possibly have to donate more disk space to socially reciprocate a larger number of friends. This is especially visible for those users with higher degrees, usually called *hubs*, whose level of contribution may be very high for comparatively little benefit.

From a global perspective, it is not hard to imagine that the degree distribution of the social graph is one the main factors impacting the system's operation. To better understand this, pretend that two users, say $a$ and $b$, want to store 3 data blocks each (see Figure 2.3). Also, assume that they have a friend in common, say $c$. Depending on the number of friends, then $a$ and $b$ will store more or less data blocks in $c$. If $a$ and $b$ had two additional friends, then $c$ would need to store only 2 data blocks, one from $a$ and one from $b$. However, if $c$ was the only friend of $a$ and $b$ in the system, $3 \cdot 2 = 6$ blocks would be allocated to $c$. This shows the importance of social connectivity on contributed storage, specially for hubs.

Real measurements of social networks [64, 65] show that while clustering is very high, the presence of hubs is characteristic of social interaction. Understanding the influence that graph properties have on the extent of storage contribution is crucial to decide to *what extent the asymmetry in contributory levels requires control and regulation* [23, 66].

As we will show in this thesis, both the *unavailability of user resources* and the *structure of the social graph* are key elements to the QoS of achieved by a social storage system. In the following, we define the properties that constitute storage QoS in Part II of this dissertation.

### 2.2.2 Storage QoS in a Social Storage System

We already described that social storage systems are built upon the resource contribution of users integrating the service. Moreover, we also mentioned that the underlying social topology plays an important role on the operation of the system, specially regarding the symmetry of resource contribution among users.

Nevertheless, irrespective of the nature of "social storage", users expect from a storage system to provide an adequate Quality of Service (QoS). In other words, social storage systems should provide reliable and fast off-site storage in order to be widely adopted by end-users.

In Part II of this dissertation, we focus on evaluating the QoS that a social storage system may achieve depending on various operational and structural aspects. To set an appropriate ground for such evaluation, in what follows we provide an overview of what we understand by QoS in a social storage system, i.e., *data availability*, *load balancing* and *transfer performance*.

Certainly, our QoS framework is focused to understand the system's performance in the *short term*. However, we can find other metrics that are necessary to evaluate the long run operation of a storage system, such as *data durability* [55]. We defer the study of data durability for future work, since permanent departures are less frequent among in a social storage system that in large-scale scenarios [20, 61].

**Data Availability**. Formally, we can define data availability as follows:

**Definition 5 (Data Availability)** *In a decentralized storage system, data availability can be defined as the fraction of time a user is able to retrieve a data object from its remote location. This applies irrespective of whether a data object is stored as a single piece or split into blocks.*

Although the concept of social storage is built upon social incentives, peer pressure, etc., the discontinuous participation of social links is intrinsic to the nature of interactions in online social networks (OSNs). In terms of storage, intermittent participation means that data may be subject to recurrent periods of unavailability, which may be long depending on the activity pattern between pairs of users. Unlike commercial cloud storage systems like Amazon S3 and Microsoft Azure that offer high data availability (e.g., 0.999), the availability of any particular file cannot be guaranteed in a decentralized social storage system. At any given time, *data availability depends on the number and availability of the social links with whom content is shared*.

As we will see later on, users will need to generate *data redundancy* (e.g., parity blocks, replicas) to mask the unavailabilities of nodes. In fact, to model and improve the data availability achieved by a social storage system is one of the contributions of this dissertation.

**Load Balancing**. Regarding load balancing, we propose the following definition:

**Definition 6 (Load Balancing)** *Given a group of storage nodes, the load balancing metric captures the (in)equality of supported work across nodes within the group.*

Load balancing is critical to the feasibility of a distributed social storage system. In terms of storage, load should be balanced across nodes regarding *inbound bandwidth*, *outbound bandwidth* and *storage space*. The lack of load balancing is directly translated into resource contribution unfairness among users, which impacts, for example, on the storage QoS in terms of transfer performance.

Perhaps, in the absence of regulatory mechanisms, *the underlaying social topology may influence the system's load balancing in terms of resource consumption* (storage, bandwidth). In this sense, to fully understand the interplay between the social topology and the system's load balancing, the local point viewpoint of a single node is not sufficient; one of our interests in this thesis is to study the system from a global, network-wide perspective.

**Transfer Performance**. We provide the following definition of transfer performance:

**Definition 7 (Transfer Performance)** *The transfer performance metric refers to the speed at which a single data object can be stored or retrieved from the system.*

For a storage service, providing fast access to data is a paramount concern, irrespective of whether it is measured in terms of *transfer times* or *bandwidth* (e.g., Mbps). Analogously, in a distributed social storage system, users will expect to store or retrieve files from the system with an acceptable performance. In this sense, the completion of a transfer can be interpreted differently depending on whether it is an upload or download transfer.

On the one hand, when a user uploads a file to the system, he will probably need to upload redundant data blocks of that file as well. As we already mentioned, the main purpose of introducing data redundancy is to alleviate data unavailability in the presence of intermittent user connectivity. On the other hand, a download transfer will be considered as completed when the necessary number of data blocks to reconstruct the original file have been downloaded. As one can infer, uploads will generally take longer transfer times than downloads.

In a social storage scenario, there are several potentially entangled factors that may influence on the performance of transfers; for instance, the *number and availability* of a user's social ties and the amount of generated *data redundancy*. Thus, guaranteeing acceptable transfer performance in a social storage system without dedicated resources poses a complex challenge, which is object of study in Part II of this dissertation.

At this point, we described the main properties that constitute our framework to evaluate the QoS of a social storage system: *data availability*, *load balancing* and *transfer performance*. In the following, we overview the data management techniques that enable a storage system to maximize the storage QoS depending on the system's conditions.

### 2.2.3 Data Management Techniques

Data management is a central operational point of any decentralized storage system. When appropriately designed, data management techniques allow a storage system to efficiently replicate data to increase data availability or to improve load balancing by properly assigning data to storage nodes, among other aspects. As one can easily infer, data management techniques may considerably differ depending on the system where they are devised for.

In what follows, we focus on data management techniques in the scope of decentralized storage systems, since social storage systems belong to this category. Note that this section aims at introducing fundamental concepts for a storage model that will be further developed in Part II of this dissertation.

#### Data Placement: Assigning Data to Nodes

In a decentralized storage system data files are normally split into *data blocks* to facilitate transfers of large files. Upon a store operation of a file, its data blocks need to be stored across a subset of storage nodes from the total $\mathcal{F}$. In other words, each data block is *assigned* to a storage node in order to be persisted. This inherently implies that the system should take a decision about *which blocks are assigned to which storage nodes*.

To analytically represent this decision, we denote by $b_f \in \{0, 1, \ldots, n\}$ the number of blocks assigned to a node $f \in \mathcal{F}$. Note that this representation allows the system to assign more than one block to a single storage node, which may be inevitable depending on the available number of storage nodes. An assignment is represented as a vector $\vec{\mathbf{b}} = \left(b_1, \ldots, b_{|\mathcal{F}|}\right)$, where the $i$th position is the number of blocks $b_i$ stored at the $i$th node. As one can infer, this assignment vector can be managed in order to implement a certain *placement policy* in the system. In other words, the system can take special care of the relationship among storage nodes and data blocks to increase performance.

To illustrate this, imagine that we aim to maximize the data availability of files within the system. Given this objective metric, we can implement a simple greedy policy, namely *availability proportional*, that proportionally stores more data blocks on the most available nodes.

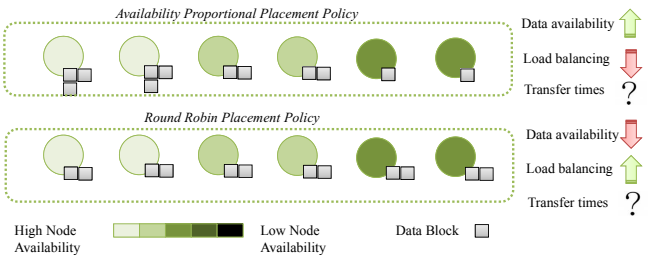



**Figure 2.4:** Storage QoS trade-offs that depend on the data placement policy.

Thus, at the moment of storing the blocks belonging to a new file, the vector $\vec{\mathbf{b}}$ will contain the most available storage nodes. Also, in a situation where the number of blocks is larger than the number of nodes, more blocks would be proportionally placed at highly available nodes. This can be observed in Figure 2.4.

However, despite the potential gain in data availability, one can easily understand that this strategy tends to overload highly available nodes, exhibiting poor load balancing. In turn, this placement policy may have negative consequences on the storage performance of the system (e.g., concurrent transfers), or it may motivate highly available nodes to leave the system due to lack of incentives.

On the other hand, a policy that does not distinguish the behavior of nodes, such as *round robin data placement*, will probably provide better load balancing. However, given the same amount of data redundancy, this policy will probably achieve lower data availability.

Therefore, a data placement policy is a pivotal element to the correct operation of the storage system, and as such, it poses QoS trade-offs that should be carefully considered.

### Data Transfer Scheduling

Upon a file transfer operation, the system should start transferring data to or from the selected storage nodes. In this sense, we term *data transfer scheduling* to the decision problem that manages the *order in which transfers occur over time*.

However, before going any further in the concept of data transfer scheduling, it should be clearly stated what is meant by the term *scheduling*. We provide some definitions to clarify this concept.

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*

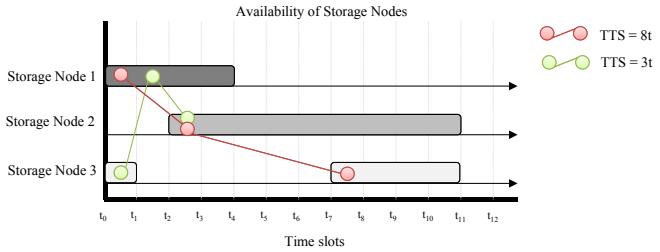



**Figure 2.5:** Relevance of data transfer scheduling decisions on file transfers in the presence of intermittent node availabilities.

*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).

We define two important concepts to understand the efficacy of scheduling policies further on in this thesis:

**Definition 8 (Minimum Time To Schedule, MTTS)** *The minimum time to schedule (MTTS) is the time a node requires to transfer all blocks of a single schedule assuming ideal conditions, i.e., the MTTS only depends on the amount of data to transfer and the current bandwidth capacity.*

**Definition 9 (Optimal Time To Schedule, OTTS)** *The optimal time to schedule (OTTS) refers to the shortest TTS assuming the dynamic participation from friends.*

To compute the OTTS it is necessary to explore all the possible scheduling combinations with *prior* knowledge of the exact ON/OFF pattern of each storage friend, which is not feasible in practice[1]. However, the OTTS will be very useful as a baseline in our evaluations.

Once provided the necessary technical definitions, let us draw an illustrative example of how a data transfer scheduling policy may impact on the storage system. Imagine a storage system with 3 storage nodes that alternate between on-line and off-line states, as described in Figure 2.5. Moreover, let us assume that we are willing to store a data object in the system, and we split it into 3 data blocks that will be stored at distinct storage nodes. To simplify the example, the block transfer time (BTT) is 1 time slot and blocks are transferred sequentially.

At time $t_0$, the system should take an important scheduling decision: whether to start transferring the first data block either to $SN_1$ or $SN_3$, since both are available. If the system starts

[1]This time is computed by modeling the scheduling with dynamics friends as a *flow network*, and using binary search to find the *shortest-time max-flow* solution similar to [67].

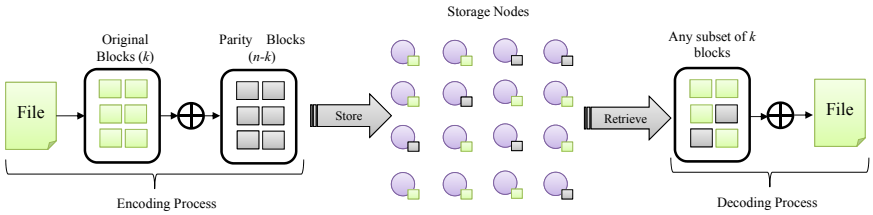



**Figure 2.6:** Example of generating, storing and retrieving a file from a distributed storage system making use of erasure coding.

transferring the first data block to $SN_1$, the second block would not be able to be transferred to $SN_2$ until $t_2$. Even worse, the last block transfer to $SN_3$ will be deferred until time $t_7$, when it is on-line again. This means that the time to schedule (TTS) of this scheduling plan would be $TTS = 8$ time units. As one can infer, this result is significantly worse than the optimal and minimum time to schedule values —in this particular example, MTTS=OTTS=3 time slots.

On the other hand, if the system starts transferring the first block to $SN_3$ the schedule would not be affected anymore by the disconnection pattern of that node. Subsequently, the second data block would be transferred to $SN_1$ at time $t_1$ and the last one would be transferred to $SN_2$ at $t_2$. In this case, the system performed the best possible schedule resulting in TTS=3 time slots. Therefore, we clearly observe that data transfer scheduling is a relevant aspect regarding the QoS of a decentralized storage system.

### Data Redundancy

To maintain the desired level of data availability, it must be carefully decided the degree of redundancy. Object *replication* is, perhaps, the simplest way of producing data redundancy, being suitable for storage of small objects that are accessed frequently. However, other redundancy schemes based on *erasure codes* can reduce the storage and communication costs compared to replication [59, 68].

Concretely, in Part II of this thesis we make use of Reed-Solomon codes (RS) [69]. Given a data object of size $B$, a RS$(n,k)$ code partitions the data object into $k$ equal-sized fragments, each of size $B/k$ bits. These $k$ fragments are then encoded to a set of $n = k + h$ redundant blocks. Since this code is a maximum distance separable (MDS) code, the stored object can be reconstructed from any $k$-subset of redundant fragments. The consequence of this property is that a RS$(n,k)$ code can tolerate the loss of any $h = n - k$ blocks with a redundancy ratio of only $n/k$.

In Figure 2.6 we illustrate a simple example of the practical use of erasure codes in a distributed storage system. That is, if we split a file into $n = 12$ blocks so that any $k = 6$ blocks

suffice to reconstruct the original file, we can tolerate 6 failures with an storage-space overhead of only 100%. If we had used replication instead, we would have needed 7 replicas to achieve the same level of fault tolerance, yielding an storage-space overhead of 700%. The use of coding is thus highly desirable in this environment where the nodes storing the data will not be available at all times.

### 2.2.4 Existing Social Storage Systems

Next, we provide an overview of some important social storage systems. Apart from helping the reader to know the most relevant systems in this regard, we believe that this section will also give a sense on the potential applicability of our contributions in Part II of this thesis.

**Friend-to-Friend (F2F) Storage Systems**. F2F storage systems originally emerged as an alternative to traditional P2P storage systems. Although many relevant systems and designs have been proposed in the literature (e.g., OceanStore [56], PAST [3], Farsite [16], etc), P2P storage systems suffer from inherent drawbacks that are hard to overcome. First, the instability of peers [18] makes difficult and costly to provide high data availability. Furthermore, despite important efforts [19], the existence of free riding and selfish behaviors complicates the efficient management of the existing resources. Moreover, many users are still reluctant to store their data in unknown hosts due to trust and security reasons.

Instead of interacting with random nodes, the main strength of the F2F paradigm lies on building storage interactions upon trust, social or real-world relationships. Thus, F2F systems assume that the existence of social connections among participants gives a node reasons to trust that these contracts will be respected, and behave accordingly [70]. This makes the whole system to operate in a more favorable scenario, reducing the overhead of dealing with high rates of free-riding and malicious behaviors. Next, we briefly describe several F2F systems:

- **BlockParty** is a distributed backup application, originally presented in the pioneering work of Li and Dabek [61], that provides an off-site backup service for home users. As other systems, BlockParty breaks the data to be backed up into chunks and distributes each chunk to one or more neighbor machines depending on the desired replication level. To ensure storage balance, the BlockParty software at a node dedicates at least as much space to storing other nodes' backups as the node wishes to use on other nodes.

- **Friendstore** [20] is a cooperative backup systems that also relies on trust relationships among participants to ameliorate the impact of free-riding and malicious behavior that

may harm the operation of the system. Friendstore puts special emphasis on keeping track of the limited and heterogeneous resources of users (bandwidth, storage space) to optimize backup operations, which are potentially resource consuming. For instance, they reduce storage consumption by applying a simple coding scheme.

- **Crashplan**[1] is an offsite backup service that enable users to store data in the cloud or in other remote locations, such as other computers in a user's network of friends and family. Although the technical details are not public, Crashplan can be seen as one of the first commercial products offering F2F storage.

**Distributed On-line Social Networks (DOSN)**. Online social networks, such as Facebook, Google+ and LinkedIn, are becoming a predominant service today. Catering for people of all ages, gender and class, social networking services have become the primary means of communication between friends, family and colleagues. However, major social networks are currently operated by private companies that control the data of users, which represents a potential threat for privacy and security [71, 72, 73].

As a reaction to the risk that a centralized social network architecture represents for users' privacy, researchers started to devise decentralized social networking systems [74, 75, 76, 77, 78]. As any other decentralized system, DOSNs integrate the spare resources contributed by users (bandwidth, storage) to provide a social networking service. Moreover, in terms of storage, a DOSN must take care all the operational aspects of any decentralized storage system, including data placement, failure detection and redundancy, among other aspects.

- **PeerSon** [77] is a pioneering DOSN and its design is built upon three pillars: *encryption*, *decentralization*, and *direct data exchange*. In PeerSon, data is stored encrypted for keeping users' data private, and decentralization —by means of leveraging an underlying P2P overlay— provides independence from OSN providers. Authors state that decentralization makes it easier to integrate direct data exchange between users' devices into the system. This, in turn, allows users to use the system without constant Internet connectivity, leveraging real-life social networking and locality.

- **Supernova** [78] represents an evolution of the PeerSon design. Supernova introduces the concept of super-peers in the system in order to improve data availability and deal with heterogeneity in a more effective manner. Moreover, super-peer resources can be

[1] `https://www.code42.com/crashplan/`

shared across users. The share a user obtains from a super-peer depends on his behavior, which is a mechanism to incentivize user cooperation.

- **SafeBook** [76] is a decentralized and privacy-preserving online social networking site. This system relies in two design principles, *decentralization* (P2P substrate) and exploiting *real-life trust*. In this setting, SafeBook integrates various privacy and security mechanisms to provide data storage and data management functions that preserve users' privacy, data integrity and availability.

- **Vis-à-Vis** [79] is a decentralized framework for OSNs based on the privacy-preserving notion of a Virtual Individual Server (VIS). The main idea behind Vis-à-Vis is to make use of existing cloud infrastructures to sustain a social networking site (running VIS instances), being owned by the users in the system instead of a single OSN company. Technically, Vis-à-Vis is self-organized into overlay networks corresponding to social groups and puts especial emphasis on preserving privacy of user location information.

**The Social Cloud**. The advent of social networks and digital relationships creates new opportunities to spur the adoption of socially oriented computing. One representative example of this trend is the concept of "social cloud" as a means of facilitating resource sharing by utilizing the relationships established between members of a social network [22, 23, 66].

A social cloud leverages preexisting trust relationships between users to enable mutually beneficial sharing. This facilitates long term sharing with lower privacy and security requirements than those that are present in traditional cloud environments. For the time being, the cloud accrues massive amounts of private information to provide for instance highly targeted advertisements. Not surprisingly, security breaches, poor judgment, or even the lack of judicial oversight leaves users vulnerable. In this sense, the "social cloud" represents a new form for the users to retake control of the cloud service, avoiding to be tracked or give personal information against their will, or in a way in which they feel uncomfortable. In fact, as pointed out by S. Pearson [80], one of the "top six" recommended privacy practices for cloud systems is to maximize user control, which is one of the outstanding feature of the "social cloud".

Another distinguishing feature of the "social cloud" is that the network comes first. It is not a cloud or middleware extended with a social network; rather, it is a social network extended with cloud functionality. Users form the basic infrastructure and share resources around their social graphs. Such an organization brings out many benefits. For instance, one of those advantages is usability, since the interface and tools for resource sharing are already

familiar to users. Another one is that it allows users to maximize the control of the cloud service by letting users choose how their resources will be used. Giving users the control over their personal information and resources engenders trust, but this can be difficult in a cloud computing scenario. This feature is very interesting for the adoption of the "social cloud", as it permits users to define a series of preferences for the management of their personal data, and take account for that, among other advantages.

# 3
# State-of-the-Art

This Chapter aims at bringing the reader closer to the concrete research problems that motivate this thesis. For the sake of clarity, we organize this Chapter into sections according to the different topics treated throughout the remaining chapters.

Moreover, in each section, we provide a complete discussion of the existing related work, comparing the achievements of this thesis to the state-of-the-art. This will ease the reader to discern the context and the extent of our contributions.

## 3.1 Dissecting a Personal Cloud Back-end

### 3.1.1 Internal Operation of Personal Cloud Services

The *internal operation and infrastructure* of a Personal Cloud remains quite unknown. That is, there is little information about the internals of these systems in order to understand the management of user metadata or how the client notification system works, to name a couple of examples. Although this can be understandable from the viewpoint of providers, such a lack of information limits the scientific contributions of the research community in this field [24].

In this sense, Drago et al. [11] presented an external measurement of Dropbox in both a university campus and residential networks. Authors unveiled that the Dropbox service is composed of metadata servers, notification servers and a storage back-end, which is actually Amazon S3. Although this is a significant contribution, authors only provided a high-level perspective of the Dropbox's architecture. In fact, from external vantage points, it is almost impossible to fully understand the internal operation of a cloud service.

Thus, authors in [11] did not answer questions like *Does Dropbox store metadata in a relational database or not?* In the affirmative case, *which is the infrastructure needed to scale out metadata storage? Which is the performance of this approach of managing metadata?* Similar questions may raise regarding other architectural elements of a Personal Cloud.

Very recently, Dropbox released a sales-oriented document explaining the security features of the service. In that document, authors state that Dropbox stores metadata in a MySQL-backed database using sharding and replication [81]. Similarly, Box [82] also explained how

they scaled a MySQL cluster to store user metadata. However, although some vendors are currently providing clues about the way they manage metadata, more work is needed to understand the trade-offs of these designs as well as to collect real system traces to foster research in this field.

**Progress beyond state-of-the-art**: In Chapter 4, we describe the internal architecture and infrastructure of a global-scale Personal Cloud, namely UbuntuOne (U1). Concretely, we provide technical details of the metadata management system of U1 and its performance, as well as the explanation of other internal elements (e.g. notifications, data model) and the interactions with a cloud storage provider (Amazon S3) to outsource data storage. To the best of our knowledge, this thesis is the first to illustrate the internals of a big Personal Cloud player in such level of detail. We also make available the collected traces for the research community.

### 3.1.2 Passive Measurements of Personal Clouds

Another interesting research issue related with the operation of Personal Clouds is to analyze the supported *storage workload and how users produce it*. This is specially interesting due to the specific usage that users may present in such application [24, 83]; a deep understanding of such usage may lead to the development of optimized data management techniques in this scenario [38] and better system designs [34, 84].

In this regard, up to date there have been some attempts to model the storage workload and user behavior of these systems. For instance, Drago et al. [11] analyzed the behavior of users in Dropbox, mainly in university campus scenarios. This work includes macroscopic workload metrics (e.g., daily traffic, characterization of storage/metadata flows), as well as remarkable aspects of the behavior of users (e.g., number of devices, sharing). A study of a similar flavor but of smaller scale has been also conducted for Microsoft SkyDrive [85].

Liu et al. [86] inspected in depth the workload patterns of users also in the context of a storage system within a university campus. This work concentrates on macroscopic storage workload metrics and the type of requests, as well as the differences in access patterns of personal and shared folders. In Chapter 4 we also analyze macroscopic metrics of U1, as well as other aspects (user behavior, metadata store performance) not discussed in [86].

Authors in [83] are particularly interested on modeling the behavior of user connectivity. This work provides valuable insights regarding the nature of connection patterns of users, as well as statistical observations to model the session behavior of users in this type of systems.

| | **Chapter 4** | Drago et al. [11] | Li et al. [24] | Liu et al. [86] | Gonçalves et al. [83] |
|---|---|---|---|---|---|
| Back-end operation | **Yes** | No | No | **Yes** | No |
| Architecture | **Yes** | **Yes** | No | **Yes** | No |
| Daily workload | **Yes** | **Yes** | No | **Yes** | No |
| Analysis of files | **Yes** | **Yes (Chunks)** | **Yes** | **Yes** | No |
| User behavior | **Yes** | **Yes** | **Yes** | No | **Yes** |
| Metadata store performance | **Yes** | No | No | No | No |
| Client sessions | **Yes** | **Yes** | No | No | **Yes** |
| Measurement Scale | 1.17M users | 35K users | 153 users | 19K users | 22K users |
| Methodology | **Server side** | **Vantage points** | **Client monitoring** | **Server side** | **Vantage points** |

**Table 3.1:** Features of Chapter 4 compared to related works.

By monitoring a reduced number of users, Li et al. [24] measured the behavior of dozens of desktop clients from various vendors in several universities. Their main objective was to understand the data reduction and management techniques implemented in Personal Cloud desktop clients (compression, deduplication). A similarity of this work with Chapter 4 is that we also focus on the workload generated by desktop clients. Additionally, our measurement includes many aspects not studied in [24], e.g., burstiness of user operations, user/system workload metrics, DDoS attacks, among others.

To ease the comparison with main prior works, we suggest the reader to inspect Table 3.1.

However, the main shortcoming of previous efforts is that they analyze specific user communities, which may be not representative enough of the global usage of a Personal Cloud service. Ideally, to fully understand the workload of these services it would be necessary to capture the activity of a large fraction of users, or even the whole population, which becomes highly impractical considering proprietary and global-scale services.

**Progress beyond state-of-the-art**: Also in Chapter 4, we captured the activity of the whole user population of U1 for one month by means of tracing its back-end servers. This valuable source of information enabled us to provide an extensive characterization of the storage workload and user behavior in U1. In our view, this contribution extends the state-of-the-art on Personal Clouds measurements.

In summary, Chapter 4 illustrates the internal infrastructure and operation of a global-scale Personal Cloud (U1) in a high level of detail. Moreover, we contribute an extensive study of the workload and user behavior of the entire user population of U1 for one month; to the best of our knowledge, this is the first analysis of a Personal Cloud at this scale.

## 3.2 Measurement and Abuse of Personal Cloud REST APIs

### 3.2.1 Personal Cloud Active Measurements

The performance evaluation of cloud storage services [9] is an interesting topic with several papers appearing recently. Hill et al. in [87] provide a quantitative analysis of the performance of the Windows Azure Platform, including storage. Bergen et al. in [88] execute an extensive measurement against Amazon S3 to elucidate whether cloud storage is suitable for scientific Grids. Similarly, [89] presents a performance analysis of the Amazon Web Services.

The problem is that these works provide no insights regarding Personal Clouds. In fact, despite their commercial popularity, only few research works have turned attention to measure the performance of Personal Cloud storage services [11, 24, 37, 38, 90].

Naturally, in a complex architecture such as a Personal Cloud, the service performance can be measured at various stages. We particularly focus on the *client-side transfer performance* of the service. In such a competitive market, this perspective of performance analysis can provide useful information about the quality and experience of clients interacting with a service. Therefore, a client may interact with a Personal Cloud making use of *Web/mobile clients*, *desktop clients* and *REST APIs*.

As a part of their study, Drago et al. [11] briefly addressed user interactions with the Dropbox's *Web interface* in campus environments. However, since this type of access is the least innovative, it has attracted less attention from the research community.

For *desktop clients*, the first analysis of Personal Cloud storage services we are aware of was [90]. Hu et al. [90] compared Dropbox, Mozy, Carbonite and CrashPlan storage services. However, their analysis was rather lightweight and only scratched the surface; the metrics provided in [90] are only backup/restore times depending on several types of backup contents. They also discussed potential privacy and security issues comparing these vendors.

In this line, authors in [37] present a complete framework to benchmark Personal Cloud desktop clients. One of their valuable contributions is to design a benchmarking framework for comparing the different data reduction techniques implemented in desktop clients (e.g, file bundling, compression, deduplication).

Similarly, Li et al. in [24] examined the performance, specifically in terms of network overhead, of various Personal Cloud desktop clients. Other works inspected the performance issues of active update patterns in synchronized files [38]. Anyway, we believe that under-

standing the interplay between data management techniques in desktop clients and service performance is still requiring research efforts.

Unfortunately, to the best of our knowledge, the analysis of Personal Cloud *REST API services* has not been addressed by the research community, mainly because desktop clients attracted most recent works in this field. However, we believe that understanding their performance may be of interest for developers integrating applications in Personal Clouds or when clients use Personal Clouds as IaaS providers. Moreover, a proper characterization of their performance may be used in modeling and simulation environments, for example.

**Progress beyond state-of-the-art**: As a major contribution of Chapter 5, we provide an active measurement of various Personal Cloud vendors. This measurements characterizes the transfer performance, variability and failures of these services, among other aspects. In our view, our work extends the state-of-the-art knowledge on how Personal Clouds behave by inspecting in depth their REST APIs.

### 3.2.2 Exploitation of Personal Clouds

In Chapter 5, apart from contributing our analysis, we argue that these services may be abused through their REST APIs over free accounts. For this reason, we found specially interesting recent efforts regarding security in Web Services [91]. In this sense, the authors of [13] observe that the current lack of integrity controls at the data level in API REST Web Services could result in profound problems regarding data integrity. Other works such as [92] exploit specific vulnerabilities on the authentication mechanisms employed in Amazon EC2 and Eucalyptus cloud control interfaces.

The abuse of cloud services is currently a relevant research concern. As described in [93], one of the major risks of cloud computing is its "abuse and nefarious use" by malicious parties (e.g. botnets, software exploits). In this line, few works have analyzed the impact of external attacks on cloud services and applications. For instance, authors in [94, 95] investigate the potential vulnerabilities of the cloud computing model, which could be exploited from fraudulent resource consumption of any Internet connected host.

Directly related to Personal Clouds, authors in [54] subvert the Dropbox client to hide files in the cloud with unlimited storage capacity. Although this work shares the same spirit than Chapter 5, we focus on the abuse of Personal Clouds from their REST API services instead of manipulating the desktop client. In fact, REST API services embody a more general form of abuse that can be exploited in more scenarios than desktop clients.

Furthermore, we prove our findings by designing and evaluating an application capable to abuse Personal Clouds via their REST APIs. Regarding abusive applications, few previous works have presented systems which benefit from the available Internet services. Close to our work, EMFS [96] is a personal storage system which aggregates cloud storage by establishing a RAID-like system on top of e-mail accounts. Other works propose backup tools or file systems benefiting from a variety of remote services, such as caches of Internet search engines, e-mail accounts and free web space [97, 98]. These works are clearly in line with our abusive application prototype, but they apply to different scenarios.

**Progress beyond state-of-the-art**: In contrast with previous research, Chapter 5 is the first work to study the potential of Personal Cloud REST APIs as a vector for exploitation. Furthermore, we developed and evaluated a file-sharing application to show how easy exploiting these services is. We believe that our insights in this field may be useful to public vendors in order to detect and mitigate the abusive use of their resources.

To summarize this section, Chapter 5 provides a thorough measurement and analysis of a novel aspect of a Personal Cloud: the REST API service. Furthermore, we investigate a new form of abuse that malicious users may perform against Personal Clouds by exploiting the REST API service over free accounts.

## 3.3 Analysis of QoS in Friend-to-Friend Storage Systems

### 3.3.1 Performance Analysis of F2F Storage Systems

In Part II of this dissertation, we focus on social storage systems. Concretely, one of our main interests is to understand the effect that aspects like the ON/OFF dynamics of participants or the structure of their social relationships have on the achievable storage QoS: data availability, load balancing and transfer performance.

Social storage systems —e.g., friend-to-friend (F2F), social clouds— originally emerged as an alternative to overcome many of the limitations of P2P storage systems, such as *free-riding* and the *lack of trust*. In fact, in the pioneering work of J. Li et al. [61], the authors argued that a user should choose its neighbors (the nodes with whom it shares data) based on existing *social links* instead of *randomly*. Such an approach provides incentives for users to cooperate and results in a more stable system which, in turn, reduces the cost of maintaining data.

However, despite the hype aroused by this new decentralized storage paradigm [61], social storage systems are still in their infancy. One of the contributions of this thesis is to provide

a deep understanding of the *specific characteristics* that govern the performance of social storage systems. In particular, we found two decisive factors to the operation of a social storage system: (i) Users may use *reduced friendsets* of strong ties to store data [65]; (ii) The probability for a user of finding logged off all the social links storing some specific content is high (i.e., *availability correlations*), particularly during night hours [99, 100].

Friendstore [20, 101] looked at how to ensure *availability* and *durability* in a social backup system by storing data only on trusted nodes, and hence discouraging free-riding. However, Friendstore evaluation was conducted by using availability traces of corporate desktop machines, which did not account for *availability correlations* existing in Internet systems. In Chapter 6 we specifically address this fundamental problem of social storage systems.

Intuitively, to build a decentralized social storage system, one may think on borrowing data management techniques from large-scale systems. Thus, a natural question that might arise could be: *Are the large-scale data management techniques suitable in a social storage scenario?*

Availability correlation is a well-known issue in large-scale distributed systems [102, 103], even for small groups [104]. For instance, [105] argues that the average user availability may be misleading when it is used to calculate data redundancy in the presence of availability correlations. Also, authors in [106] introduced new metrics to quantify the degree of online-offline correlation of nodes. They applied these metrics to improve the performance of task scheduling and file storage applications.

In Chapter 6, we present a novel history-based data availability estimation tailored to F2F storage systems. In this line, in the presence of peer heterogeneity and availability patterns, Kermarrec et al. proposed very recently [107] to resort to historical node availabilities to improve data placement and repairs. However, our work differs from [107] in several aspects: i) Contrary to a P2P scenario, the number of available nodes in a F2F system is extremely reduced; ii) In [107], the storage nodes are selected by their level of anti-correlation. This is not the case for a F2F system where nodes cannot be selected from a large set: they are restricted to a user's *trustworthy* social links. Overall, one of the main objectives of both approaches is to improve the data availability of the system.

Important research efforts have been focused on providing an adequate replica placement to guarantee data availability [108, 109]. Instead of replicas, authors in [110] proposed to store data blocks to nodes proportional to their availability. They also provided a numerical method to estimate data availability in a system with heterogeneous nodes.

The only work we are aware of that studied data availability in F2F storage systems is [99], which showed that F2F systems cannot guarantee high data availability: if no friends are online, then the data stored in the system will not be accessible by any means. However, this work did not consider the correlated dynamics of friends and how this affects to the storage service.

**Progress beyond state-of-the-art**: As a major contribution of this dissertation, in Chapter 6 we model and improve the poor data availability experienced in a F2F storage system. Furthermore, we also investigate data management techniques tailored to this particular scenario. This includes, for instance, an effective technique to calculate and generate the necessary amount of data redundancy in the presence of small friendsets with correlated availabilities. Clearly, our efforts go far beyond the current literature in social storage systems.

### 3.3.2 Hybrid or Cloud-assisted Architectures

Another major contribution visible in the second part of this thesis is the design (Chapter 6) and implementation (Chapter 7) of a cloud-assisted or hybrid social storage architecture. This architecture combines resources from both end-users and the cloud storage services to overcome the storage QoS limitations of purely decentralized social storage systems.

In this context, peer-assisted storage systems also combine the spare network bandwidth and storage space of peers with that of a cloud storage service such as Amazon S3. The key feature of peer-assisted storage is that it is comparable to the traditional client-server architecture but at a fraction of its costs [111]. A representative example was Wuala[1], a commercial storage service that now only stores files in data centers but that in the past it stored (encoded) fragments of the data on peers to save bandwidth at the server side [112]. Another example is AmazingStore [113], which augments a centralized cloud-based storage service with a P2P network to improve its resilience to correlated failures. Similarly, FS2You [114] is a large-scale online storage system with peer-based assistance and semipersistent file availability that was developed to reduce server bandwidth costs. Their measurement study demonstrated the feasibility of combining peers with cloud resources.

Another related type of systems are user-assisted systems; they combine *dedicated* user resources and cloud services to build a storage system. That is, Ctera [115] and Cleversafe [116] are online storage providers that sell network attached storage (NAS) devices that users or

[1]http://www.wuala.com/

small and medium enterprises (SMEs) install in their offices. The data stored in these NAS devices is replicated to datacenters and immediately accessible through an online service. Cleversafe even uses erasure codes as its redundancy scheme to optimize the utilization of the contributed storage resources, spreading stored data across several NAS devices, owned by different customers. However, as in the case of a traditional cloud service, users do not hold the *control of their data*; it is replicated and managed in the server-side. Moreover, users should acquire dedicated hardware to become part of the system, which differs from our targeted scenario.

**Progress beyond state-of-the-art**: In Chapter 6, we propose a cloud-assisted architecture tailored to the specific problems of F2F storage systems: availability correlations and small friendsets. Moreover, this architecture is complemented with a battery of specific data management techniques for social storage scenarios (data availability, scheduling).

In summary, Chapter 6 contributes to the state-of-the-art paying particular attention to the specific problems of F2F storage systems: small friendsets and availability correlations. In this setting, we study the suitability of traditional data management techniques (data availability, redundancy) and devise new ones. Finally, we design a novel cloud-assisted F2F storage architecture to overcome the QoS limitations of pure decentralization.

## 3.4 Empirical Analysis of Social Cloud Storage

### 3.4.1 Understanding Storage QoS in the Social Cloud

Many works in the literature discuss on the use of social networks for building computing systems and incentivizing resource sharing. One can find countless examples of applications that leverage existing social networks to manage and authenticate users and even recruit volunteers. For instance, both ASPEN [117] and PolarGrid [118] use social networks to manage users and facilitate resource sharing.

The social cloud model, first proposed in [22], takes a different tack by extending cloud-like functionality to online social networks instead of incorporating social networking to existing computation platforms. Since its born, a plethora of works have been examining the potential of this new social paradigm, particularly for underpinning computation [119, 120].

However, in the case of storage, only the research works [22, 23] partially explain some of the barriers to overcome towards the realization of socially oriented cloud storage. More specifically, these works concentrate on how to support storage trading through various social

market metaphors but do not give any discussion on the operational requirements of the social cloud storage like data availability and the amount of storage space to be contributed by friends. In Chapter 7, we aim at filling this gap by spotting concrete evidence of the existing operational hurdles in the social cloud storage.

In addition, there is a great deal of synergy between the social cloud and P2P networking paradigm in that services are provided by a network of peers. The P2P literature is full of examples of storage systems where the storage capacity is contributed by a pool of distributed peers such as Samsara [19] and PAST [121]. However, these systems lack of accountability, familiar interface, and the social incentives that minimize the administrative overhead, which are precisely the costs that social storage systems are meant to avoid.

Closer to the scope of a social cloud, F2F storage systems benefit from the trust relationships among users to provide a more reliable storage service [20, 61, 101]. The authors in [61] argued that a user should choose its neighbors (the nodes with which it shares data) based on existing *social relationships* instead of *randomly*. Similarly, Friendstore [20, 101] enables users to mutually back-up data via real-world negotiations. Although these systems could provide the storage functionality of a social cloud, to the best of our knowledge, none of them is fully integrated with a real-world social network. This is a primary requirement to realize a social cloud.

**Progress beyond state-of-the-art**: In Chapter 7, we implement `FriendBox`: the first social cloud storage application integrated with a real social network (Facebook). Furthermore, thanks to our insights in Chapter 6, our application leverages resources from friends and cloud resources to let users infer the right balance between data control and QoS.

### 3.4.2 Impact of the Social Network on the Performance of a Social Cloud

In a social cloud, storage interactions among users are defined by the social graph. Thus, an interesting aspect that has not received enough attention is the role that the social network topology plays on the storage QoS of a social cloud. This is one of the key points in Chapter 7.

Related to this topic, a myriad of research efforts have been devoted to build Decentralized Online Social Networks (DOSNs) [75, 76, 77, 78], i.e., social networks that can operate in a decentralized fashion thanks to the resources contributed by users. DOSNs are assumed to provide higher security and privacy guarantees than nowadays's massive online social networks, such as Facebook and Google+.

Buchegger et al. present in [77] a prototype implementation of a DOSN system. Moreover, authors discuss about the relevance of geographical/temporal storage diversity to data availability and the asymmetry of user contributed versus consumed resources. These research topics are very aligned to our empirical study presented in Chapter 7.

Regarding data placement, Sharma et al. [78] investigated the effect of various placement strategies to the data availability in an DOSN. The data placement strategies included in this work considered storing data only at friends, in super-peers as well as mixing them with strangers (i.e., users that are not friends). However, the number of candidate storage nodes was generally higher than the expected amount of *strong ties* (e.g., family, close friends) in the social circles of users to sustain a permanent personal storage service. Furthermore, authors did not explored in depth the role of the social topology structure (node degree, clustering) on the storage QoS.

The closest work that we are aware of in this respect is [122]. Zuo et al. present a metric to quantify the strength of indirect ties (i.e., *friends of friends*). This metric is then used to extend user friendsets in a F2F system while preserving social incentives. Among the presented use cases, authors consider a F2F storage system scenario. In fact, this work can be seen as a potential solution to the problems that we empirically analyze and characterize in Chapter 7.

**Progress beyond state-of-the-art**: Compared with previous works, Chapter 7 goes a step further by providing novel insights on the interplay between the social network topology and the storage QoS of a social cloud. We believe that understanding this interplay is vital to appraise to what extent the social cloud can emerge as a true alternative to existing commercial and non-profit storage systems.

To conclude, we present `FriendBox` in Chapter 7: a social cloud system for storage that efficiently combines resources from trusted friends and cloud services to provide a flexible, trusted and private personal storage service. Moreover, we conduct an experimental study with `FriendBox` to understand the implications of the social topology (e,g, degree, clustering) on the achievable storage QoS.

# Part I

# Measurement and Analysis of Personal Clouds

# 4

# Dissecting a Personal Cloud Back-end

**Summary**

In this Chapter, we focus on understanding the nature of Personal Clouds by presenting the *internal structure* and *measurement study* of UbuntuOne (U1). We first detail the U1 architecture, core components involved in the U1 metadata service hosted in the datacenter of Canonical, as well as the interactions of U1 with Amazon S3 to outsource data storage. Moreover, by means of tracing the U1 servers for one month, we provide an extensive analysis of the storage workload, user behavior and the performance of the metadata back-end. Finally, we discuss potential improvements to the operation of U1 that may be of interest to similar systems.

*A paper with the results of this Chapter has been submitted for publication.*

## 4.1 Introduction

Today, users require ubiquitous and transparent storage to help handle, synchronize and manage their personal data. In a recent report, Forrester research [123] forecasts a market of 12 billion in the US in paid subscriptions to personal and user-centric cloud services by 2016. In response to this demand, Personal Clouds like Dropbox, Box and UbuntuOne (U1) have proliferated and become increasingly popular, attracting companies such as Google, Microsoft, Amazon or Apple to offer their own integrated solutions in this field.

In a nutshell, a Personal Cloud service offers automatic backup, file sync, sharing and remote accessibility across a multitude of devices and operating systems. The popularity of these services is based on their easy to use Software-as-a-Service (SaaS) storage facade to ubiquitous Infrastructure-as-a-Service (IaaS) providers like Amazon S3 and others.

Unfortunately, due to the proprietary nature of these systems, very little is known about their performance and characteristics, including the workload they have to handle daily. And indeed, the few available studies have to rely on the so-called "black-box" approach, where traces are collected from a single or a limited number of measurement points, in order to infer their properties. This was the approach followed by the most complete analysis of a Personal Cloud to date, the measurement of Dropbox conducted by Drago et al. [11]. Although this work describes the overall service architecture, it provides no insights on the operation and infrastructure of the Dropbox's back-end. And also, it has the additional flaw that it only focuses on small and specific communities, like university campuses, which may breed false generalizations.

Similarly, several Personal Cloud services have been externally probed to infer their operational aspects, such as data reduction and management techniques [24, 37, 38], or even transfer performance [29, 90]. However, from external vantage points, it is impossible to fully understand the operation of these systems without fully reverse-engineering them.

In this Chapter, we present results of our study of U1: the Personal Cloud of Canonical, integrated by default in Linux Ubuntu OS. Despite the shutdown of this service on July 2014, the distinguishing feature of our analysis is that it has been conducted using data collected by the provider itself. U1 provided service to several millions of users at the time of the study on January-February 2014, which constitutes the first complete analysis of the performance of a Personal Cloud in the wild. Such a unique data set has allowed us to reconfirm results from prior studies, like that of Drago et al. [11], which paves the way for a general characterization of these systems. But it has also permitted us to expand the knowledge base on these services,

| *UbuntuOne Analisys* | **Finding** | **Implications and Opportunities** |
|---|---|---|
| Storage Workload (4.4) | 90% of files are smaller than 1MByte. | Object storage services normally used as a cloud service are not optimized for managing small files [124]. |
| | 18.5% of the upload traffic is caused by file updates. | Changes in file metadata cause high overhead since the U1 client does not support delta updates (e.g. `.mp3` tags). |
| | We detected a deduplication ratio of 17% in one month. | File-based cross-user deduplication provides an attractive trade-off between complexity and performance [24]. |
| | DDoS attacks against U1 are frequent. | Further research is needed regarding secure protocols and automatic countermeasures for Personal Clouds. |
| User Behavior (4.5) | 1% of users that manage files generate 65% of the traffic. | Very active users may be treated in an optimized manner to reduce storage costs. |
| | Data management operations, such as uploads or file deletions, are normally executed in long sequences. | This correlated behavior can be exploited by caching and prefetching mechanisms in the server-side. |
| | User operations are *bursty*; users transition between long, idle periods and short, very active ones. | User behavior combined with the user per-shard data model impacts the metadata back-end load balancing. |
| Metadata Back-end Performance (4.6) | A 20-node database cluster provided service to 1.17M users without symptoms of congestion. | The *user-centric data model* of a Personal Cloud makes relational database clusters a simple yet effective approach to scale out metadata storage. |
| | RPCs service time distributions accessing the metadata store exhibit long tails. | Several factors at hardware, OS and application-level are responsible for poor tail latency in RPC servers [125]. |
| | In short time windows, load values of API servers/DB shards are very far from the mean value. | Further research is needed to achieve better load balancing under this type of workload. |

**Table 4.1:** Summary of some of our most important findings and their implications.

which now represent a considerable volume of the Internet traffic. According to Drago et al. [11], the total volume of Dropbox traffic accounted for a volume equivalent to around one third of the YouTube traffic on a campus network. Consequently, we believe that the results of our study can be useful for both researchers, ISPs and data center designers, giving hints on how to anticipate the impact of the growing adoption of these services. In summary, our contributions are the following:

**Back-end architecture and operation of U1**. This Chapter provides a comprehensive description of the U1 architecture, being the first study to depict the back-end infrastructure of a real-world vendor. Similarly to Dropbox [11], U1 decouples the storage of *file contents* (data) and *their logical representation* (metadata). Canonical only owns the infrastructure for the metadata service, whereas the actual file contents are stored separately in Amazon S3. Among other insights, we found that U1 API servers are characterized by long tail latencies and that a sharded database cluster is an effective way of storing metadata in these systems. Interestingly, these issues may arise in other systems that decouple data and metadata as U1 does [34].

**Workload analysis and user behavior in U1**. By tracing the U1 servers in the Canonical datacenter, we provide an extensive analysis of its *back-end activity* produced by the *entire user*

*population* of U1 for one month (1.17M distinct users). Our analysis confirms already reported facts, like the execution of user operations in long sequences [11] and the potential waste that file updates may induce in the system [24, 38]. Moreover, we provide new observations, such as a taxonomy of files in the system, the modeling of burstiness in user operations or the detection of attacks to U1, among others. Table 4.1 summarizes some of our key findings.

**Potential improvements to Personal Clouds**. We suggest that a Personal Cloud should be aware of the *behavior of users* to optimize its operation. Given that, we discuss the implications of our findings to the operation of U1. For instance, despite U1 was frequently used for editing files, file updates were responsible for 18.5% of upload traffic mainly due to the lack of delta updates in the desktop client. Furthermore, we detected 3 DDoS attacks in one month, motivating the need for further research in automatic attack countermeasures in secure and dependable storage protocols. Although our observations may not apply to *all* existing services, we believe that our analysis can help to improve the next generation of Personal Clouds [34, 38].

**Publicly available dataset**. We contribute our dataset (773GB) to the community and it is available at `http://cloudspaces.eu/datasets/u1_measurement`. To our knowledge, this is the first dataset that contains the back-end activity of a large-scale Personal Cloud. We hope that our dataset provides new opportunities to researchers in further understanding the internal operation of Personal Clouds, promoting research and experimentation in this field.

The rest of this Chapter is organized as follows. We describe in Section 4.2 the architecture of U1 and its metadata back-end. In Section 4.3 we explain the trace collection methodology. In Section 4.4, 4.5 and 4.6 we analyze the storage workload, user activity and back-end performance of U1, respectively. We discuss the implications of our insights and draw conclusions in Section 4.7.

## 4.2 The U1 Personal Cloud

U1 was a suite of online services from Canonical that enabled users to store and sync files online and between computers, as well as sharing files/folders with others using file synchronization. Until the service shutdown in July 2014, U1 provided desktop and mobile clients and a Web front-end. U1 was integrated with other Ubuntu services, like Tomboy for notes and U1 Music Store for music streaming.

In this section, we first describe the U1 storage protocol used for communication between clients and the server-side infrastructure (Sec. 4.2.1). This will facilitate the understanding of the system architecture (Sec. 4.2.2). We then discuss the details of a U1 desktop client (Sec. 4.2.3). Finally, we give details behind the core component of U1, its metadata back-end (Sec. 4.2.4).

### 4.2.1 U1 Storage Protocol

U1 uses its own protocol (`ubuntuone-storageprotocol`) based on TCP and Google Protocol Buffers[1]. In contrast to most commercial solutions, the protocol specifications and client-side implementation are publicly available[2]. Here, we describe the protocol in the context of its *entities* and *operations*. Operations can be seen as end-user actions intended to manage one/many entities, such as a file or a directory.

#### Protocol Entities

In the following, we define the main entities in the protocol. Note that in our analysis, we characterize and identify the role of these entities in the operation of U1.

**Node**: Files and directories are *nodes* in U1. For files, U1 decouples their logical representation from their actual contents. Drawing a comparison to a file system, the inodes are stored in the metadata service and the extents are stored in Amazon S3. The protocol supports CRUD operations on nodes (e.g. list, delete, etc.). The protocol assigns Universal Unique Identifiers (UUIDs) to both node objects and their contents, which are generated in the back-end.

**Volume**: A volume is a container of node objects. During the installation of the U1 client, the client creates an initial volume to store files with id=0 (root). There are 3 types of volumes: i) *root/predefined*, ii) *user defined folder* (UDF), which is a volume created by the user, and iii) *shared* (sub-volume of another user to which the current user has access).

**Session**: A user interacts with the server in the context of a U1 storage protocol session (not HTTP or any other session type). This session is used to identify the requests of a single user during the session lifetime. Usually, sessions do not expire automatically. A client may disconnect, or a server process may go down, and that will end the session. For this reason, in parallel with a session, a user establishes a TCP connection with U1 that is used to detect these

[1] `https://wiki.ubuntu.com/UbuntuOne`
[2] `https://launchpad.net/ubuntuone-storage-protocol`

| *API Operation* | *Related RPC* | *Description* |
|---|---|---|
| `ListVolumes` | `dal.list_volumes` | This operation is normally performed at the beginning of a session and lists all the volumes of a user (root, udf, shared). |
| `ListShares` | `dal.list_shares` | This operation lists all the volumes of a user that are of *type share*. In this operation, ther field `shared by` is the owner of the volume and `shared to` is the user to which that volume was shared with. In this operation, the field `shares` represents the number of volumes *type share* of this user. |
| `(Put/Get)Content` | see Appendix A | These operations are the actual file uploads and downloads, respectively. The notification goes to the U1 back-end but the actual data is stored in a separate service (Amazon S3). A special process is created to forward the data to Amazon S3. Since the upload management in U1 is complex, we refer the reader to Appendix A for a description in depth of upload transfers. |
| `Make` | `dal.make_dir` `dal.make_file` | This operation is equivalent to a "touch" operation in the U1 back-end. Basically, it creates a file node entry in the metadata store and normally precedes a file upload. |
| `Unlink` | `dal.unlink_node` | Delete a file or a directory from a volume. |
| `Move` | `dal.move` | Moves a file from one directory to another. |
| `CreateUDF` | `dal.create_udf` | Creates a volume of type udf. |
| `DeleteVolume` | `dal.delete_volume` | Deletes a volume and the contained nodes. |
| `GetDelta` | `dal.get_delta` | Get the differences between the server volume and the local one (generations). |
| `Authenticate` | `auth.get_user_id_from_token` | Operations managed by the servers to create sessions for users. |

**Table 4.2:** Description of the most relevant U1 API operations.

events. To create a new session, an OAuth [51] token is used to authenticate clients against U1. Tokens are stored separately in the Canonical authentication service (see 4.2.4).

### API Operations

The U1 storage protocol offers an API consisting of the *data management* and *metadata operations* that can be executed by a client. Metadata operations are those operations that do not involve transfers to/from the data store (i.e., Amazon S3), such as listing or deleting files, and are entirely managed by the synchronization service. On the contrary, uploads and downloads are, for instance, typical examples of data management operations.

In Table 4.2 we describe the most important protocol operations between users and the server-side infrastructure. We traced these operations to quantify the system's workload and the behavior of users.

## 4.2.2 Architecture Overview

As mentioned before, U1 has a 3-tier architecture consisting of *clients*, *synchronization service* and the *data/metadata store*. Similarly to Dropbox [11], U1 *decouples* the storage of *file contents* (data) and *their logical representation* (metadata). Canonical only owns the infrastructure for the metadata service, which processes requests that affect the virtual organization of files in user volumes. The actual contents of file transfers are stored separately in Amazon S3.

However, U1 treats *client requests* differently from Dropbox. Namely, Dropbox enables clients to send requests either to the metadata or storage service depending on the request type. Therefore, the Dropbox infrastructure only processes metadata/control operations. The cloud storage service manages data transfers, which are normally orchestrated by computing instances (e.g. EC2).

In contrast, U1 receives both metadata requests and data transfers of clients. Internally, the U1 service discriminates client requests and redirects them either to the metadata store or the storage service, respectively. For each upload and download request, a new process is instantiated to manage the transfer between the client and S3 (see A). Therefore, the U1 model is simpler from a design perspective, yet this comes at the cost of delegating the responsibility of processing data transfers to the metadata back-end.

**U1 Operation Workflow.** Imagine a user that initiates the U1 desktop client (4.2.3). At this point, the client sends an `Authenticate` API call (see Table 4.2) to U1, in order to establish a new session. An API server receives the request and contacts to the Canonical authentication service to verify the validity of that client. Once the client has been authenticated, a persistent TCP connection is established between the client and U1. Then, the client may send other management requests on user files and directories.

To understand the synchronization workflow, let us assume that two clients are online and work on a shared folder. Then, a client sends an `Unlink` API call to delete a file from the shared folder. Again, an API server receives this request, which is forwarded in form of Remote Procedure Call (RPC) to a RPC server (4.2.4). As we will see, RPC servers translate RPC calls into database query statements to access the correct metadata store shard (PostgreSQL cluster). Thus, the RPC server deletes the entry for that file from the metadata store.

When the query finishes, the result is sent back from the RPC server to the API server that responds to the client that performed the request. Moreover, the API server that handled the `Unlink` notifies the other API servers about this event that, in turn, is detected by the API server to which the second user is connected. This API server notifies via push to the second client, which deletes that file locally.

Next, we describe in depth the different elements involved in this example of operation: The desktop client, the U1 back-end infrastructure and other key back-end services to the operation of U1 (authentication and notifications).

### 4.2.3 U1 Desktop Client

U1 provides a user friendly desktop client, implemented in Python (GPLv3), with a graphical interface that enables users to manage files. It runs a daemon in the background that exposes a message bus (DBus) interface to handle events in U1 folders and make server notifications visible to the user through OS desktop. This daemon also does the work of deciding what to synchronize and in which direction to do so.

By default, one folder labeled ∼/`Ubuntu One`/ is automatically created and configured for mirroring (root `volume`) during the client installation. Changes to this folder (and any others added) are watched using `inotify`. Synchronization metadata about directories being mirrored is stored in ∼/`.cache/ubuntuone`. When remote content changes, the client acts on the incoming unsolicited notification (push) sent by U1 service and starts the download. Push notifications are possible since clients establish a TCP connection with the metadata service that remains open while online.

In terms of data management, Dropbox desktop clients deduplicate data at chunk level [11]. In contrast, U1 resorts to file-based cross-user deduplication to reduce the waste of storing repeated files [24]. Thus, to detect duplicated files, U1 desktop clients provide to the server the SHA-1 hash of a file prior to the content upload. Subsequently, the system checks if the file to be uploaded already exists or not. In the affirmative case, the new file is *logically linked* to the existing content, and the client does not need to transfer data.

Finally, as observed in [24], the U1 client applies compression to uploaded files to optimize transfers. However, it does not perform advanced techniques, such as file bundling[1], delta updates and sync deferment, to buffer frequent changes to the same file, leading to potential inefficiencies.

### 4.2.4 U1 Metadata Back-end

The entire U1 back-end is all inside a single datacenter and its objective is to manage the metadata service. The back-end architecture appears in Fig. 4.1 and consists of *metadata servers* (API/RPC), *metadata store* and *data store*.

**System gateway**. The gateway to the back-end servers is the load balancer. The load balancer (HAProxy, ssl, etc.) is the visible endpoint for users and it is composed of two racked servers.

[1]Li et al. [24] suggest that U1 may group small files together for upload (i.e. bundling), since they observed high efficiency uploading sets of small files. However, U1 does not bundle small files together. Instead, clients establish a TCP connection with the server that remains open during the session, avoiding the overhead of creating new connections.

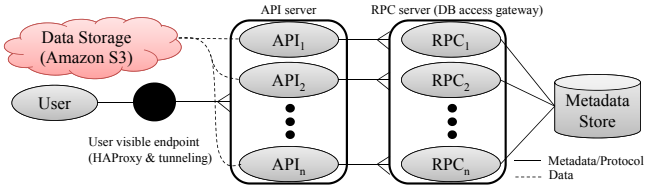



**Figure 4.1:** Architecture of U1 back-end.

**Metadata store**. U1 stores metadata in a PostgreSQL database cluster composed of 20 large Dell racked servers, configured in 10 shards (master-slave). Internally, the system routes operations *by user identifier* to the appropriate shard. Thus, metadata of a user's files and folders reside always in the same shard. This data model *effectively* exploits sharding, since normally there is no need to lock more than one shard per operation (i.e. lockless). Only operations related to shared files/folders may require to involve more than one shard in the cluster.

**API/RPC servers**. Beyond the load balancer we find the API and RPC database processes that run on 6 separate racked servers. API servers receive commands from the user, perform authentication, and translate the commands into RPC calls. In turn, RPC database workers translate these RPC calls into database queries and route queries to the appropriate database shards. API/RPC processes are more numerous than physical machines (normally $8-16$ processes per physical machine), so that they can migrate among machines for load balancing. Internally, API and RPC servers, the load balancer and the metadata store are connected though a switched 1Gbit Ethernet network.

**Data storage**. Like other popular Personal Clouds, such as Dropbox or SugarSync, U1 stores user files in a separate cloud service. Concretely, U1 resorts to Amazon S3 (`us-east`) to store user data. This solution enables a service to rapidly scale out without a heavy investment in storage hardware. In its latests months of operation, U1 had a $\approx 20,000$\$ monthly bill in storage resources, being the most important Amazon S3 client in Europe.

With this infrastructure, U1 scaled up to 4 million registered users (1.17M were traced in this measurement).

## Authentication Service

The authentication service of U1 is shared with other Canonical services within the same datacenter and it is based on OAuth [51]. The first time a user interacts with U1, the desktop client requires him to introduce his credentials (email, password). The API server that handles

the authentication request contacts the authentication service to generate a new token for this client. The created token is associated in the authentication service with a new user identifier. The desktop client also stores this token locally in order to avoid exposing user credentials in the future.

In the subsequent connections of that user, the authentication procedure is easier. Basically, the desktop client sends a connection request with the token to be authenticated. The U1 API server responsible for that requests asks the authentication service if the token does exist and has not expired. In the affirmative case, the authentication service retrieves the associated user identifier, and a new session is established. During the session, the token of that client is cached to avoid overloading the authentication service.

The authentication infrastructure consists of 1 database server with hot failover and 2 application servers configured with crossed stacks of Apache/Squid/HAProxy.

### Notifications

Clients detect changes in their volumes by comparing their local state with the server side on every connection (generation point). However, if two related clients are online and their changes affect each other (e.g. updates to shares, new shares), API servers notify them directly (push). To this end, API servers resort to the TCP connection that clients establish with U1 in every session.

Internally, the system needs a way of notifying changes to API servers that are relevant to simultaneously connected clients. Concretely, U1 resorts to RabbitMQ (1 server) for communicating events between API servers[1], which are subscribed in the queue system to send and receive new events to be communicated to clients.

Next, we describe our measurement methodology to create the dataset used in our analysis.

## 4.3 Data Collection

We present a study of the U1 service back-end. In contrast to other Personal Cloud measurements [11, 24, 29], we did not deploy vantage points to analyze the service externally. Instead, we inspected directly the U1 metadata servers to measure the system. This has been done in collaboration with Canonical in the context of the FP7 CloudSpaces[2] project. Canonical anonimyzed sensitive information to build the trace, following strict ethical guidelines.

[1] If connected clients are handled by the same API process, their notifications are sent immediately, i.e. there is no need for inter-process communication with RabbitMQ.

[2] `http://cloudspaces.eu`

| | |
|---|---|
| Trace duration | 30 days (01/11 - 02/10) |
| Trace size | 773 GB (3,391M lines) |
| Back-end servers traced | 6 servers (all) |
| Unique users | 1,170,880 |
| Unique files | 137.63M |
| User sessions | 42.5M |
| Transfer operations | 194.3M |
| Total upload traffic | 105TB |
| Total download traffic | 120TB |

**Table 4.3:** Summary of the trace.

The traces are taken at both *API* and *RPC* server stages. In the former stage we collected important information about the storage workload and user behavior, whereas the second stage provided us with valuable information about the requests' life-cycle and the metadata store performance.

We built the trace capturing a series of service *logfiles*. Each logfile corresponds to the entire activity of a single API/RPC process in a machine for a period of time. Each logfile is within itself strictly *sequential and timestamped*. Thus, causal ordering is ensured for operations done for the same user. However, the timestamp between servers is not dependable, even though machines are synchronized with NTP (clock drift may be in the order of ms).

To gain better understanding on this, consider a line in the trace with this logname: `production-whitecurrant-23-20140128`. They will all be `production`, because we only looked at production servers. After that prefix is the name of the physical machine, followed by the number of the server process. The mapping between services and servers is dynamic within the time frame of analyzed logs, since they can migrate between servers to balance load. In any case, the identifier of the process is unique within a machine. After that is the date the logfile was "cut" (there is one log file per server/service and day).

Database sharding is in the metadata store back-end, so it is behind the point where traces were taken. This means that in these traces any combination of server/process can handle any user. To have a strictly sequential notion of the activity of a user we should take into account the U1 *session* and sort the trace by timestamp (a user may have more than one parallel connection). A session starts in the least loaded machine and lives in the same node until it finishes, making user events strictly sequential. Thanks to this information we can estimate system and user service times.

Approximately 1% of traces are not analyzed due to failures parsing of the logs.

## Dataset

The trace is the result of merging all the logfiles (773GB of `.csv` text) of the U1 servers for 30 days (see Table 4.3).

The trace contains the API operations (request type `storage/storage_done`) and their translation into RPC calls (request type `rpc`), as well as the session management of users (request type `session`). This provides different sources of valuable information. For instance, we can analyze the *storage workload* supported by a real-world cloud service (users, files, operations). Since we captured file properties such as file size and hash, we can study the storage system in high detail (contents are not disclosed).

**Dataset limitations**. We mentioned that timestamps among servers are not dependable since they may be different (in order of ms). Also, the dataset only includes events originating from desktop clients. Other sources, namely the web front-end and the mobile clients, are not included. This is because the different client types are handled by different software stacks that were not logged. Finally, we detected that sharing among users is limited.

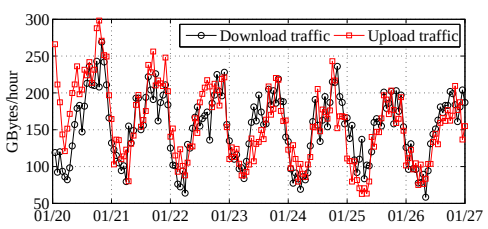
## 4.4 Storage Workload

### 4.4.1 Macroscopic Daily Usage

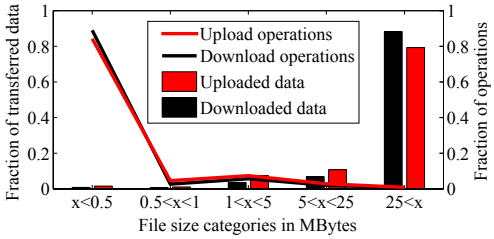
First, we quantify the storage workload supported by U1 for one month. Moreover, we pay special attention to the behavior of files in the system, to infer potential improvements. We also unveil attacks perpetrated to the U1 service.

**Storage traffic and operations**. Fig. 4.2a provides a time-series view of the upload/download traffic of U1. We observe in Fig. 4.2a that U1 exhibits important *daily patterns*. To wit, the volume of uploaded GBytes per hour can be up to 10x higher in the central day hours compared to the nights. This observation is aligned with previous works, that detected time-based variability in both the usage and performance of Personal Cloud services [11, 29]. This effect is probably related to the working habits of users, since U1 desktop clients are by default initiated automatically when users turn on their machines.

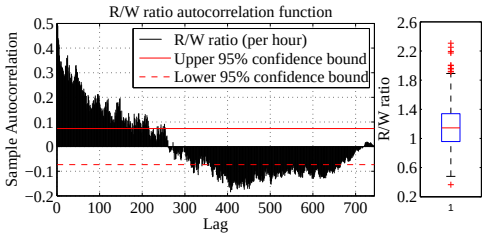
Another aspect to explore is the relationship between file size and its impact in terms of upload/download traffic. To do so, in Fig. 4.2b, we depict in relative terms the fraction of transferred data and storage operations for distinct file sizes. As can be observed, a very small amount of large files ($>$ 25MBytes) capitalizes 79.3% and 88.2% of upload and download traffic, respectively. Conversely, 84.3% and 89.0% of upload and download operations are related to small files ($<$ 0.5MBytes). As reported in other domains [126, 127, 128], we conclude




**(a)** Transferred traffic time-series.




**(b)** Traffic vs file size category.




**(c)** R/W ratio analysis.

**Figure 4.2:** Macroscopic storage workload metrics of U1.

that in U1 the workload in terms of *storage operations* is dominated by small files, whereas a small number of large files generate most of the *network traffic*.

For uploads, we found that 10.05% of total upload operations are *updates*, that is, an upload of an existing file that has distinct hash/size. However, in terms of traffic, file updates represent 18.47% of the U1 upload traffic. This can be partly explained by the lack of delta updates in the U1 client and the heavy file-editing usage that many users exhibited (e.g., code developers). Particularly for media files, U1 engineers found that applications that modify the metadata of files (e.g., tagging `.mp3` songs) induced high upload traffic since the U1 client uploads again files upon metadata changes, as they are interpreted as regular updates.

To summarize, Personal Clouds tend to exhibit daily traffic patterns, and most of this traffic is caused by a small number of large files. Moreover, desktop clients should efficiently handle file updates to minimize traffic overhead.

**R/W ratio**. The read/write (R/W) ratio represents the relationship between the downloaded and uploaded data in the system for a certain period of time. Here we examine the variability of the R/W ratio in U1 (1-hour bins). The boxplot in Fig. 4.2c shows that the R/W ratio *variability can be important*, exhibiting differences of 8x within the same day. Moreover, the median (1.14) and mean (1.17) values of the R/W ratio distribution point out that the U1 workload is *slightly read-dominated*, but not as much as it has been observed in Dropbox [11].

This indicates that users mainly use U1 as a storage service, rather than for sharing content.

We also want to explore if the R/W ratios present patterns or dependencies along time due to the working habits of users. To verify whether R/W ratios are independent along time, we calculated the autocorrelation function (ACF) for each 1-hour sample (see Fig. 4.2c). To interpret Fig. 4.2c, if R/W ratios are completely uncorrelated, the sample ACF is approximately normally distributed with mean 0 and variance $1/N$, where $N$ is the number of samples. The 95% confidence limits for ACF can then be approximated to $\pm 2/\sqrt{N}$.

As shown in Fig. 4.2c, R/W ratios are not independent, since most lags are outside 95% confidence intervals, which indicates long-term correlation with alternating positive and negative ACF trends. This evidences that the R/W ratios of U1 workload are not random and follow a pattern also guided by the working habits of users.

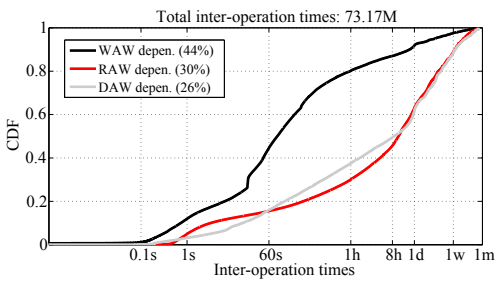
Concretely, averaging R/W ratios for the same hour along the whole trace, we found that from 6am to 3pm the R/W ratio shows a linear decay. This means that users download more content when they start the U1 client, whereas uploads are more frequent during the common working hours. For evenings and nights we found no clear R/W ratio trends.

We conclude that different Personal Clouds may exhibit disparate R/W ratios, mainly depending on the purpose and strengths of the service (e.g., sharing, content distribution). Moreover, R/W ratios exhibit patterns along time, which can be predicted in the server-side to optimize the service.

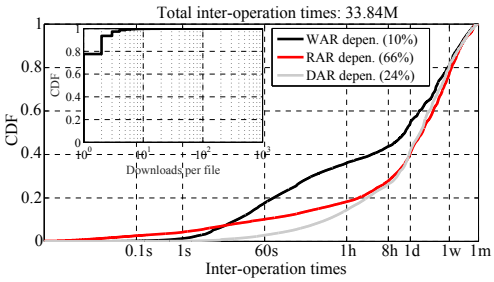
### 4.4.2 Analysis of Files in U1

**File operation dependencies**. Essentially, in U1 a file can be *downloaded (or read)* and *uploaded (or written)* multiple times, until it is eventually *deleted*. Next, we aim at inspecting the dependencies among file operations [86, 129], which can be *RAW* (Read-after-Write), *WAW* (Write-after-Write) or *DAW* (Delete-after-Write). Analogously, we have *WAR*, *RAR* and *DAR* for operations executed after a read.

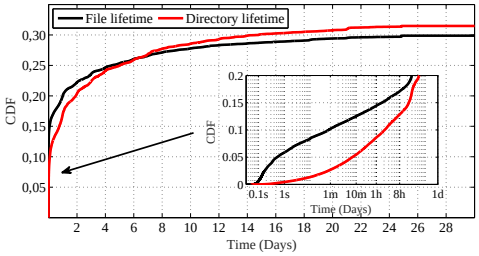
First, we inspect file operations that occur after a write (Fig. 4.3a). We see that *WAW* dependencies are the most common ones (30.1% of 170.01M in total). This can be due to the fact that users *regularly update synchronized files*, such as documents of code files. This result is consistent with the results in [129] for personal workstations where block updates are common, but differs from other organizational storage systems in which files are almost immutable [86]. Furthermore, the 80% of WAW times are shorter than 1 hour, which seems reasonable since users may update a single text-like file various times within a short time lapse.




**(a)** X-after-Write inter-arrival times.




**(b)** X-after-Read inter-arrival times.




**(c)** File/directory lifetime.

**Figure 4.3:** Usage and behavior of files in U1.

In this sense, Fig. 4.3a shows that *RAW* dependencies are also relevant. Two events can lead to this situation: (i) the system synchronizes a file to another device right after its creation, and (ii) downloads that occur after every file update. For the latter case, reads after successive writes can be optimized with sync deferment to reduce network overhead caused by synchronizing intermediate versions to multiple devices [24]. This has not been implemented in U1.

Second, we inspect the behavior of X-after-Read dependencies (Fig. 4.3b). As a consequence of active update patterns (i.e., write-to-write) and the absence of sync deferment, we see in Fig. 4.3b that *WAR* transitions also occur within reduced time frames compared to other transitions. Anyway, this dependency is the least popular one yielding that files that are *read tend not to be updated again*.

In Fig. 4.3b, 40% of RAR times fall within 1 day. RAR times are shorter than the ones reported in [86], which can motivate the introduction of *caching mechanisms* in the U1 backend. Caching seems specially interesting observing the inner plot of Fig. 4.3b that reveals a long tail in the distributions of reads per file. This means that a small fraction of files is very popular and may be effectively cached.

By inspecting the Delete-after-X dependencies, we detected that around 12.5M files in U1

were *completely unused* for more than 1 day before their deletion (9.1% of all files). This simple observation on dying files evidences that *warm and/or cold data exists* in a Personal Cloud, which may motivate the involvement of warm/cold data systems in these services (e.g., Amazon Glacier, f4 [130]). To efficiently managing warm files in these services is object of current work.

**Node lifetime**. Now we focus on the lifetime of user files and directories (i.e., nodes). As shown in Fig. 4.3c, 28.9% of the new files and and 31.5% of the recently created directories are deleted within one month. We also note that the lifetime distributions of *files and directories are very similar*, which can be explained by the fact that deleting a directory in U1 triggers the deletion of all the files it contains.

This figure also unveils that a large fraction of nodes are deleted within *few hours after their creation*, especially for files. Concretely, users delete 17.1% of files and 12.9% of directories within 8 hours after their creation time.

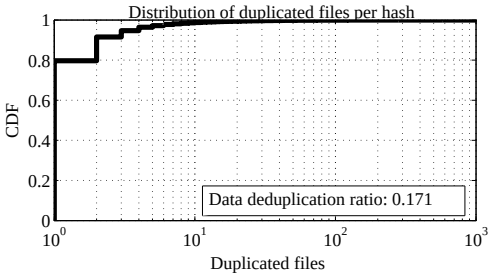
All in all, in U1 files exhibit similar lifetimes than files in local file systems. For instance, Agrawal et al. in [127] analyzed the lifetimes of files in corporative desktop computers for five years. They reported that around 20% to 30% of files (depending on the year) in desktop computers present a lifetime of one month, which agrees with our observations. This suggests that *users behave similarly deleting files* either in synchronized or local folders.

### 4.4.3 File Deduplication, Sizes and Types

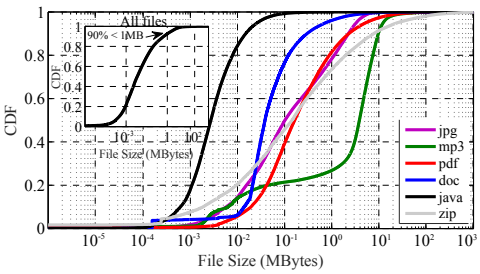
**File-based deduplication**. The deduplication ratio ($dr$) is a metric to quantify the proportion of duplicated data. It takes real values in the interval $[0, 1)$, with 0 signaling no file deduplication at all, and 1 meaning full deduplication. It is expressed as $dr = 1 - (D_{unique}/D_{total})$, where $D_{unique}$ is the amount of unique data, and $D_{total}$ is equal to the total storage consumption.

We detected a $dr$ of 0.171, meaning that the 17% of files in the trace can be deduplicated. This is slightly better than the deduplication ratio reported by Canonical ($\approx$ 11%), and similar (18%) to that given by the recent work of Li et al. [24]. This suggests that file-based cross-user deduplication could be a practical approach to reduce storage costs in U1.

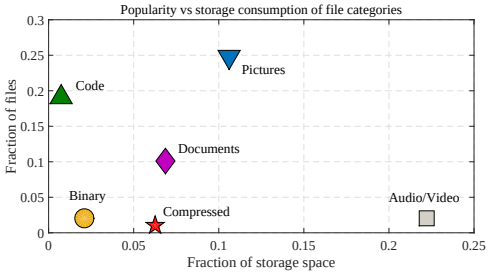
Moreover, Fig. 4.4a demonstrates that the distribution of file objects w.r.t unique contents exhibits a long tail. This means that a small number of files accounts for a very large number of duplicates (e.g., popular songs), whereas 80% files present no duplicates. Hence, files with many duplicates represent a *hot spot* for the deduplication system, since a large number of logical links point to a single content.




**(a)** Duplicated files per hash.




**(b)** Size of files per extension.




**(c)** Number/storage share of file types.

**Figure 4.4:** Characterization of files in U1.

**File size distribution**. The inner plot of Fig. 4.4b illustrates the file size distribution of transferred files in the system. At first glance, we realize that the *vast majority of files are small* [126, 127, 128]. To wit, 90% of files are smaller than 1MByte. In our view, this can have important implications on the performance of the back-end storage system. The reason is that Personal Clouds like U1 use object storage services offered by cloud providers as data store, which has not been designed for storing very small files [124].

In this sense, Fig. 4.4b shows the file size distribution of the most popular file extensions in U1. Non-surprisingly, the distributions are very disparate, which can be used to model realistic workloads in Personal Cloud benchmarks [37]. It is worth to note that in general, incompressible files like zipped files or compressed media are larger than compressible files (docs, code). This observation indicates that compressing files *does not provide much benefits* in many cases.

**File types: number vs storage space**. We classified files belonging to the 55 most popular file extensions into 7 categories: Pics (`.jpg`, `.png`, `.gif`, etc.), Code (`.php`, `.c`, `.js`, etc.), Docs (`.pdf`, `.txt`, `.doc`, etc.), Audio/Video (`.mp3`, `.wav`, `.ogg`, etc.), Application/Binary (`.o`, `.msf`, `.jar`, etc.) and Compressed (`.gz`, `.zip`, etc.). Then, for each category, we calculated the ratio of the number of files to the total in the system. We did the same for the storage space. This

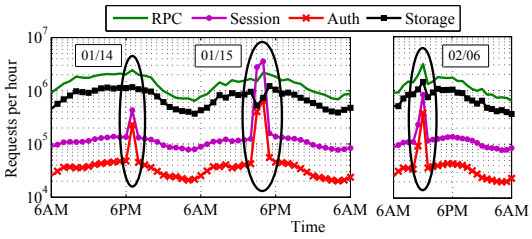



**Figure 4.5:** DDoS attacks detected in our trace.

captures the relative importance of each content type.

Fig. 4.4c reveals that Audio/Video category is one of the most relevant types of files regarding the share of consumed storage, despite the fraction of files belonging to this class is low. The reason is that U1 users stored `.mp3` files, which are usually larger than other popular text-based file types.

Further, the Code category contains the highest fraction of files, indicating that many U1 users are code developers who frequently update such files, despite the storage space required for this category is minimal. Docs are also popular (10.1%), subject to updates and hold 6.9% of the storage share. Since the U1 desktop client lacks delta updates and deferred sync, such frequent updates suppose a high stress for desktop clients and induce significant network overhead [24].

### 4.4.4 Threats for Personal Clouds: DDoS

A Distributed Denial of Service (DDoS) can be defined as the attempt to disrupt the legitimate use of a service [131]. Normally, a DDoS attack is normally accompanied by some form of fraudulent resource consumption in the victim's side.

Surprisingly, we found that DDoS attacks to U1 are *more frequent* than one can reasonably expect. More specifically, we found 3 evidences of such attacks in our traces (January 15, 16 and February 6)[1]. These DDoS attacks had as objective to *share illegal content* through the U1 infrastructure.

As visible in Fig. 4.5, all the attacks resulted in a dramatic increase of the number of session and authentication requests per hour —both events related to the management of user sessions. Actually, the authentication activity under attack was 5 to 15 times higher than usual, which directly impacts the Canonical's *authentication subsystem*.

[1]Our interviews with Canonical engineers confirmed that these activity spikes correspond to DDoS attacks, instead of a software release or any other legitimate event.

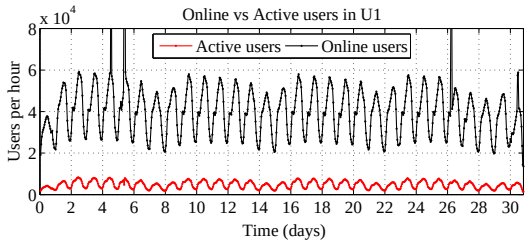



**Figure 4.6:** Fraction of active users per hour.

The situation for API servers was even worse: during the second attack (01/16) API servers received an activity 245x higher than usual, whereas during the first (01/15) and last (02/06) attacks the activity was 4.6x and 6.7x higher than normal, respectively. Therefore, the most affected components were the API servers, as they serviced both *session and storage operations*.

We found that these attacks consisted on sharing a *single user id* and its credentials to distribute content across thousands of desktop clients. The nature of this attack is similar to the *storage leeching problem* reported in [30], which consists of exploiting the freemium business model of Personal Clouds to illicitly consume bandwidth and storage resources.

Also, the reaction to these attacks was not automatic. U1 engineers manually handled DDoS by means of deleting fraudulent users and the content to be shared. This can be easily seen on the storage activity for the second and third attack, which decays within one hour after engineers detected and responded to the attack.

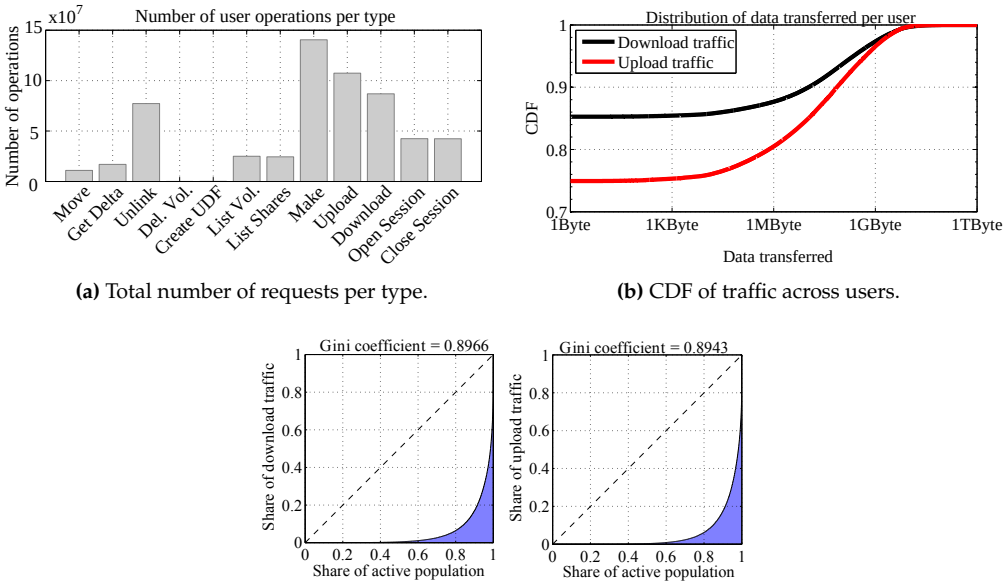
These observations confirm that Personal Clouds are a suitable target for attack as other Internet systems, and that these situations are indeed common. We believe that further research is needed to build and apply secure storage protocols to these systems, as well as new countermeasures to automatically react to this kind of threats.

## 4.5 Understanding User Behavior

Understanding the behavior of users is a key source of information to optimize large-scale systems. This section provides several insights about the behavior of users in U1.

### 4.5.1 Distinguishing Online from Active Users

**Online and active users**. We consider a user as *online* if his desktop client exhibits any form of interaction with the server. This includes automatic client requests involved in maintenance or notification tasks, for which the user is not responsible for. Moreover, we consider a user as




**(a)** Total number of requests per type.

**(b)** CDF of traffic across users.

**(c)** Lorenz curve of traffic vs active users.

**Figure 4.7:** User requests and consumed traffic in U1 for one month.

*active* if he performs data management operations on his volumes, such as uploading a file or creating a new directory.

Fig. 4.6 offers a time-series view of the number of online and active users in the system per hour. Clearly, online users are more numerous than active users: The percentage of active users ranges from 3.49% to 16.25% during the whole trace. This observation reveals that the actual storage workload that U1 supports is light compared to the potential usage of its user population, and gives a sense on the scale and costs of these services with respect to their popularity.

**Frequency of user operations**. Here we examine how frequent the protocol operations are in order to identify the hottest ones. Fig. 4.7a depicts the absolute number of each operation type. As shown in this figure, the most frequent operations correspond to *data management operations*, and in particular, those operations that relate to the download, upload and deletion of files.

This result is very interesting, because it proves that the U1 protocol scales well, since the operations that users issue to manage their sessions and are typically part of the session start up such as `ListVolumes` are significantly less frequent. And consequently, the major part of the processing burden comes from active users as desired. This is essentially explained by the

fact that the U1 desktop client does not need to regularly poll the server during idle times, thereby limiting the number of requests not linked to data management.

As we will see in 4.6, the frequency of API operations will have an immediate impact on the back-end performance.

**Traffic distribution across users**. Now, we turn our attention to the distribution of consumed traffic across users. In Fig. 4.7b we observe an interesting fact: in one month, only 14% of users downloaded data from U1, while uploads represented 25%. This indicates that a minority of users are responsible for the storage workload of U1.

To better understand this, we measure how (un)equal the traffic distribution across active users is (170K users in the trace are *active*). To do so, we resort to the Lorenz curve and the Gini coefficient[1] as indicators of inequality. The Gini coefficient varies between 0, which reflects complete equality, and 1, which indicates complete inequality (i.e., only one user consumes all the traffic). The Lorenz curve plots the proportion of the total income of the population ($y$ axis) that is cumulatively earned by the bottom $x$% of the population. The line at 45 degrees thus represents perfect equality of incomes.

Fig. 4.7c reports that the consumed traffic across active users is very unequal. That is, the Lorenz curve is very far from the diagonal line and the Gini coefficient is close to 1. The reason for this inequality is clear: 1% of active users account for the 65.6% of the total traffic (147.52TB). Providers may benefit from this fact by identifying and treating these users more efficiently.

**Types of user activity**. To study the activity of users, we used the same user classification than Drago et al. in [11]. So we distinguished among *occasional*, *download/upload only* and *heavy* users. A user is *occasional* if he transfers less than 10KB of data. Users that exhibit more than three orders of magnitude of difference between upload and download (e.g., 1GB versus 1MB) traffic are classified as either *download-only* or *upload-only*. The rest of users are in the *heavy* group.

Given that, we found that 85.92% of *all* users are occasional (mainly *online* users), 7.07% upload-only, 2.12% download-only and 4.87% are heavy users. Our results clearly differ from the ones reported in [11], where users are 30% occasional, 7% upload-only, 26% download-only and 37% heavy. This may be explained by two reasons: (i) the usage of Dropbox is more extended that the usage of U1, and (ii) users in a university campus are more active than other types of users captured in our trace.

[1] `http://en.wikipedia.org/wiki/Gini_coefficient`

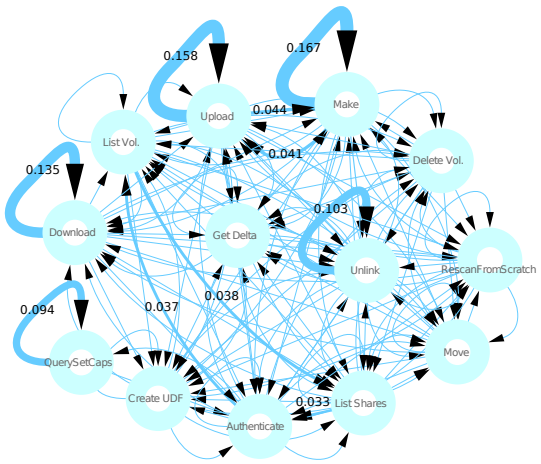



**Figure 4.8:** Desktop client transition graph through API operations. Global transition probabilities are provided for main edges.

### 4.5.2 Characterizing User Interactions

**User-centric request graph**. To analyze how users interact with U1, Fig. 4.8 shows the sequence of operations that desktop clients issue to the server in form of a graph. Nodes represent the different protocol operations executed. And edges describe the transitions from one operation to another. The width of edges denotes the global frequency of a given transition. Note that this graph is user-centric, as it aggregates the different sequence of commands that every user executes, not the sequence of operations as they arrive to the metadata service.

Interestingly, we found that the *repetition of certain operations* becomes really frequent across clients. For instance, it is highly probable that when a client transfers a file, the next operation that he will issue is also another transfer —either upload or download. This phenomenon can be partially explained by the fact that many times users synchronize data at *directory granularity*, which involves repeating several data management operations in cascade. File editing can be also a source of recurrent transfer operations. This behavior can be exploited by predictive data management techniques in the server side (e.g., download prefetching).

Other sequences of operations are also highlighted in the graph. For instance, once a user is authenticated, he usually performs a `ListVolumes` and `ListShares` operations. This is a regular initialization flow for desktop clients. We also observe that `Make` and `Upload` operations are quite mixed, evidencing that for uploading a file the client first needs to create the metadata entry for this file in U1.

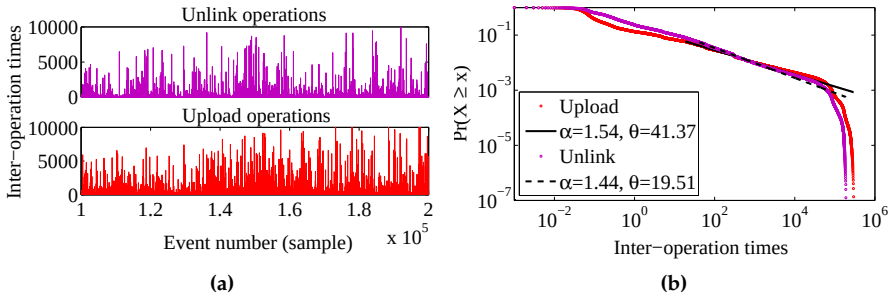



**Figure 4.9:** Time-series view of inter-arrival times and their approximation to a power-law.

**Burstiness in user operations**. Next, we analyze interarrival times between consecutive operations of the same user. We want to verify whether inter-operation times are Poissonian or not, which may have important implications to the back-end performance. To this end, we followed the same methodology proposed in [132, 133], and obtained a time-series view of `Unlink` and `Upload` inter-operation times and their approximation to a power-law distribution in Fig. 4.9.

Fig. 4.9a exhibits large spikes for both `Unlink` and `Upload` operations, corresponding to *very long inter-operation times*. This is far from an exponential distribution, where long inter-operation times are negligible. This shows that the interactions of users with U1 are not Poissonian [132].

Now, we study if the `Unlink` and `Upload` inter-operation times exhibit *high variance*, which indicates *burstiness*. In all cases, while not strictly linear, these distributions show a downward trend over almost six orders of magnitude. This suggests that high variance of user inter-arrival operations is present in time scales ranging from seconds to several hours. Hence, users issue requests in a *bursty non-Poissonian way*: during a short period a user sends several operations in quick succession, followed by long periods of inactivity. A possible explanation to this is that users manage data at the *directory granularity*, thereby triggering multiples operations to keep the files inside each directory in sync.

Nevertheless, we cannot confirm the hypothesis that these distributions are heavy-tailed. Clearly, Fig. 4.9b visually confirms that the empirical distributions of user `Unlink` and `Upload` inter-arrivals can be only approximated with $P(x) \approx x^{-\alpha}, \forall x > \theta$, $1 < \alpha < 2$, for a central region of the domain.

We also found that metadata operations follow more closely a power-law distribution than data operations. The reason is that the behavior of metadata inter-operation times are not

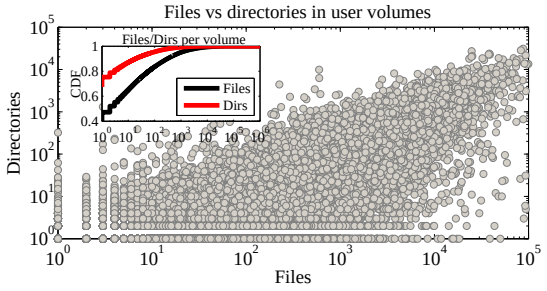



**Figure 4.10:** Files and directories per volume.

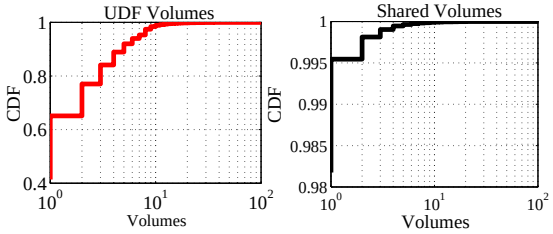



**Figure 4.11:** Distribution of shared/udf volumes across users.

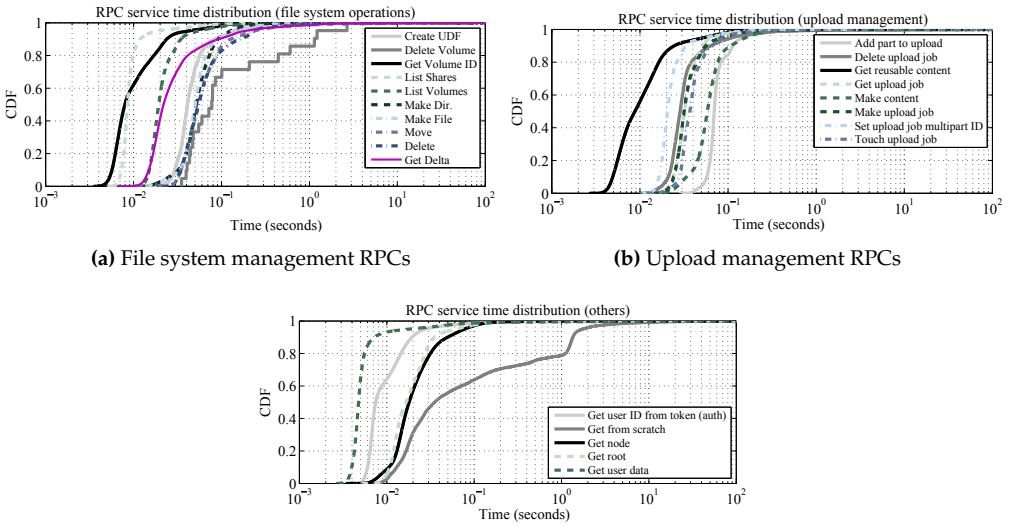
affected by the actual data transfers.

In conclusion, we can see that user operations are bursty, which has strong implications to the operation of the back-end servers (4.6).

### 4.5.3 Inspecting User Volumes

**Volume contents**. Fig. 4.10 illustrates the relationship between files and directories within user volumes. As usual, files are much more numerous than directories. And we have that over 60% of volumes have been associated with at least one file. For directories, this percentage is only of 32%, but there is a strong correlation between the number of files and directories within a volume: Pearson correlation coefficient is 0.998. What is relevant is, however, that a small fraction of volumes is heavy loaded: 5% of user volumes contain more than 1,000 files.

**Shared and UDF volumes**. At this point, we study the distribution of user-defined/shared volumes across users. As pointed out by Canonical engineers, sharing is not a popular feature of U1. Fig. 4.11 shows that only 1.8% of users exhibits at least one shared volume. On the contrary, we observe that user-defined volumes are much more popular; we detected user-defined volumes in 58% of users —the rest of users only use the root volume. This shows that the majority of users have some degree of expertise using U1.




**(a)** File system management RPCs

**(b)** Upload management RPCs

**(c)** Other read-only RPCs

**Figure 4.12:** Distribution of RPC service times accessing to the metadata store.

Overall, these observations reveal that U1 was used more as a storage service rather than for collaborative work.

## 4.6 Metadata Back-end Analysis

In this section, we focus on the interactions of RPC servers against the metadata store. We also quantify the role of the Canonical authentication service in U1.

### 4.6.1 Performance of Metadata Operations

Here we analyze the performance of RPC operations that involve contacting the metadata store.

Fig. 4.12 illustrates the distribution of service times of the different RPC operations. As shown in the figure, all RPCs exhibit *long tails of service time distributions*: from 7% to 22% of RPC service times are very far from the median value. This issue can be caused by several factors, ranging from interference of background processes to CPU power saving mechanisms, as recently argued by Li et al. in [125].

Also useful is to understand the relationship between the service time and the frequency of each RPC operation. Fig. 4.13 presents a scatter plot relating RPC median service times with their frequency, depending upon whether RPCs are of type *read*, *write/update/delete* or

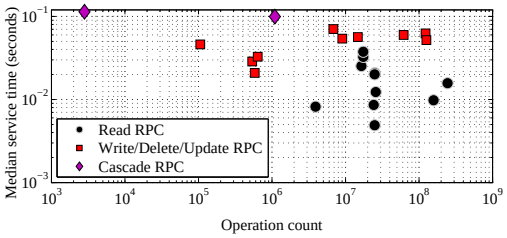



**Figure 4.13:** We classified all the U1 RPC calls into 3 categories, and every point in the plot represents a single RPC call. We show the median service time vs frequency of each RPC (1 month).

*cascade*, i.e., whether other operations are involved. This figure confirms that the type of an RPC strongly determines its performance. First, *cascade* operations (`delete_volume` and `get_from_scratch`) are the slowest type of RPC —more than one order of magnitude slower compared to the fastest operation. Fortunately, they are relatively infrequent. Conversely, *read* RPCs, such as `list_volumes`, are the fastest ones. Basically, this is because *read* RPCs can exploit lockless and parallel access to the pairs of servers that form database shards.

*Write/update/delete* operations (e.g. `make_content`, or `make_file`) are slower than most *read* operations, but exhibiting comparable frequencies. This may represent a performance barrier for the metadata store in scenarios where users massively update metadata in their volumes or files.

### 4.6.2 Load Balancing in U1 Back-end

We are interested in analyzing the internal load balancing of both API servers and shards in the metadata store. In the former case, we grouped the processed API operations by physical machine. In the latter, we distributed the RPC calls contacting the metadata store across 10 shards based on the user id, as U1 actually does. Results appear in Fig. 4.14, where bars are mean load values and error lines represent the standard deviation of load values across API servers and shards per hour and minute, respectively.

Fig. 4.14 shows that server load presents a *high variance across servers*, which is symptom of bad load balancing. This effect is present irrespective of the hour of the day and is more accentuated for the metadata store, for which the time granularity used is smaller. Thus, this phenomenon is visible in short or moderate periods of time. In the long term, the load balancing is adequate; the standard deviation across shards is only of 4.9% when the whole trace is taken.

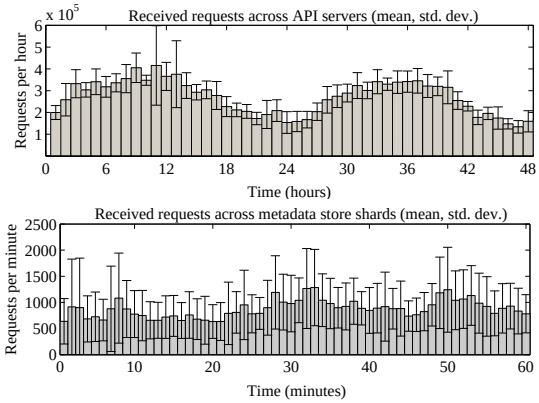



**Figure 4.14:** Load balancing of U1 API servers and metadata store shards.

Three particularities should be understood to explain the poor load balancing. First, user load is *uneven*, i.e., a small fraction of users is very active whereas most of them present low activity. Second, the cost of operations is *asymmetric*; for instance, there are metadata operations whose median service time is 10x higher than others. Third, users display a *bursty* behavior when interacting with the servers; for instance, they can synchronize an entire folder. So, operations arrive in a correlated manner.

We conclude that the load balancing in the U1 back-end can be significantly improved, which is object of future work.

### 4.6.3 Authentication Activity & User Sessions

**Time-series analysis**. Users accessing the U1 service should be authenticated prior to the establishment of a new session. To this end, U1 API servers should contact a separate and shared authentication service of Canonical.

Fig. 4.15 depicts a time-series view of the session management load that API servers support to create and destroy sessions, along with the corresponding activity of the authentication subsystem. In this figure, we clearly observe that the authentication and session management activity is closely related to the habits of users. In fact, daily patterns are evident. The authentication activity is 50% to 60% higher in the central hours of the day than during the night periods. This observation is also valid for week periods: on average, the maximum number of authentication requests is 15% higher on Mondays than on weekends. Moreover, we found that 2.76% of user authentication requests from API servers to the authentication service fail.

**Session length**. Upon a successful authentication process, a user's desktop client creates a new U1 session.

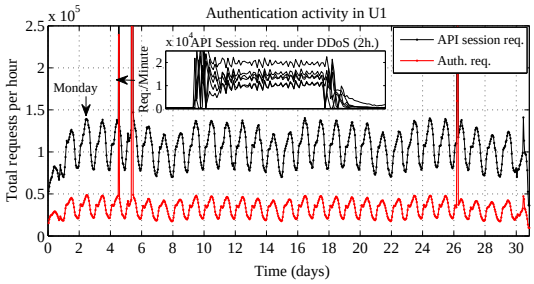



**Figure 4.15:** API session management operations and authentication service requests.

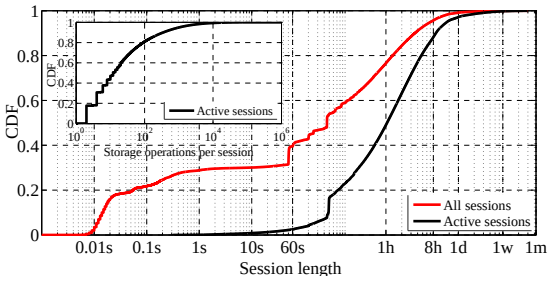



**Figure 4.16:** Distribution of session lengths and storage operations per session.

U1 sessions exhibit a similar behavior to Dropbox *home* users in [11] (Fig. 4.15). Concretely, 97% of sessions are shorter than 8 hours, which suggests a strong correlation with user working habits. Moreover, we also found that U1 exhibits a high fraction of very short-lived sessions (i.e. 32% shorter than 1s.), probably due to the operation of NAT and firewalls that normally mediate between clients and servers [134]. Overall, Fig. 4.15 suggests that *domestic users are more representative* than other specific profiles, such as university communities, for describing the connection habits of an entire Personal Cloud user population.

We are also interested in understanding the data management activity related to U1 sessions. To this end, we differentiate those sessions that exhibited any type of data management operation (e.g., upload, download, delete file, etc.) during their lifetime, namely, *active sessions*.

First, we observed that the majority of U1 sessions (and, therefore, TCP connections) do not involve any type of data management. That is, only 5.57% of connections in U1 are active (2.37M out of 42.5M), which, in turn, tend to be much longer than *cold* ones. From a back-end perspective, the unintended consequence is that a fraction of server resources is *wasted keeping alive TCP connections* of cold sessions.

Moreover, similarly to the distribution of user activity, the inner plot of Fig. 4.15 shows that 80% of active sessions exhibited at most 92 storage operations, whereas the remaining

20% accounted for 96.7% of all data management operations. Therefore, there are sessions much more active than others.

A provider may benefit from these observations to optimize session management. That is, depending on a user's activity, the provider may wisely decide if a desktop client works in a *pull* (cold sessions) or *push* (active sessions) fashion to limit the number of open TCP connections [135].

## 4.7 Discussion and Conclusions

In this Chapter, we focus on understanding the nature of Personal Cloud services by presenting the internal structure and measurement study of UbuntuOne (U1). The objectives of our work are threefold: (i) to unveil the internal operation and infrastructure of a real-world provider, (ii) to reconfirm, expand and contribute observations on these systems to generalize their characteristics, and (iii) to propose potential improvements for these systems.

Our study unveils several aspects that U1 *shares* with other large-scale Personal Clouds. For instance, U1 presents clear similarities with Dropbox regarding the way of decoupling data and metadata of users, which seems to be a standard design for these systems [34]. Also, we found characteristics in the U1 workload that reconfirm observations of prior works [11, 24] regarding the *relevance of file updates*, the *effectiveness of deduplication* or the execution of user operations in *long sequences*, among other aspects. Therefore, our analysis and the resulting dataset will enable researchers to get closer to the nature of a real-world Personal Cloud.

Thanks to the scale and back-end perspective of our study, we expanded and contributed insights on these services. That is, we observed that the distribution of activity across users in U1 is even *more skewed* than in Dropbox [11] or that the behavior of domestic users dominate session lengths in U1 compared to other user types (e.g., university). Among the novelties of this work, we modeled the *burstiness of user operations*, we analyzed the *behavior of files* in U1, we provided evidences of *DDoS attacks* to this service, and we illustrated the performance of the U1 *metadata back-end*.

An orthogonal conclusion that we extract from our study is that understanding *the behavior of users is essential* to adapt the system to its actual demands and reduce costs. In the following, we relate some of our insights to the running costs of U1 as well as potential optimizations, which may be of independent interest for other large-scale systems:

*Optimizing storage matters*. A key problem to the survival of U1 was the growing costs of outsourcing data storage [136], which is directly related to the data management techniques integrated in the system. For instance, the fact that file updates were responsible for 18.5% of upload traffic in U1, mainly due to the lack of delta updates in the desktop client, gives an idea of the margin of improvement (Section 4.4.1). Actually, we confirmed that a simple optimization like file-based deduplication could readily save 17% of the storage costs. This calls to further research and the application of advanced data reduction techniques, both at the client and server sides.

*Take care of user activity*. This observation is actually very important, as we found that 1% of U1 users that manage files generated 65% of traffic (Section 4.5.1), showing a weak form of the Pareto Principle. That is, a very small fraction of the users represented most of the OPEX for U1. A natural response may be to limit the activity of free accounts, or at least to treat active users in a more cost effective way. For instance, distributed caching systems like Memcached, data prefetching techniques, and advanced sync deferment techniques [24] could easily cut the operational costs down. On the other hand, U1 may benefit from cold/warm storage services (e.g., Amazon Glacier, f4 [130]) to limit the costs related to most inactive users.

*Security is a big concern*. Another source of expense for a Personal Cloud is related to its exploitation by malicious parties. In fact, we found that DDoS attacks aimed at sharing illegal content via U1 are indeed frequent (Section 4.4.4).The risk that these attacks represent to U1 is in contrast to the limited automation of its countermeasures. We believe that further research is needed to integrate secure storage protocols and automated countermeasures for Personal Clouds. In fact, understanding the common behavior of users in a Personal Cloud (e.g., storage, content distribution) may provide clues to automatically detect anomalous activities [95].

**Conclusions**. This Chapter presented the first measurement of a global-scale Personal Cloud back-end, in particular of U1. First, we have described in depth its protocol, the major elements of its architecture and the internal metadata infrastructure. Second, we have contributed central insights regarding the storage workload, user behavior and the metadata back-end of the U1 service. Finally, we have discussed the implications and opportunities of pursuing a successful operation of U1, which may be of general interest for large scale systems and promote research in this field.

# 5

# Actively Measuring Personal Clouds: Analysis and Abuse

**Summary**

Understanding the transfer QoS of cloud storage services is of great interest to end-users, researchers and developers. In this Chapter, we present a measurement study of three major Personal Clouds: Dropbox, Box and SugarSync. Actively accessing to free accounts through their REST APIs, we analyzed important aspects to characterize their transfer QoS, such as transfer speed, variability and failure rate. Moreover, we demonstrate that combining open APIs and free accounts may lead to abuse by malicious parties. We also propose countermeasures to limit the impact of abusive applications in this scenario.

*The papers with the results of this Chapter appeared in [29, 30]*

## 5.1 Introduction

The hype around the Personal Cloud model [123] has promoted the appearance of a myriad of very competitive offerings (e.g., Dropbox, Box) that nowadays populates the market. This makes, in turn, Personal Clouds to aggressively react and improve their service to retain their market share. Essentially, Personal Clouds make their offering more interesting for new customers adding *innovative functionalities* to their service and delivering *freemium* accounts.

First, Personal Clouds are incorporating a large corpus of value-added functionalities to their service (e.g. collaborative editors, media viewers). In this sense, major companies provide *open REST APIs* for developers to create clever applications that make their service even more attractive. From a functional perspective, these APIs enable an application to transfer files to/from *user accounts*, blurring the lines between a Personal Cloud service and a pure IaaS provider as Amazon S3. Such a powerful abstraction hides the complexity of block-level data management and constitutes a rich substrate to cultivate a developer ecosystem.

Secondly, most vendors offer *free accounts* to lure new customers and gain market share. These free accounts normally include reduced storage space, as well as virtually unlimited transfers. Moreover, as paid accounts, free accounts provide standard functionalities, such as access from syncing desktop clients and Web front-ends. The impact of this *freemium* business model is remarkable: In 2012, from the 100 million of Dropbox users only 4% are estimated to pay for storage [137]. To better understand these elements is the goal of this Chapter.

In this Chapter, our objectives are: (i) To *measure and characterize the transfer QoS* of Personal Cloud REST APIs, and (ii) understand the potential *exploitability of these APIs over free accounts*.

The motivation of our first objective is that, despite their broad adoption, very little is known about the transfer QoS of Personal Cloud REST APIs. Furthermore, there is no public information about the control policies that vendors may enforce, as well as the factors impacting on their service performance. In our view, exploring these services is specially interesting in the case of *free accounts*, since most users *freemium* users.

Thus, we present a measurement study of various Personal Clouds. Concretely, during two months, we have actively measured the REST API service of Dropbox, Box and SugarSync free accounts. We gathered information from more than $900,000$ storage operations, transferring around 70TB of data. We analyzed important aspects to characterize their QoS, such as *in/out transfer speed*, *service variability* and *failure rate*. To our knowledge, this work is the first to deeply explore many facets of these popular services and reveals new insights. Some of our most relevant research observations are summarized in Table 5.1.

| *Measurement of REST APIs* | **Finding** | **Implications and Opportunities** |
|---|---|---|
| Measuring Transfer Performance (4.4) | The transfer performance of these services greatly varies from one provider to another. | This is a valuable insight for designers and developers to select the best vendor for their needs. |
| | North American clients experience transfers several times faster than European ones for the same Personal Cloud. | The geographic location of a client importantly impacts on the speed of transfers. |
| | In general, transfer speeds of files can be approximated using well-known statistical distributions. | This opens the door to create Personal Cloud simulation environments. |
| Variability of Transfer Performance (4.5) | We found that uploads are more variable than downloads. | Personal Clouds tend to perform a more restrictive bandwidth control to outgoing traffic. |
| | The variability of transfers depends on several factors, such as the traffic type (in/out) or the hour of the day. Actually, we found daily patterns in the Dropbox service. | This represents a source of uncertainty to users and developers employing these services. |
| Service Failures and Breakdowns (4.6) | These services are in general reliable and, in some cases, service failures can be modeled as a Poisson process. | This allows researchers to develop tractable analytical models for Cloud storage. |
| | We observed a radical change in the transfer speed of SugarSync in late May 2012. | This suggests that Personal Clouds may change their freemium QoS unexpectedly, due to internal policy changes or agreements. |

**Table 5.1:** Summary of some of our most important findings and their implications.

Our second objective in this Chapter is to unveil a form of abuse that malicious parties may perpetrate on Personal Clouds. That is, the unintended consequence of combining REST APIs and free accounts is that these companies are exposing *automated access to a free storage infrastructure*, which may lead to *abuse by malicious parties*. Nothing prevents a malicious user from acquiring an arbitrary number of free accounts from a single vendor and access to them via REST APIs, given the quick registration process that it requires. Furthermore, that user may aggregate accounts from various providers to build a larger and even better storage facility by exploiting storage diversity.

Although aggregating free accounts might not be interpreted as an attack by itself, thanks to open APIs these accounts can be used to materialize illicit actions against Personal Clouds. For instance, as we reported in Chapter 4, users may perpetrate DDoS attacks, fraudulent resource consumption, or they could use free accounts as a storage layer to support abusive applications. We call this vulnerability the *storage leeching problem*.

We describe the roots of the storage leeching problem and shows how easy is to benefit from it. To this end, we implemented `Boxleech`: a proof-of-concept file-sharing application able to distribute digital content by abusing Personal Clouds. This application transparently aggregates the limited-space free accounts from multiple providers into a single larger storage space while achieving better transfer performance than that received from a single provider. Considering this problem, we provide some discussion about possible countermeasures to deliver a more secure API service to developers.

The contributions of the present Chapter can be summarized as follows:

- We present an active measurement of various Personal Clouds to characterize their transfer QoS via the available REST API service.
- We unveil a new form of abuse that malicious parties may exploit for consuming resources of Personal Clouds to carry out illicit activities.
- We discuss the most important observations of our measurement and their implications, as well as potential countermeasures to ameliorate the impact of storage leeching.
- We contribute the collected measurement data set and we make it publicly available for the research community[1].

The rest of this Chapter is organized as follows. Our methodology is described in Section 5.2. The measurement data analysis appears in Section 5.3. We describe the storage leeching problem in Section 5.4. The design of `Boxleech` appears in Section 5.5. In Section 5.6 we show the evaluation of `Boxleech` compared with Personal Cloud desktop clients. To close the Chapter, in Section 5.7 we provide technical discussion on the implications of our measurement and the countermeasures to the storage leeching problem, as well as our conclusions.

## 5.2 Measurement Methodology

From May 10, 2012, to July 15, 2012, we installed several vantage points in our university network (*Universitat Rovira i Virgili*, Spain) and PlanetLab [138] to measure the performance of three of the major Personal Cloud services in the market: Dropbox[2], Box[3] and SugarSync[4]. The measurement methodology was based on the REST interfaces that these three Personal Cloud storage services provide to developers.

As we discussed in Chapter 2, Personal Clouds provide REST APIs, along with their client implementations, to make it possible for developers to create novel applications. These APIs incorporate authorization mechanisms (OAuth [51]) to manage the credentials and tokens that grant access to the files stored in user accounts. A developer first registers an application in the Cloud provider website and obtains several tokens. As a result of this process, and once

[1] `http://ast-deim.urv.cat/trac/pc_measurement`
[2] `http://www.dropbox.com`
[3] `http://www.box.net`
[4] `http://www.sugarsync.com`

the user has authorized that application to access his storage space, the Personal Cloud storage service gives to the developer an *access token*. Including this *access token* in each API call, the application can operate on the user data.

There are two types of API calls: *meta-info* and *data management* calls. The former type refers to those calls that retrieve information about the state of the account (i.e., storage load, filenames), whereas the latter ones are those calls targeted at managing the stored files in the account. We will analyze the performance of the most important data management calls: `PUT` and `GET`, which serve to store and retrieve files.

### 5.2.1 Measurement Platform

We employed two different platforms to execute our tests: University laboratories and PlanetLab. The reason behind this is that our labs contain *homogeneous and dedicated machines* that are under our control, while PlanetLab allows the analysis of each service from *different geographic locations*.

*University laboratories*: We gathered 30 machines belonging to the same laboratory to perform the measurement. These machines were Intel Core2 Duo equipped with 4GB DDR2 RAM. The employed operating system was a Debian Linux distribution. Machines were internally connected to the same switch via a 100Mbps Ethernet links.

*PlanetLab*: We collected 40 PlanetLab nodes divided into two geographic regions: Western Europe and North America. This platform is constituted by heterogeneous (bandwidth, CPU) machines from several universities and research institutes. Moreover, there were two points to consider when analyzing data coming from PlanetLab nodes: i) Machines might be concurrently used by other processes and users, and ii) The quota system of these machines limited the amount of in/out data transferred daily.

Specifically, we used the PlanetLab infrastructure for a high-level assessment of Personal Clouds depending on the client's geographic location. However, the mechanisms to enforce bandwidth quotas in PlanetLab nodes may induce the appearance of artifacts in bandwidth traces. This made PlanetLab not suitable for a fine-grained analysis in our context.

| Location | Op. Type | Operations | Transferred Data |
|---|---|---|---|
| University Labs | GET | 168,396 | 13.509 TB |
| | PUT | 247,210 | 15.945 TB |
| PlanetLab | GET | 354,909 | 31.751 TB |
| | PUT | 129,716 | 9.803 TB |

**Table 5.2:** Summary of Measurement Data (May 10 − July 15)

### 5.2.2 Workload Model

Usually, Personal Cloud services impose file size limitations to their REST interfaces, for we used only files of four sizes to facilitate comparison: 25MB, 50MB, 100MB and 150MB[1]. This approach provides an appropriate substrate to compare all providers with a large amount of samples of equal-size files. Thanks to this, we could observe performance variations of a single provider managing files of the same size.

We executed the following workloads:

*Up/Down Workload*. The objective of this workload was twofold: Measuring the maximum up/down transfer speed of operations and detecting correlations between the transfer speed and the load of an account. Intuitively, the first objective was achieved by alternating upload and download operations, since the provider only needed to handle one operation per account at a time. We achieved the second point by acquiring information about the load of an account in each API call.

The execution of this workload was continuously performed at each node as follows: First, a node created synthetic files of a size chosen at random from the aforementioned set of sizes. That node uploaded files until the capacity of the account was full. At this point, that node downloaded all the files also in random order. After each download, the file was deleted.

*Service Variability Workload*. This workload maintained in every node a nearly continuous upload and download transfer flow to analyze the performance variability of the service over time. This workload provides an appropriate substrate to elaborate a time-series analysis of these services.

The procedure was as follows: The upload process first created files corresponding to each defined file size which were labeled as "reserved", since they were not deleted from the account. By doing this we assured that the download process was never interrupted, since at

[1]Although the official limitation in some cases is fixed to 300MB per file, we empirically proved that uploading files larger than 200MB is highly difficult. In case of Box this limitation is 100MB.

least the reserved files were always ready for being downloaded. Then, the upload process started uploading synthetic random files until the account was full. When the account was full, this process deleted all files with the exception of the reserved ones to continue uploading files. In parallel, the download process was continuously downloading random files stored in the account.

Finally, we executed the experiments in different ways depending on the chosen platform. In the case of PlanetLab, we employed the *same machines in each test*, and therefore, we needed to sequentially execute all the combinations of workloads and providers. This minimized the impact of hardware and network heterogeneity, since all the experiments were executed in the same conditions. On the contrary, in our labs we executed in parallel a certain workload for all providers (i.e. assigning 10 machines per provider). This provided two main advantages: The measurement process was substantially faster, and fair comparison of the three services was possible for the same period of time.

We depict in Table 5.2 the total number of storage operations performed during the measurement period.

### 5.2.3 Setup, Software and Data Collection

Prior to the start of our experiments, we created around 150 new user free accounts from the targeted Personal Clouds. That is 120 new accounts for PlanetLab experiments (40 nodes $\times$ 3 Personal Clouds), and 30 accounts for the experiments in our labs (10 accounts per Personal Cloud deployed in 30 machines). We also registered as developers 35 applications to access the storage space of user accounts via REST APIs, obtaining the necessary tokens to authenticate requests. We assigned to every node a single new free account with access permission to the corresponding application. The information of these accounts was stored in a database hosted in our research servers. Thus, nodes executing the measurement process were able to access the account information remotely.

Measurement processes were implemented as Unix and Python scripts that ran in every node. These scripts employed third party tools during their execution. For instance, to synchronize tasks, such as logging and starting/finishing experiments, we used the `cron` time-based job scheduler. To gather bandwidth information we used `vnstat`, a tool that keeps a log of network traffic for a selected interface. Nodes performed storage operations against Personal Clouds thanks to the API implementations released in their websites.

The measurement information collected in each storage operation was sent periodically from every node to a database hosted in our research servers. This automatic process facilitated the posterior data processing and exploration. The measurement information that nodes sent to the database describes several aspects of the service performance: operation type, bandwidth trace, file size, start/end time instants, time zone, capacity and load of the account, and failure information.

## 5.3 Measuring Personal Cloud REST APIs

### 5.3.1 Transfer Capacity of Personal Clouds

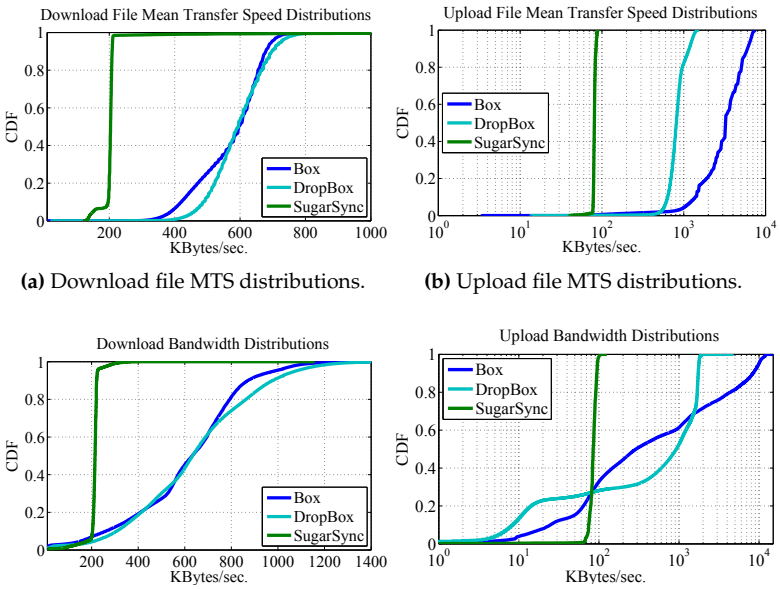
In this section, the transfer capacity of Box, Dropbox and SugarSync is characterized using the following indicators:

- *File Mean Transfer Speed (MTS)*. This metric is defined as *the ratio of the size of a file, $S$, to the time, $T$, that was spent to transfer it: $MTS = S/T$ (KBytes/sec)*.
- *Bandwidth Distributions*. We define as a *bandwidth trace* the set of values that reflects the transfer speed of a file at regular intervals of 2 secs. To obtain a single empirical distribution, we aggregated the bandwidth traces of all the transfers separated by uploads and downloads. We refer to the resulting empirical distribution as the *aggregated bandwidth distribution*.

**Transfer speeds**. Fig. 5.1 reports these metrics for both workloads (up/down and service variability) executed in our university labs during 10 days. First, Fig. 5.1 evidences an interesting fact: *Personal Clouds are heterogeneous in terms of transfer speed*. For instance, Fig. 5.1b shows that Box and Dropbox present an upload MTS several times faster than SugarSync. The same observation holds for downloads. Moreover, the heterogeneity of these services also depends on the *traffic type* (in/out). This can be appreciated by comparing Fig. 5.1a with Fig. 5.1b: *Dropbox exhibits the best download MTS while Box presents the fastest uploads.*

This proves that *the transfer performance of these services greatly varies among providers*, and consequently, developers should be aware of this in order to select an adequate provider.

Among the examined Personal Clouds, Dropbox and SugarSync are *resellers* of major Cloud storage providers (Amazon S3 and Carpathia Hosting, respectively). On the other hand, Box claims to be owner of several datacenters. In our view, it is interesting to analyze this Cloud ecosystem and the possible implications to the service delivered to end-users.




**(a)** Download file MTS distributions.

**(b)** Upload file MTS distributions.

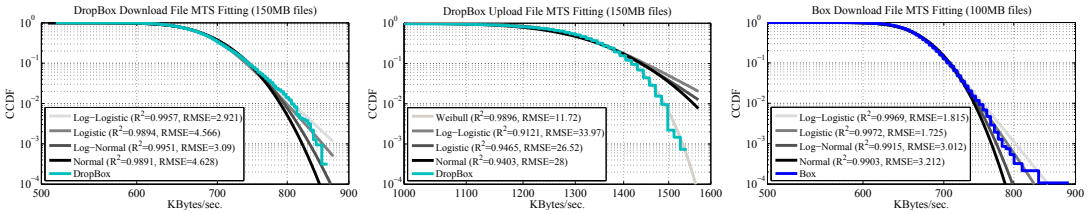
**(c)** Download bandwidth distributions.

**(d)** Upload bandwidth distributions.

**Figure 5.1:** Transfer capacity of Box, Dropbox and SugarSync free account REST API services. The data represented in these figures corresponds to the aggregation of the up/down and service variability workloads during 10 days (June/July 2012) in our university laboratories.

In this sense, in Fig. 5.1 we observe that Personal Clouds apply *distinct internal control policies* to the inbound/outbound bandwidth provided to users. To wit, both Dropbox and Box exhibit an *upload transfer capacity remarkably better than the download capacity*. This means that the datacenter outgoing traffic is more *controlled and restricted* than the incoming traffic. This agrees well with the current pricing policies of major Cloud providers (Amazon S3, Google Storage) which do not charge inbound traffic whereas the outbound traffic is subject to specific rates (see `http://aws.amazon.com/en/s3/pricing/`).

In SugarSync, both the upload and download transfer speeds are constant and low. Interestingly, SugarSync presents slightly faster downloads than uploads, though only a small fraction of downloads (less than 1%) exhibits a much higher transfer speed than the rest. These observations are also supported by Fig. 5.1c and Fig. 5.1d: the captured download bandwidth values fall into a small range $[200, 1300]$ KB/sec. Also, the shape of these distributions are not steep, which reflects that there is a strong control in the download bandwidth. On the contrary, upload bandwidth distributions present more irregular shapes and they cover a wider range of values, specially for Box. As a possible explanation to this behavior, the experiments




**(a)** Dropbox down. file MTS fitting (150MB files).

**(b)** Dropbox up. file MTS fitting (150MB files).

**(c)** Box down. file MTS fitting (100MB files).

**Figure 5.2:** Distribution fittings of upload/download file mean transfer speeds (MTS) of the examined Personal Clouds (up/down workload, university labs).

of Fig. 5.1 were executed from our university labs (Spain) to exclude the impact of geographic heterogeneity. Considering the fact that the majority of Personal Cloud datacenters are located in USA [11], this may have implications in the cost of the traffic sent to Europe. This could motivate the enforcement of more restrictive bandwidth control policies to the outbound traffic.

**Characterization of transfers**. To characterize the transfer performance of both Dropbox and Box (the constant behavior of SugarSync deserved no further analysis), three checks were made to determine the shape of the transfer distributions with sufficient confidence. We used the same methodology of [139].

First, visual inspection of per-file MTS distributions against the most similar standard distributions was performed. Second, we performed a linear regression analysis on the best-fit lines of the quantile-quantile plots from the fitted distributions and empirical data. From this analasys, we obtained the coefficient of determination, $R^2 \in [0,1]$. A value of $R^2$ close to 1 signals that the candidate distribution fits the data. Finally, we used the Kolmogorov-Smirnov (KS) test to assess the statistical validity of the fittings. Essentially, this test is used to check whether a fitted distribution matches the empirical distribution by finding the maximum differences between both distributions[1].

As seen in Fig. 5.2a and 5.2c, both Dropbox and Box download file MTS *can be approximated using log-logistic or logistic distributions, respectively*. This argument is supported by the coefficient of determination, $R^2$, which in the case of Box is $R^2 = 0.9972$, and for Dropbox is

[1]Chi-square test was not used since it works well only when the number of items that falls into any particular bin is approximately the same. However, it is relatively difficult to determine the correct bin widths in advance for different measured data sets, and thus the results of the this test can vary depending on how the data samples are divided [139].

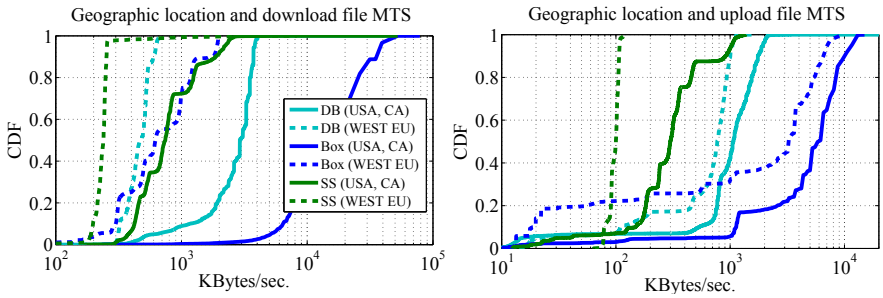



**Figure 5.3:** File MTS distributions of PlanetLab nodes from June 22 to July 15 2012 depending on their geographic location (up/down workload). Clearly, USA and Canada nodes exhibit faster transfers than European nodes.

| Geo. Location | *Metric* | Box | Dropbox | SugarSync |
|---|---|---|---|---|
| USA & CA | $\bar{D}_{MTS}/\bar{U}_{MTS}$ | 3.198 | 2.482 | 2.522 |
| | $\tilde{D}_{MTS}/\tilde{U}_{MTS}$ | 2.550 | 2.722 | 2.500 |
| WEST EU | $\bar{D}_{MTS}/\bar{U}_{MTS}$ | 0.255 | 0.681 | 2.589 |
| | $\tilde{D}_{MTS}/\tilde{U}_{MTS}$ | 0.190 | 0.682 | 2.387 |

**Table 5.3:** Download/Upload transfer speed ratio of Personal Clouds depending on the client's geographic location.

$R^2 = 0.9957$. However, we observe that these fittings differ from the empirical data in the tails of highest transfer speed values. Further, we performed fittings depending on the file size, obtaining closer fittings as the file size grew. The heavier tails found in empirical data but not captured well in the fittings led the KS test to reject the null hypothesis at significance level $\alpha = 0.05$, although in the case of Dropbox, this rejection is borderline (KS-test=0.0269, critical value=0.0240, $p$-value=0.197).

Regarding uploads, we find that Dropbox file MTS *can be modeled by a Weibull distribution* with shape parameter $\mu = 1339.827$ and scale parameter $\sigma = 14.379$ (Fig. 5.2b). In addition to the high $R^2 = 0.9896$, the KS test *accepted the null hypothesis* at significance level $\alpha = 0.05$ (KS-test=0.0351, critical value=0.0367, $p$-value=0.0025).

Due to the high variability, we found that Box uploads do not follow any standard distribution. The implications of these observations are relevant. With this knowledge, researchers can model the transfer speed of Personal Cloud services employing specific statistical distributions.

**Transfers & geographic location**. Next, we analyze transfer speeds depending on the geographic location of vantage points.

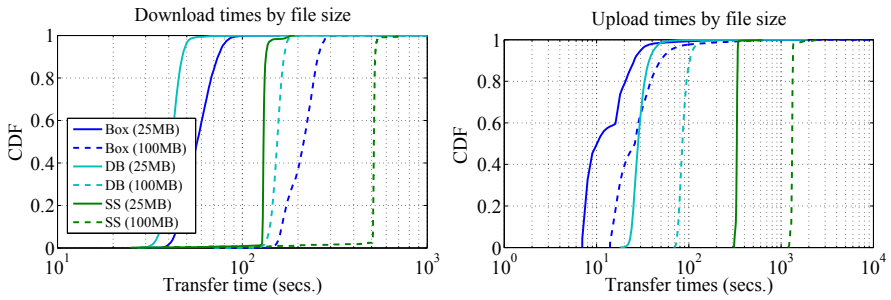



**Figure 5.4:** Transfer times distributions by file size.

In Fig. 5.3, we illustrate the file MTS obtained from executing the up/down workload during 3 weeks in PlanetLab. As can be seen in the figure, *Personal Clouds provide a much greater QoS in North American countries than in European countries*. Intuitively, the location of the datacenter plays a critical role in the performance of the service delivered to users. Observe that this phenomenon is orthogonal to all the examined vendors.

Finally, we quantify the relative download/upload transfer performance delivered by each service as a function of the geographic location of users. To this end, we used a simple metric, what we call the *download/upload ratio* ($D/U$), which is the result of dividing the download and upload transfer speeds of a certain vendor. In Table 5.3, we calculated this ratio over the mean ($\bar{U}, \bar{D}$) and median ($\tilde{U}, \tilde{D}$) values of the file MTS distributions of each provider depending on the geographic location of nodes.

In line with the results obtained in our labs, *European nodes receive a much higher transfer speed when uploading than when downloading* ($D/U < 1$). However, contrary to conventional wisdom, *North American nodes exhibit just the opposite behavior*. This is clearly visible in Dropbox and Box. However, this ratio is constant in SugarSync, irrespective of the geographic location.

### 5.3.2 Variability of Transfer Performance

In this section, we analyze which factors can contribute to the variance in transfer speed observed in Personal Clouds. We study three potential factors, which are *the size of file transfers*; *the load of accounts*; and *time-of-day effects*.

**Variability over file size**. We first investigate the role that file size plays on *transfer times* and *transfer speeds*. Fig. 5.4 and Table 5.4 report the results for both metrics as function of file size, respectively. Unless otherwise stated, results reported in this subsection are based on executing the up/down workload in our university labs during 5 days.

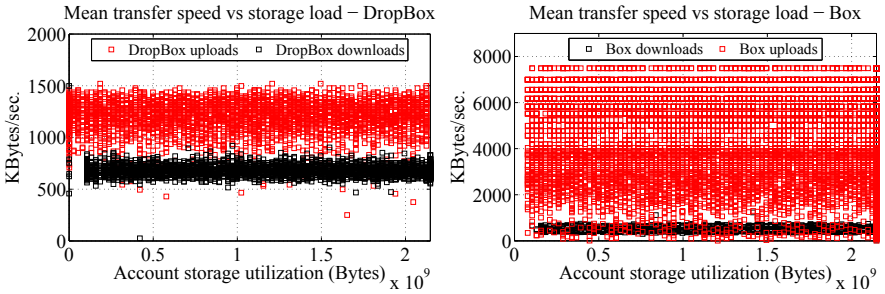



**Figure 5.5:** Relationship between file MTS and the storage load of an account.

Fig. 5.4 plots the transfer time distribution for all the evaluated Personal Clouds. As shown in the figure, for the same provider, all the distributions present a similar shape, which suggests that *the size of file transfers is not a source of variability*. As expected, the only difference is that the distributions for large file sizes are shifted to the right towards longer time values. Significant or abnormal differences were not observed when transferring large files compared to small data files. This observation is applicable to all evaluated Personal Clouds. This leads us to the conclusion that these Personal Clouds *do not perform aggressive bandwidth throttling policies to large files*.

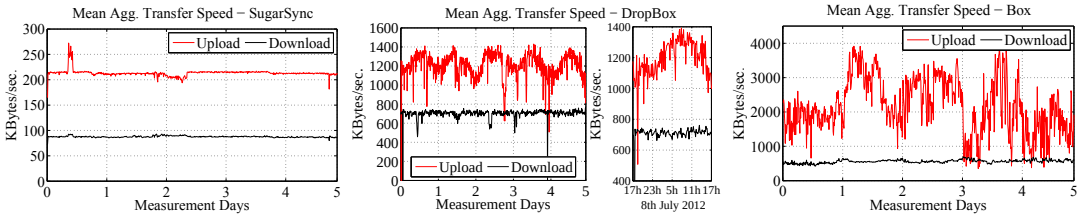
An interesting fact appreciable in Table 5.4 is that *managing larger files report better transfer speeds than in case of small files*. Usually, these improvements are slight or moderate (0.5% to 25% higher MTS); however, uploading 100MB files to Box exhibits a MTS 48% higher than uploading 25MB files to this service. In our view, this phenomena is due to the variability in the incoming bandwidth supplied by Box, and the TCP slow start mechanism, which makes difficult for small file transfers to attain high performance [140].

Further, we found that all the measured Personal Cloud vendors tend to perform a more restrictive bandwidth control to outgoing traffic. This can be easily confirmed by inspecting the obtained standard deviations $\sigma$ of file MTS listed in Table 5.4. Clearly, *the inbound traffic in Dropbox and Box is much more variable than the outbound traffic*. On the contrary, despite its limited capacity, the source of highest transfer variability in SugarSync is in the outbound traffic, which a clear proof of the existing heterogeneity in Personal Clouds.

**Variability over load account**. Next we explore if Personal Clouds perform any control to the transfer speed supplied to users based on the amount of data that users have in their accounts. To reveal any existing correlation, dispersion graphs were utilized to plot the relationship between the MTS and the load of an account at the instant of the storage operation.

| | | Upload File MTS Distribution (KBps) | | | | | | | | Download File MTS Distribution (KBps) | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Size | Provider | Min. | Q1 | Median | Q3 | Max. | Mean ($\mu$) | Std. Dev. ($\sigma$) | CV ($\sigma/\mu$) | Min. | Q1 | Median | Q3 | Max. | Mean ($\mu$) | Std. Dev. ($\sigma$) | CV ($\sigma/\mu$) |
| 25MB | Dropbox | 13.54 | 819.20 | 903.94 | 1008.24 | 1456.36 | 896.28 | 151.56 | 0.1691 | 24.89 | 582.54 | 624.152 | 672.16 | 970.90 | 626.94 | 71.23 | 0.1136 |
| | Box | 14.70 | 1379.71 | 2383.13 | 3276.80 | 3744.91 | 2271,29 | 973.06 | 0.3963 | 163.84 | 397.19 | 459.90 | 534.99 | 794.38 | 463.72 | 87.76 | 0.0837 |
| | SugarSync | 41.87 | 78.25 | 78.96 | 80.17 | 86.23 | 79.26 | 2.82 | 0.0356 | 136.53 | 198.59 | 200.11 | 201.65 | 1048.57 | 201.35 | 37.89 | 0.1882 |
| 50MB | Dropbox | 213.99 | 970.90 | 1092.27 | 1191.56 | 1497.97 | 1069.12 | 152.23 | 0.1424 | 210.56 | 624.15 | 663.66 | 699.05 | 888.62 | 661.55 | 58.02 | 0.0877 |
| | Box | 5.26 | 2496.61 | 4369.07 | 4766.25 | 5825.42 | 3721.12 | 1357.18 | 0.3647 | 14.15 | 623.16 | 647.26 | 672.16 | 887.42 | 646.22 | 44.33 | 0.0686 |
| | SugarSync | 40.27 | 78.72 | 79.44 | 80.41 | 86.95 | 79.59 | 3.08 | 0.0387 | 144.43 | 200.88 | 202.43 | 204.00 | 2496.61 | 216.57 | 149.28 | 0.6893 |
| 100MB | Dropbox | 250.26 | 1127.50 | 1219.27 | 1310.72 | 1519.66 | 1205.69 | 143.05 | 0.1186 | 25.09 | 647.27 | 676.50 | 708.49 | 1497.97 | 680.32 | 50.94 | 0.0749 |
| | Box | 4.71 | 2912.71 | 3883.61 | 6168.09 | 7489.83 | 4350.37 | 1797.32 | 0.3252 | 14.43 | 436.91 | 487.71 | 579.32 | 1233.62 | 507.82 | 89.36 | 0.0539 |
| | SugarSync | 42.23 | 78.96 | 79.62 | 80.66 | 87.31 | 79.64 | 3.74 | 0.0470 | 145.64 | 202.03 | 204.00 | 205.20 | 3744.91 | 223.49 | 219.50 | 0.9822 |

**Table 5.4:** Summary of file MTS distributions by file size.




**(a)** SugarSync does not present important changes in both in/out traffic speed over time.

**(b)** We observe daily patterns in the Dropbox upload transfer speed. Download transfer speed remains stable.

**(c)** Box upload transfers are highly variable and probably affected by daily patterns.

**Figure 5.6:** Evolution of Personal Clouds upload/download transfer speed during 5 days. We plotted in a time-series fashion the mean aggregated bandwidth of all nodes (600 secs. time-slots) executing the service variability workload in our university laboratories (3rd−8th July 2012).

As shown in Fig. 5.5, *we were unable to find any correlation between the file MTS and the load of an account* in any of the measured Personal Clouds. This suggests that the transfer speed delivered to users remains the same irrespective of the current amount of data stored in an account. This conclusion is important to characterize which types of control mechanisms are actually applied to these storage services.

**Variability over time**. We now analyze how the transfer speed varies over time. To better capture these variations, we used the data from the *service variability workload*, which was aimed to maintain a constant transfer flow and was executed at our university labs. The results are shown in Fig. 5.6 where the *mean aggregated bandwidth* of all nodes as a whole is plotted in time intervals of 600 seconds. As expected, we found that the transfer speed of these services behave differently depending on the provider. To wit, while SugarSync exhibits a *stable service for both uploads and downloads*, at the price of a modest transfer capacity (Fig. 5.6a), the upload transfer speed varies significantly over time for Dropbox and Box.

Appreciably, *Dropbox exhibits appreciable daily upload speed patterns* (Fig. 5.6b). Data represented in Fig. 5.6 was gathered between July 3, 6:00p.m. and July 8, 3:00p.m. Clearly, during

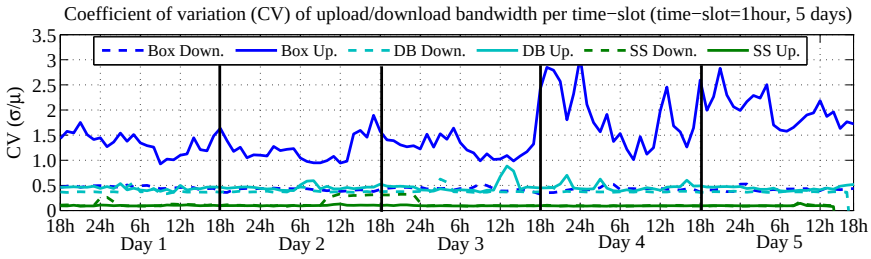



**Figure 5.7:** Evolution of transfer speed variability over time (service variability workload, university labs).

night hours (1 a.m.−10 a.m.), transfer speed was between 15% to 35% higher than during diurnal hours. This phenomenon has been also detected in the experiments performed in PlanetLab, thereby discarding any artificial usage pattern induced by our university network. Moreover, considering that Dropbox uses Amazon S3 as storage back-end, our results are consistent with other recent works [141] that observed similar patterns in other Amazon services.

Further, we found that Box upload service may be subjected to high variability over time. Indeed, we observed *differences in upload transfer speed by a factor of* 5 *along the same day*. This observation is consistent with the analysis of the file MTS distribution where significant heterogeneity was present. More interestingly, Box uploads appear to be also affected by *daily patterns*. Concretely, the periods of highest upload speed occurred during the nights, whereas the lowest upload speeds were observed during the afternoons (3 p.m. −10 p.m.). Due to the huge variability of this service, a long-term measurement is needed to provide a solid proof of this phenomenon, though.

With respect to downloads, we observed no important speed changes over time in any system. This suggests that *downloads are more reliable and predictable, probably due to a more intense control of this type of traffic by the datacenter*.

To specifically compare the variability among services over time, we made use of the *Coefficient of Variation* (CV), which is a dimensionless and normalized measure of dispersion of a probability distribution, specifically designed to compare data sets with different scales or different means. The CV is defined as:

$$CV = \frac{1}{\bar{x}} \sqrt{\frac{1}{N-1} \sum_{i=1}^{N} (x_i - \bar{x})^2},$$

where $N$ is the number of measurements; $x_1$, .., $x_N$ are the measured results; and $\bar{x}$ is the mean of those measurements.

| *Downloads* | Dropbox | Box | SugarSync |
|---|---|---|---|
| 25MB | 0.047%($\frac{5}{10,503}$) | 0.572%($\frac{68}{11,878}$) | 0.115%($\frac{2}{1,740}$) |
| 50MB | 0.082%($\frac{8}{9,745}$) | 0.698%($\frac{80}{11,445}$) | 0.057%($\frac{1}{1,727}$) |
| 100MB | 0.044%($\frac{4}{9,026}$) | 0.716%($\frac{80}{11,169}$) | 0.059%($\frac{1}{1,691}$) |
| 150MB | 0.042%($\frac{3}{7,136}$) | – | 0.076%($\frac{1}{1,359}$) |
| *Uploads* | Dropbox | Box | SugarSync |
| 25MB | 0.384%($\frac{41}{10,689}$) | 0.566%($\frac{227}{40,043}$) | 0.889%($\frac{8}{899}$) |
| 50MB | 0.450%($\frac{48}{10,663}$) | 1.019%($\frac{405}{39,719}$) | 1.079%($\frac{10}{926}$) |
| 100MB | 0.502%($\frac{54}{10,740}$) | 2.097%($\frac{836}{39,875}$) | 1.988%($\frac{18}{905}$) |
| 150MB | 1.459%($\frac{58}{3,974}$) | – | 3.712%($\frac{33}{889}$) |

**Table 5.5:** Server-side failures of API operations ($3_{rd} - 8_{th}$ July 2012).

Fig. 5.7 depicts the CV in 1-hour time slots of the *aggregated bandwidth* provided by each Personal Cloud vendor. Clearly, it can be observed important differences across the vendors. Concretely, SugarSync experiences low variability with a CV of only 10%. Dropbox with a CV around 50%, however, exhibits a much higher variability than SugarSync, including isolated spikes in the upload bandwidth that reach a CV of 90%. In this sense, the Box download bandwidth capacity exhibits a similar trend. Finally, the highest observed variability was for Box uploads. In the first 3 days of the experiment, Box exhibited a mean CV of 125% approx. However, in the last part of the experiment some spikes reached a CV of 300%, suggesting that it is really *hard to predict* the behavior of this service.

### 5.3.3 Service Failures and Breakdowns

Another important aspect of any Cloud storage service is *at what rate* users experience failures, and *whether the pattern of failures can be characterized by a simple failure process like a Poisson process*, which allows researchers to develop tractable analytical models for Personal Clouds.

For this analysis, any event *server-side* notification signaling that a storage operation did not finish *successfully* was counted as a *failure*, thereby excluding any failure, where abnormal or degraded service was observed[1]. Table 5.5 summarizes the server-side failures observed during a 5-day measurement based on the variability workload run at our labs.

**Failure rates**. Table 5.5 illustrates a clear trend: *in general, uploads are less reliable than downloads*. This phenomenon is present in all the Personal Clouds measured and becomes *more important for larger files*. As can be observed, downloads are up to 20X more reliable than uploads (Dropbox, SugarSync), which is an important characteristic of the service delivered to users. In this

[1]We filtered the logged error messages depending on their causes as detailed in the API specifications. We considered as errors most of the responses with $5XX$ HTTP status codes as well as other specific errors related with timed out or closed connections in the server side.

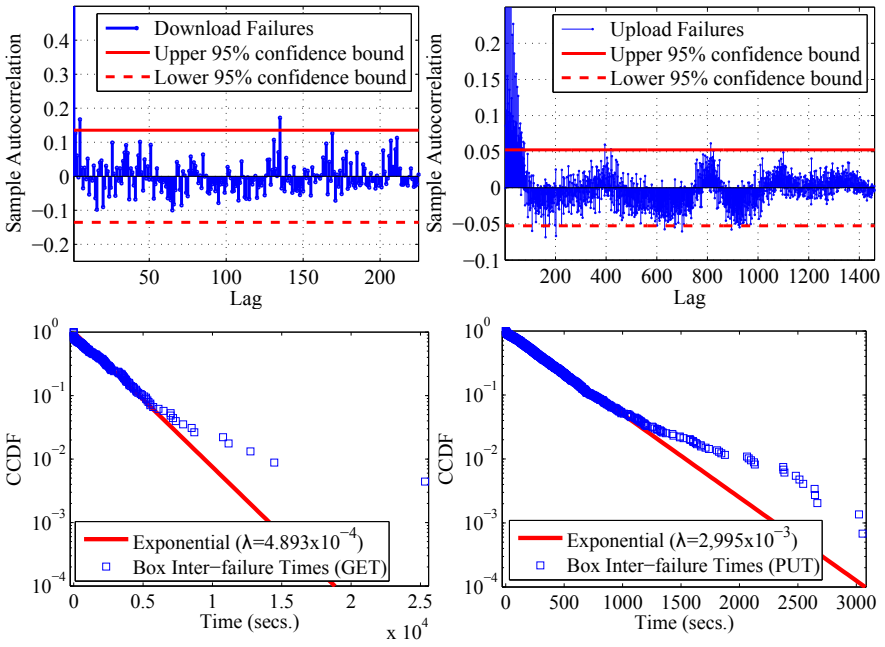



**Figure 5.8:** Failure interarrival times autocorrelation (upper graphics) and exponential fitting of failure interarrival times (lower graphics) for Box.

sense, although failures among uploads and downloads are not so high, Box seems to provide the least reliable service. Anyway, failure rates are generally below 1%, which suggests that these *free storage services are reliable*.

**Poissonity of failures**. Now we study whether service failures appear Poisson or not, because Poisson failures allow for easy mathematical tractability. Poisson failures are characterized by interarrival times which are independent of one another and are distributed exponentially [104], and for which the failure rate is constant. In this case, we focused only on Box, since it was the only service for which enough observations were available for the statistical analysis to be significant.

To verify whether failures are independent, we calculated the autocorrelation function (ACF) for consecutive failures in the time series and depicted it in Fig. 5.8[1]. When the failures are completely uncorrelated, the sample ACF is approximately normally distributed with mean 0 and variance $1/N$, where $N$ is the number of samples. The 95% confidence limits for ACF can then be approximated to $0 \pm \frac{2}{\sqrt{N}}$. As shown in Fig. 5.8, in the case of download failures, autocorrelation coefficients for most lags lie within 95% confidence interval, which

[1]Due to lack of space, we refer the reader to [104] for a technical description in depth of this methodology to assess Poissonity.

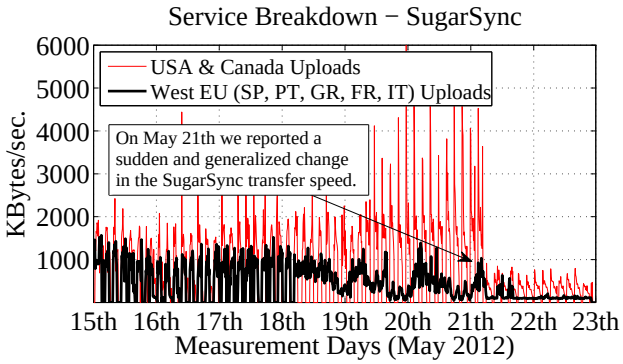



**Figure 5.9:** We observe a radical change in the upload transfer speed of SugarSync from May 21 onwards. After May 21 all the tests performed against SugarSync reported very low transfer speeds. This reflects a change in the QoS provisioned to the REST APIs of free accounts.

demonstrates that failure interarrival times are independent of one another. However, uploads failures are not independent, since the first lags exhibit high ACF values, which indicates short-term correlation, with alternating positive and negative ACF trends.

To conclude Poissonity for failures, failure interarrival times must be exponentially distributed, for we report the *coefficient of determination*, $R^2$, after performing linear regression on the distribution $log_{10}(1 - \{Pr\{X < x\})$, where $Pr\{X < x\}$ is the empirical failure interarrival time distribution obtained for Box. In the case of downloads $R^2 = 0.9788$ whereas for uploads $R^2 = 0.9735$. This means that failure interarrival times *approximately follow an exponential distribution*, which is evidenced in Fig. 5.8, where most of the samples match the exponential fitting, with the exception of those at the end of the tail. Hence, Box download failures can be *safely considered as being Poisson*. Although upload interarrival times can be well fitted by the exponential distribution, they are not independent and further analysis is needed to their characterization.

**Service breakdowns**. Apart from the "hard" failures, there are other types of "soft" failures related with the deterioration of the QoS. And indeed, we captured a strong evidence of this in late May 2012 (Fig. 5.9). In Fig. 5.9 we present a time-series plot of the aggregated upload MTS of PlanetLab nodes against SugarSync. This information is divided for those nodes located in West Europe and USA & Canada[1].

Clearly, the behavior of the upload speed of SugarSync changed radically from May 21 onwards (Fig. 5.9). Before that date, SugarSync provided high transfer upload speed, compa-

[1] Spikes present in Fig. 5.9 are due to the PlanetLab quota system, which limits the amount of data that users can transfer daily.

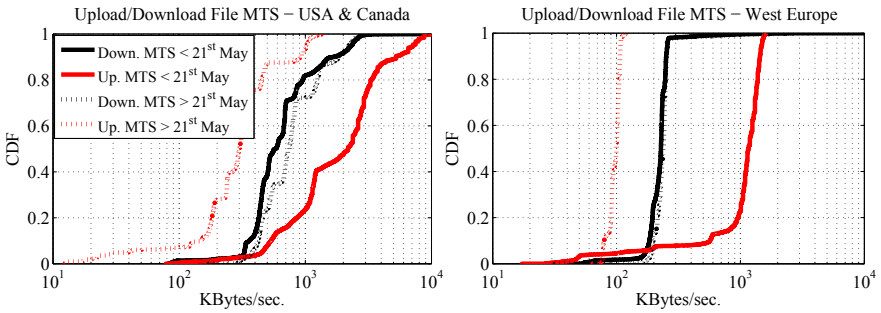



**Figure 5.10:** PlantLab experiments against SugarSync before and after the service breakdown reported in May 21. We observe an important service degradation for uploads, whereas the download service remains unaltered.

rable to current performance of Box. However, in May 21 SugarSync bandwidth provisioning policies changed dramatically; the upload MTS was reduced from 1,200KBps to 80KBps in Western Europe —a similar trend can be observed in USA and Canada. Note that we accessed to the SugarSync service from a variety of nodes and accounts, discarding thus the possibility of IP filtering and account banning.

In this sense, Fig. 5.10 shows the upload/download MTS distributions for measurements performed before and after the service breakdown —executing the same workload (up/down workload) over the same nodes. Clearly, the change in the transfer speed of SugarSync was focused on uploads, that previously exhibited a good performance. On the other hand, we see that the download service was almost unaltered after May 21. These observations apply to both geographic regions. This means that Personal Clouds may change their *freemium* QoS unexpectedly, due to internal policy changes.

At this point, we have characterized the transfer QoS of three major Personal Clouds through their REST API service. In what follows, we describe how malicious parties can exploit the available REST API service over free accounts to abuse Personal Cloud and consume storage resources freely.

## 5.4 The Storage Leeching Problem

As we mentioned before, major Personal Clouds such as Dropbox, Box and SugarSync provide open REST APIs for developers to create clever applications over their service, in order to make their offering more attractive. From a functional viewpoint, these APIs enable an application to upload/download files to/from user accounts, blurring the lines between a Personal

Cloud service and a pure IaaS storage provider as Amazon S3. However, the unintended consequence is that it is very easy for a user to aggregate multiple free accounts from the same or from different Personal Clouds to obtain a free storage space comparable to paid accounts.

The roots of the problem lie deeply in the lack of *accountable* identities. Personal Clouds do not provide mechanisms to enforce the rule that *one real person* gets *one virtual identity* in their online services, what is known as the *Sybil attack* [142]. As an illustrative example, Box requires only the first name, last name, email and password for a user to set up an account of 5 GB of free storage. This *quick* registration process makes it possible for one real person to get multiple accounts and here is when the open nature of these REST APIs facilitate the abuse of the storage service. Box REST API allows a developer to enable up to four other users per application yet in development status, so nothing prevents a malicious developer from aggregating his 25 GB of free storage as a single unit. In the case of Box, this new form of abuse may have economic consequences. At the time of this writing, a Box account of 25 GB costs $9.99 per month.

The extent of the abuse can be even worse if the abuser aggregates accounts from multiple providers. In such a case, the abuser can take benefit of storage diversity to obtain even a better service than what can be delivered from a single provider. By an intelligent allocation of file chunks to different providers, a malicious user can improve download times, upload times or both, and obtain a unified account with better QoS than a paid account totally free of charge.

We use the term "*storage leeching*" to refer to this generic form of abuse because the abusers or leechers seek to benefit from free storage while trying to leave unnoticed. This form of abuse is hard to prevent because it is under the umbrella of the *freemium* business model adopted by Personal Cloud companies. That is, storage providers offer free and paid premium accounts that are very similar in all aspects except for the amount of storage space offered. This, in conjunction with the business strategy to cultivate a developer ecosystem through the release of open APIs, makes it really hard for these companies to prevent storage leeching.

To illustrate the potential consequences of storage leeching, let us describe a real example. During the development of this piece of research, we executed several experiments against the REST API service of three major vendors: Box, Dropbox and SugarSync. We consumed around 45.26TB of download traffic, 25.75TB of upload traffic and 450GB of storage. Excluding the number of transactions, in terms of Amazon S3 pricing[1], *our experiments represent a cost of*

[1] `http://aws.amazon.com/en/s3/pricing/`

$5,431.2 *in download traffic, plus a monthly storage cost of* $42.75. This evidences that it is very easy to exploit these services.

We believe that the storage leeching problem is a substrate over which many abusive applications might exploit Personal Clouds. For instance, a single user may aggregate free accounts as a storage backend to support an *illegal webpage* which exhibits prohibited contents or even, as a part of a *peer-assisted storage* system [25]. Even worse, a malicious user may share with others the access tokens of a certain account, which enables any other user to access the stored data. The potential damage of this form of exploitation may be important, since it leverages the creation of applications such as *file-sharing*, where *users not registered in any Personal Cloud can freely consume resources and illicitly benefit from these services*.

The next section describes the design of a proof-of-concept file-sharing application that abuses Personal Clouds.

## 5.5 Boxleech: An Abusive File-sharing Application

Boxleech is a proof-of-concept file-sharing application able to disseminate illegal or copyrighted content by abusing Personal Clouds. Essentially, it aggregates free accounts from multiple Personal Clouds into a single storage unit that can be freely accessed by users interested in a certain content. In particular, Boxleech aggregates free accounts from Dropbox, Box and SugarSync, three major storage vendors, which shows the potential impact of storage leeching and the simplicity to exploit public APIs to abuse Personal Clouds.

In Fig. 5.11, we provide a general overview of the functioning of Boxleech. We observe two Personal Clouds where a malicious user has registered a developer application and few free accounts. Besides, he enables the REST API access to these accounts obtaining the required tokens. Using the Boxleech client, he uploads chunks corresponding to an illicit content he wants to share. Finally, he generates and distributes the metadata file which contains the information to enable any other Boxleech user to download the content.

The design of Boxleech can be divided into three main blocks: *data management*, *metadata* and *chunk assignment*.

**Data management**. First, similar to Dropbox and the likes, which internally do not use the concept of files, Boxleech splits every file into chunks of up to 100MB in size. There were three good reasons for this: i) To surpass the file size limitations commonly imposed in the REST API access to free accounts, ii) To exploit storage diversity by allocating chunks of the

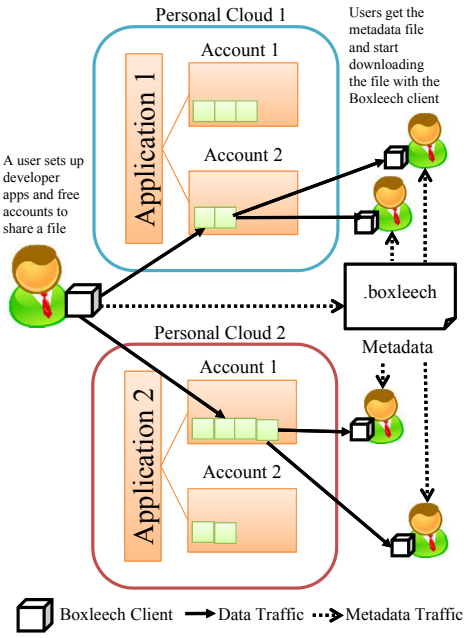



**Figure 5.11:** Users abusing Personal Clouds by sharing illicit contents with `Boxleech`. Once users get the metadata file that contains the account access credentials for each chunk, they are able to download the shared content.

same file to different Personal Clouds and, iii) To make it impossible for a single provider to store an entire copy of an illicit content. Currently, `Boxleech` applies a simple fragmentation algorithm to create equally-sized file chunks. However, more elaborated mechanisms such as Erasure Codes [25] may be introduced to increase data availability via data redundancy.

Locally, `Boxleech` maintains an index which relates every chunk with the file it belongs to, as well as the information about the cloud account where it has been stored. To manage data chunks from these Personal Clouds, the implementation of `Boxleech` includes the client API of all of them. Clearly, supporting a new provider will require introducing the corresponding API implementation in the application.

**Metadata**. The objective of `Boxleech` metadata files (`.boxleech`) is to map a set of chunks corresponding to the same content to their location in diverse Personal Cloud accounts. A metadata file is formed by a set of rows, each one containing the following information for a data chunk: *[chunk_id], [order], [provider], [access_credentials]*. The first two fields describe the identifier of a chunk (e.g. hash value) and the chunk position, which is needed to reconstruct the file after downloading it. The *access_credentials* field includes the necessary access information to download that chunk from the appropriate provider. In the case of Dropbox and

Box, there is only need to include the access token for the account where the chunk is stored. However, SugarSync requires to include the secret/application keys as well as the account login/password information to renew the token after its expiration.

`Boxleech` is capable of generating a `.boxleech` file for a content that a user has uploaded, as well as to interpret these files to download contents shared by other users. Similarly to a *.torrent* file [143], there are several ways of indexing and distributing these metadata files, such as a Web server (tracker) or a Distributed Hash Table (DHT). Going further, we advocate for building a metadata index also exploiting Personal Cloud accounts. To illustrate this, let us assume that each metadata file is named with the *hash* of the original content name (e.g. film title). Making use of consistent hashing [144], we can partition the hash identifier space among a set of storage accounts, which adopt the role of traditional hash buckets. Hence, `Boxleech` clients are able to deterministically search for a hash value in the appropriate account. This leverages an integral file-sharing service entirely supported by exploited resources from Personal Clouds[1].

**Chunk Assignment**. The allocation of chunks when exploiting various Personal Clouds plays a critical role on the speed of transfers.

As we already observed in Section 5.1, Fig. 5.1 evidences an interesting fact: *Personal Clouds provide very disparate transfer speeds*. For instance, in Fig. 5.1b we observe that Box and Dropbox provide a upload MTS several times faster than SugarSync —the same observation holds for downloads. Moreover, the heterogeneity of these services also depends on the *traffic type* (in/out). This can be appreciated comparing Fig. 5.1a and 5.1b: Dropbox exhibits the best download MTS values whereas Box clearly provides the fastest uploads. Hence, we conclude that Personal Clouds are heterogeneous in terms of transfer performance.

`Boxleech` exploits this feature to show that leechers can obtain even faster transfers by intelligently allocating the file chunks to various providers[2]. This allocation depends on the *chunk assignment policy*. In Section 5.6, we propose and evaluate `Boxleech` using several chunk assignments.

**Initialization**. To share content with `Boxleech`, all we needed to do was to sign up for some free accounts and then register as a developer in each storage service. Once registered, we instantiated a fake application with the intention to receive an application and secret key pair.

[1] In [54], Mulazzani et. al. point out that Dropbox is being used to store and share *.torrent* files, as well as to distribute copyrighted material.

[2] We confirmed through experimentation that multiple parallel download transfers from a single data object do not decrease transfer performance. This provides an appropriate substrate to build an efficient file-sharing system.

Using these keys, we validated our credentials and obtained the authorizing tokens that must be passed in every API call. All this process was done with little human interaction since the core idea of the freemium model is to recruit as much users as possible through a simple sign-up process.

## 5.6 Experimental Evaluation

Next, we evaluate `Boxleech` and we compare its performance with desktop clients delivered by Dropbox and SugarSync to illustrate the potential benefits of storage leeching.

### 5.6.1 Setup & Methodology

**Scenario**. We executed our experiments in our university laboratories. We used 12 machines in order to run the different software configurations employed in our tests. We employed Intel Core i5 machines equipped with 4GB of DDR3 memory. The operating system was Windows 7 (Dropbox, SugarSync clients) and Linux Debian (`Boxleech`). Machines were connected to the same switch via a Fast Ethernet link.

**Software**. *Personal Cloud Desktop Clients*. Dropbox and SugarSync provide free and closed desktop clients to maintain in sync files from multiple devices and the Cloud[1]. In our experiments, both clients were explicitly configured to provide the maximum transfer capacity. Moreover, in the case of Dropbox, we deactivated the LAN Sync option which permits the synchronization of multiple devices in the same network.

*`Boxleech`*. Our file-sharing application employed the standard API implementations to access storage accounts. Specifically, we used two configurations of free accounts in our tests: i) 3 free accounts, one for each Personal Cloud analyzed in this article and ii) 5 Box accounts, to test large storage operations to the same provider. `Boxleech` made use of parallel transfers when transferring chunks in and out from each account. In case of a failed storage operation on a chunk, it performed retries until making the operation succeed. This could increase transfer times in case of multiple failures.

**Workload**. For both desktop clients and `Boxleech` we executed an alternate upload and download workload. Basically, it consisted on generating a new file, uploading it to the account and downloading it before its deletion.

[1] Sugarsync client version: 1.9.71.94365.20120712. Dropbox client version 1.4.11. Box is excluded from this evaluation since it currently does not provide a free desktop client.

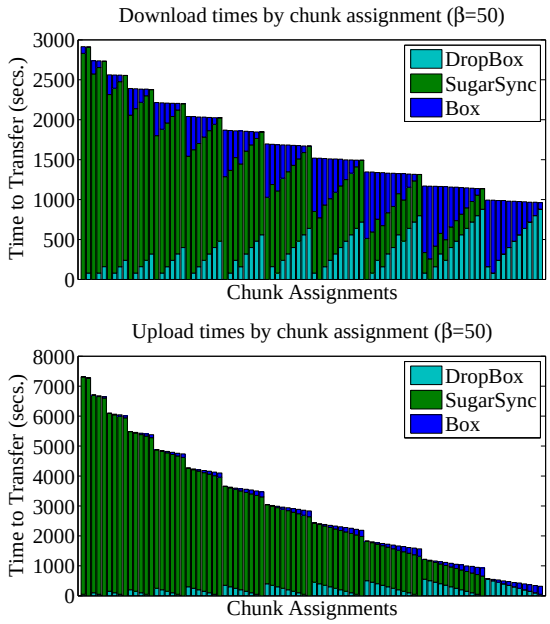



**Figure 5.12:** Impact of chunk assignment on transfer times (chunks are assumed to be sequentially transferred). Clearly, different assignments report disparate transfer performance, which is essential to effectively exploit the service.

Specifically, in the case of desktop clients, the workload is executed by pairs of computers —each one dedicated either to upload or download files. First, the upload script created a synthetic file which is stored in the desktop client *watch directory* of the computer responsible for uploads. This script was continuously checking in the server-side whether the client finished the upload or not. In parallel, the download script was waiting in the second computer until the upload had finished. Then, it started measuring the download time until the remote file was available in that computer. When the download concluded, the download script deleted the file, which served as a notification to the upload script to repeat the process again.

Storage operations were performed over synthetic random and compressed files. This was necessary to prevent desktop clients from applying caching, deduplication and compression mechanisms over these files. Our results are based in approx. 100 storage operations for each software and configuration.

**Chunk Assignments**. To explore the impact of chunk assignments in depth, we performed a battery of Monte Carlo simulations over the *empirical data* collected in our measurement (see Fig. 5.1). Fig. 5.12 plots the impact of different chunks allocations. The abscissa axis shows the upload/download transfer time measured in seconds for transferring a file of $F = 600$

MB. For each possible allocation of $n$ chunks among Box, Dropbox and SugarSync, there is one corresponding bar in Fig. 5.12. Note that depending on the chunk size $\beta$, the number of chunks $n$ will vary ($n = F/\beta$). The colored segments in the bars represent the time incurred to sequentially transfer the chunks assigned to a given Personal Cloud.

As expected, assigning more chunks to Dropbox reduces the download time, since this vendor exhibited the fastest download capacity in our experiment. This always holds, irrespective of the chunk size. In any case, allocating the majority of chunks among Box and Dropbox ensures to the abusive application good download performance while improving load balancing among both providers.

On the other hand, due to its poor performance, allocating more chunks to SugarSync yields higher upload times. The impact on transfer times of SugarSync uploads is much higher than in the case of downloads. In this sense, we observe an important improvement as more chunks are assigned to Box, which exhibits the fastest upload service in our experiment.

As a result of these observations, we implemented and tested three simple allocation policies to assess the potential benefits of exploiting storage diversity. These policies are:

- *Round Robin (RR)*: This strategy is extremely simple to implement and has been adopted in many real systems. This placement allocates the same amount of chunks to each user account in order to ensure fairness and reliability. This policy serves as a performance baseline and it does not make use of any source of information.

- *Upload/Download Proportional (UP, DP)*: Based in our analysis, we propose two new placements to reduce upload and download transfer times. Both placements assign a number of chunks in proportion to the transfer capacity of each Personal Cloud. The transfer capacities has been extracted from our measurement study, and therefore, UP and DP are informed assignment policies.

Next, we evaluate the differences in performance between both types of placement policies (informed and non-informed).

### 5.6.2 Experimental Results

**Single provider**. One simple form of storage leeching is to aggregate free accounts from the same provider. In the next experiment, we want to verify if the aggregation of accounts from the same provider entails some performance degradation. For this reason, we aggregated 5 Box accounts and uploaded large amounts of data. The results are shown in Fig. 5.13.

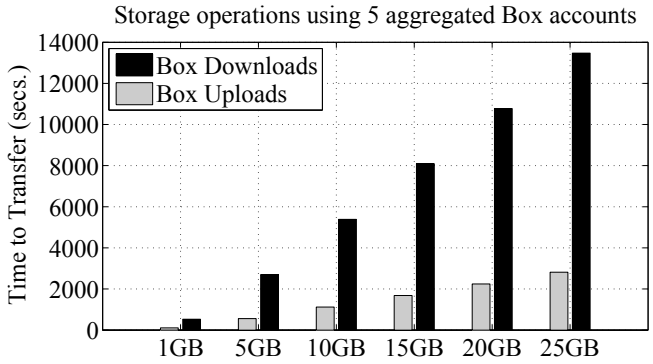



**Figure 5.13:** Transfer performance of `Boxleech` aggregating 5 Box accounts. We observe that the upload capacity of Box is really high and can be effectively exploited to store and share large amounts of data.

Fig. 5.13 shows the storage and retrieval performance of `Boxleech` aggregating 5 Box accounts for different amounts of data. Although the linear behavior of transfer times was expected, it is surprising to see the upload speed `Boxleech` with this configuration. Actually, the average transfer speed of chunks was $\approx$ 11.5MBps[1], which is a high-quality free service. One of the most important conclusions of this experiment is that *aggregating an arbitrary number of free accounts is extremely easy*. Furthermore, aggregating several accounts of the same Personal Cloud does not seem to degrade the service performance, meaning than exploiting a single provider is a feasible leeching strategy.

As an important remark, note that *a single user is able to consume around* 25*GB of storage and upload traffic, as well as* 5*GB of download traffic in one hour*. Thus, one can easily imagine the economic expense in terms of consumed resources that a large user population may cause to a provider.

**Multiple providers**. Next, we focus on the transfer speed of `Boxleech` compared with Dropbox and SugarSync desktop clients. For this experiment we used 600MB synthetic files, which emulates a scenario of users sharing music albums.

In Fig. 5.14 we infer that `Boxleech` obtains better transfer speed than Dropbox and SugarSync clients in many configurations. For downloads, `Boxleech` using the Download Proportional policy (DP) provides a transfer speed nearly *2 times higher than the obtained by the SugarSync client*, which is the client exhibiting fastest downloads. To wit, the DP policy assigns more chunks to Box and Dropbox services, which present the highest download speeds

[1]Note that such a speed cannot be continuously maintained since we start a new TCP connection for each chunk to be transmitted.

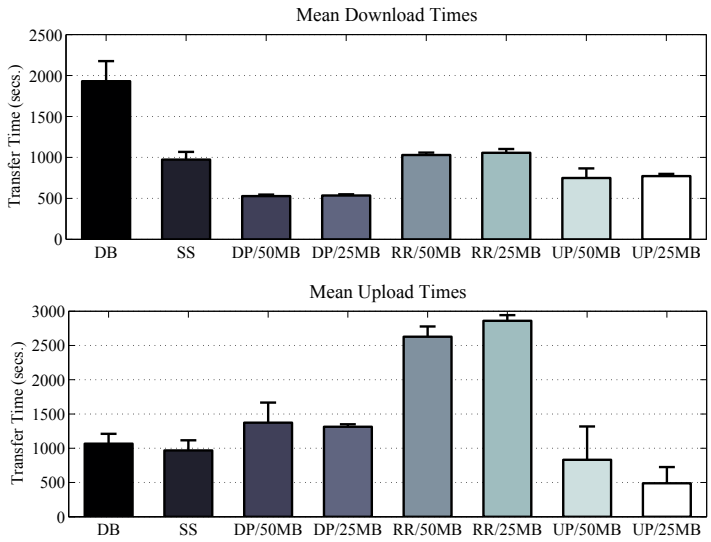



**Figure 5.14:** Mean transfer times and standard deviation (error bar) of `Boxleech` under distinct configurations and Dropbox (DB) and SugarSync (SS) clients.

in our measurement. This makes `Boxleech` downloads considerably faster. Note that even using the simple Round Robin (RR) policy, `Boxleech` reports a download speed similar to the SugarSync client.

For uploads, we see that `Boxleech` is able to obtain comparable or even better transfer speed than Personal Cloud desktop clients. That is, `Boxleech` using the Upload Proportional (UP) policy with chunks of 25MB presents upload times over 55% shorter than its counterparts. In this sense, the RR policy reports the worst performance due to the amount of chunks uploaded to the SugarSync service, which is really limited. However, this is the only policy that provides storage balance among accounts. Thus, there is a *trade-off between storage balance and transfer speed* when exploiting accounts from multiple providers.

Appreciably, for both uploads and downloads, we see that our informed chunk assignment policies provide a higher transfer speed than the RR policy. Hence, *the information of our measurement helps to better exploit storage diversity*.

Fig. 5.15 helps to understand the reported file transfer times in Fig. 5.14. Fig. 5.15 shows the chunk transfer time distributions for each Personal Cloud used by `Boxleech`. We found that a *small fraction of chunks* exhibit really large transfer times. This phenomenon is specially pronounced in Box uploads. This impacts on file transfer times of our application since all chunks should be transferred before finishing. This effect might be induced by the management of

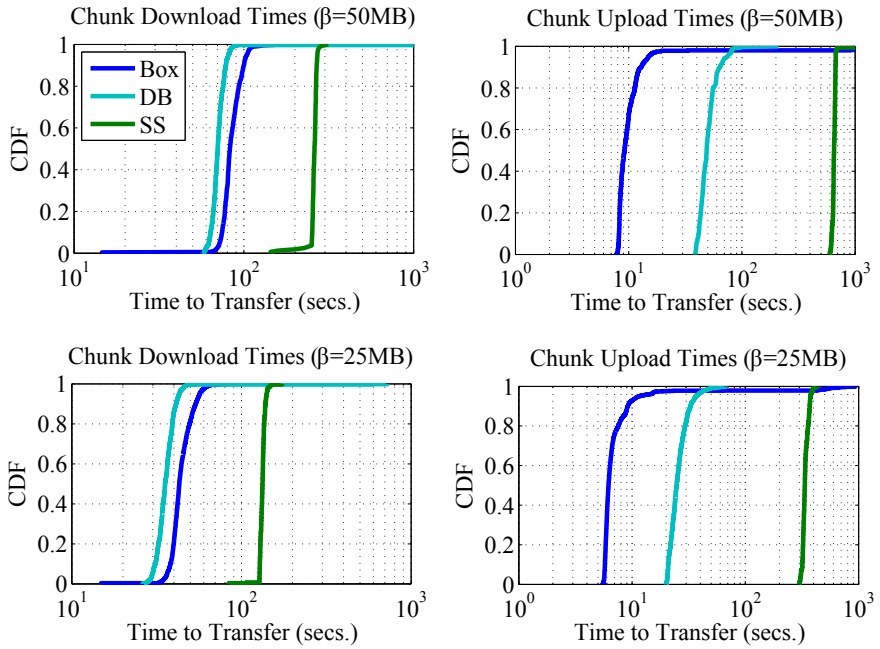



**Figure 5.15:** `Boxleech` chunk transfer times distributions for both uploads and downloads, as well as for different chunk sizes $\beta$. Probably, due to the management of parallel transfers of `Boxleech`, a small fraction of chunks present really large transfer times.

parallel transfers in `Boxleech`, that should be carefully addressed for those applications which want to optimize transfers.

Moreover, Fig. 5.14 shows that the chunk size ($\beta$) does not have important implications to the transfer performance, at least in case of moderate file sizes.

Another interesting observation comes from the analysis of Dropbox and SugarSync clients (Fig. 5.14). In our experiments, Dropbox exhibits a much greater REST API transfer speed than SugarSync. However, we see that the transfer performance of both clients is quite similar in case of uploads. Furthermore, we observe that SugarSync provides a download speed much better than the Dropbox client. This suggests that: i) these clients may implement bandwidth control mechanisms in order to *restrict the resource consumption coming from free accounts*, and ii) the performance of *REST APIs is not necessarily related with the performance of the desktop client*.

## 5.7 Discussion and Conclusions

In this section, we discuss (i) the most important insights from our measurement of Personal Cloud REST APIs and (ii) several countermeasures to ameliorate the impact of storage leech-

ing. First, here we summarize the most relevant technical observations obtained from this measurement:

- *Characterization of transfers*. In some cases, we observed that transfer time distributions can be characterized by known statistical distributions like the *log-logistic* and the *logistic* for downloads in Dropbox and Box, respectively. We also found that upload transfer times are Weibull distributed in Dropbox. In SugarSync, we observed a constant and very limited transfer performance. This characterization opens the door to create Personal Cloud modeling and simulation environments.

- *High service variability*. The variability of Personal Cloud services is significant and induced by many factors. To wit, we discovered that uploading to Dropbox is substantially faster at nights ($15\% - 35\%$), which proves the presence of daily usage patterns. We also found that the magnitude of the variation is not constant over time. An example of this is Box. While Box uploads exhibited a mean variability of 125% at the beginning of our experiment, the CoV reached 300% at the end. Further, we found that *uploads are more variable than downloads*.

- *Reliability and Poissonity of failures*. In general, we found that Personal Clouds are *reliable*, exhibiting failure rates below 1%. We also found that for Box, failure interarrival times approximately follow an exponential distribution. Moreover, Box download failures can be modeled as a Poisson process, which is analytically simple.

- *QoS changes and data lock-in*. We found that SugarSync changed its *freemium* QoS unexpectedly. Concretely, the mean upload speed delivered by SugarSync suddenly dropped from $1,200$ KBps to 80 KBps in EU. This emphasizes the relevance of the *data lock-in* problem, where a customer gets trapped in a provider whose service is unsatisfactory but cannot move to a new one because of the amount of data stored in it.

In summary, our measurement provides a novel characterization of Personal Clouds REST API services that expands the knowledge base on these systems. Our insights and the publicly available dataset will enable researchers and practitioners to better understand, model and predict the behavior of Personal Clouds.

Second, in our view we just scratched the surface of the set of exploitation possibilities that the *storage leeching problem* permits. In addition to benefit abusive applications like `Boxleech`,

storage leeching is a vector to materialize many other threats, such as denial-of-service attacks against Personal Clouds [13] or fraudulent resource consumption [94, 95].

In this sense, we highlight the danger that a quick account registration process (e.g. no *captchas*) represents to Personal Clouds. As a lesson learned from working with Personal Cloud APIs, we created over 140 free accounts and 35 developer applications of various vendors in few hours. This represents a virtual storage capacity of around 450GB. Moreover, it is not hard to imagine expert hackers creating scripts to facilitate, even more, the initial registration process for non-expert users. In our view, leveraging storage leeching to the masses would have important economic implications to Personal Clouds.

Although introducing countermeasures to provide a secure API service is strategic decision from a vendor's viewpoint, we propose the following ones based on our experience:

- *Enforce accountable user identities*. The main requirement to access free storage and register an application as developer is an email account. Thus, if email accounts are easy to create, any user can rapidly gather an arbitrary amount of free storage. We suggest to introduce filters in the creation of Personal Cloud free accounts and/or registering applications to enforce that one user obtains one account (phone number, human intervention). Currently, systems like Facebook introduce very restrictive procedures to their developer environments.

- *Expiration time for developer applications*. To discourage malicious users to exploit open APIs as a durable storage substrate, we believe that introducing expiration mechanisms to both developer applications and the related free accounts could be an effective countermeasure. By doing this, Personal Clouds would force abusers storing their data in the system to periodically migrate it.

- *Identify anomalous workloads*. According to our conclusions in Chapter 4, Personal Clouds could benefit from research efforts focused on identifying fraudulent resource consumption to detect abuse in storage accounts related to developer applications. This suggestion comes from our empirical experience: we actively performed tests against free accounts for 2 months. In that time, we have not detected any change in the service provided by Personal Clouds, even though the executed workload could be easily detected as an anomaly.

Finally, it is surprising that many vendors do not implement this kind of security measures in their API service, even though it is technically possible. Perhaps, Personal Clouds assume

the risk of a possible abuse of their service motivated by luring as many users and developers as possible. However, observing the behavior of SugarSync, where the REST API transfer performance is substantially worse than that exhibited by the desktop client, it seems probable that other providers will restrict the *freemium* API service in the future.

**Conclusions**. To lure customers and developers, Personal Clouds provide open REST APIs to create new applications that make their service even more attractive. In this Chapter, we first provided a characterization of the transfer QoS of these services, analyzing relevant aspects of their performance such as transfer speed, variability and failures. This information may be of great interest no only to end-users, but also to developers integrating their applications in Personal Clouds, or to researches willing to model the behavior of these services.

Moreover, we observed that the unintended consequence of combining REST APIs over free accounts is that it is very easy for a user to abuse the service by aggregating free accounts, from one or several providers, to obtain a high-quality storage service. We demonstrated this observation with a practical example of an abusive file-sharing application.

Finally, we provided technical discussion on both the implications of our measurement and some potential countermeasures to mitigate the impact of storage leeching, which pushes forward the understanding of Personal Clouds.

# Part II

# Exploring QoS in Social Storage Systems

# 6

# Analysis of QoS in Friend-to-Friend Storage Systems

**Summary**

Due to the growing necessity for secure and private off-site storage, it is increasingly common to find storage systems where users interact just with a set of trustworthy participants, such as in Friend-to-Friend (F2F) networks. In this Chapter, we argue that this kind of systems are highly affected by availability correlations and very small friendsets, which calls for a deep analysis of the storage QoS in this particular setting. We also inspect the applicability of traditional data management techniques (e.g. data availability, redundancy calculation) in this context. Moreover, to overcome the QoS limitations of purely decentralized systems, we propose a hybrid architecture that combines F2F storage systems and the availability of cloud storage services to let users infer the right balance between user control and QoS.

*The papers with the results of this chapter appeared in [26, 27].*

## 6.1 Introduction

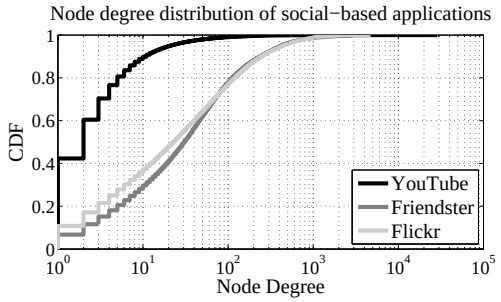
Friend-to-Friend (F2F) storage systems are currently an interesting research topic and they constitute an alternative approach to leverage personal storage [20, 22, 25, 26, 61]. The F2F paradigm is based on the synergy between social networks and storage systems: users store their data in a set of trustworthy friends. Thus, data is neither stored in a centralized server nor in unknown peers, enabling users to retain the *control of their data*. Moreover, the social component of F2F systems alleviates many undesirable problems present in large-scale storage systems —e.g. security, trust, incentives.

Generally, F2F storage systems have been treated as a particular case of P2P systems where nodes are connected by social relationships [20, 61]. However, very little attention has been paid to the characterization of these systems. Understanding the characteristics of F2F systems is crucial for providing an adequate storage service to users (Fig. 6.1). However, we consider two main aspects which clearly differentiate F2F systems from traditional P2P systems:

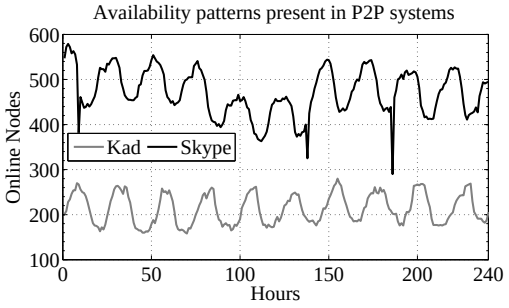
**High Availability Correlations**: Availability correlation can be understood as the high probability that given an ON (OFF) user, his friends are ON (OFF) as well. Indeed, measurements of online social networks have shown that friends present significant correlation in their activity patterns [145]—in line with other popular P2P applications [102, 103]. This implies that it is probable to find all friends of a user simultaneously offline (e.g. night) which makes it impossible to maintain high data availability even when placing one replica at each friend [99]. Fig. 6.1b shows the presence of correlations in real P2P systems.

**Extremely Small Friendsets**: Users in a F2F storage system are likely to hold a *reduced* number of trustable friends. To wit, over 63% of Facebook users have less than 100 friends [65], 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 [65]. As we can see in Fig. 6.1a, other social-based applications present even lower connectivity among users.

The combination of these issues poses new challenges which remain unsolved in a F2F scenario. Thus, our first contribution is to analyze the applicability of traditional data management techniques (e.g. data availability, redundancy calculation) in this context and their effect on the achievable storage QoS (e.g., data availability, transfers). Moreover, provided that traditional techniques to estimate data availability are severely biased in a F2F environment, we propose history-based method to calculate data availability tailored to heterogeneous and correlated availabilities.




**(a)** Degree distribution of social-based content distribution applications [146].




**(b)** Availability patterns of users in real P2P systems (10 days, 1000 nodes).

**Figure 6.1:** Characterization of F2F storage systems.

Unfortunately, the previous analysis confirms our intuition that it is extremely hard for a F2F system to provide a good storage service to users due to availability correlations. To improve the QoS, our idea was to take advantage of the superior availability of the cloud to find the right balance between user control and QoS.

The second contribution of this Chapter is a hybrid architecture called `F2Box` that combines F2F systems and cloud storage to get "the best of both worlds". The design of `F2Box` allows users to decide the level of QoS they want to obtain in basis 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.

To improve upload and download transfer times, `F2Box` is accompanied by new scheduling policies at two levels: at the friend-to-friend level and at the friend-to-cloud level. Further, a new method to adjust the amount of redundancy as a function of the availability patterns is introduced in `F2Box`. Finally, we analyze the existing trade-off between QoS and monetary cost. In summary, our contributions in this Chapter are the following:

- We analyze the impact of availability correlations on data availability provided by a small group of friends. Contrary to conventional wisdom, we found that correlations can be exploited to achieve an adequate trade-off between data availability and redundancy.

- We evaluate the performance of common approaches of estimating data availability when users are correlated. Given that these techniques are severely biased in a F2F en-

vironment, we propose history-based method to calculate data availability tailored to heterogeneous and correlated availabilities.

- We explore the relationship between data availability and download times. Our results suggest that, due to availability patterns, we should distinguish between if a file is currently available and if it is retrievable in a reasonable amount of time.

- We realize that it is hard to achieve an adequate storage service in a pure decentralized F2F system. To improve the QoS of a purely decentralized system, our idea is to take advantage of a cloud to find the right balance between user control and QoS.

- We design a hybrid F2F storage architecture called `F2Box`. We evaluate several aspects of this design, including the existing trade-off between QoS and monetary cost.

The remainder of this Chapter is structured as follows. In Section 6.2 we model a decentralized F2F system that will help us understand the main QoS problems in this setting. Section 6.3 presents a new approach to estimate data availability in a F2F system, and how we benefit from it to calculate the appropriate level of data redundancy. The results of our analysis of storage QoS in F2F systems appear in Section 6.4. We propose a hybrid alternative architecture to improve the storage QoS of F2F systems in Section 6.5 and we evaluate it in Section 6.6. We provide some technical discussion and conclude the Chapter in Section 6.7.

## 6.2 System Model

In this section, we extend the definitions of Chapter 2 to model thoroughly a F2F storage system. In our view, this model will help us to understand the storage QoS achievable in this scenario.

For a node $f$, we denote by $\mathcal{F}$ the set of friends at which $f$ wants to store data. We assume that this set is built up by leveraging upon real trust between users, for example, in an online social network (OSN) like Facebook or Orkut. Since our focus is on home users, we assume that node $f$ has *limited* download and upload bandwidth, denoted by $d_f$ and $\mu_f$, respectively. We also *limit* the number of parallel connections to $P_d$ and to $P_\mu$ for downloads and uploads, respectively. The storage capacity at node $f$ is denoted by $s_f$.

Friends alternate between *online* (ON) and *offline* (OFF) states. In addition, their online sessions may be correlated over time. Correlation can be understood as the high probability

that given an online user, his friends are online as well, which corresponds well with the strong diurnal pattern observed empirically in OSNs like Facebook [145].

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 data availability that otherwise would remain hidden.

Technically, to represent the availability of a host $f$, we use a vector $AV_f$ of size $T$, where its $i$th position $AV_v[i] = 1$ if $f$ was ON at time $t_i$, or 0 otherwise, where $t_i = \Delta \cdot i$, $\Delta > 0$. $\Delta$ represents the length of a time slot. With this representation, we can measure both types of correlations adapting the metrics in [106] into Definition 10:

**Definition 10 (Presence Matching)** *The Presence Matching (PM) metric measures the level of coincidence of the ON sessions of two nodes $a, b$:*

$$PM_{a,b} = \frac{|AV_a[t] \cap AV_b[t]|}{|AV_a[t] \cup AV_b[t]|}, \forall t \in T \text{ where } AV_i[t] = 1, i \in \{a, b\}$$

*Analogously, the Disconnection Matching (DM) metric gives the same information about OFF durations ($AV_i[t] = 0$).*

We extend the above two metrics by calculating the average pairwise PM and DM measures within a group $\mathcal{F}$. We refer to these metrics by *Group Presence Matching* (GPM) and *Group Disconnection Matching* (GDM), respectively. Given a group $\mathcal{F}$, we calculate the GPM over $\mathcal{F}$ as follows:

$$GPM(\mathcal{F}) = \frac{\sum_{i,j \in \mathcal{F}, i \neq j} PM_{i,j}}{\sum_{i=1}^{|\mathcal{F}|-1} i} \tag{6.1}$$

Naturally, the calculation of GDM is analogous but for DM instead of PM.

### 6.2.1 Estimating Data Availability to Generate Redundancy

As the number of friends is small and they can be temporarily offline, F2F storage systems provide data availability ($\delta$) by means of redundancy. Concretely, to assure a given level of data availability, our F2F system makes use of *Erasure Codes* (ECs), which has been proven to be more efficient in terms of redundancy than replication [68]. As introduced in Chapter 2, 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 $n$.

Blocks generated during the encoding process are assigned to the friendset members. This assignment depends on a *data placement policy* —we overview various policies in Section 6.2.3. For analytical treatment, we denote by $b_f \in \{0, 1, \ldots, n\}$ the number of blocks assigned to a friend $f \in \mathcal{F}$. An assignment is represented as a vector $\vec{\mathbf{b}} = \left(b_1, \ldots, b_{|\mathcal{F}|}\right)$, where the $i$th position is the number of blocks $b_i$ stored at the $i$th friend.

Naturally, the amount of redundancy generated will depend on the availability of the nodes that store data blocks. Traditionally, given the number of fragments for a file $k$ and the target level of data availability $\delta$, the number of encoded blocks to upload $n$ and hence, the redundancy rate $\frac{n}{k}$, has been determined as follows:

$$\delta = \sum_{i=k}^{n} \binom{n}{i} \bar{a}^i (1-\bar{a})^{n-i}, \tag{6.2}$$

where $\bar{a}$ is the average host availability of a group $\mathcal{F}$. (6.2) simply accounts for all the possible combinations of finding $\geq k$ blocks out of $n$, and the probability that this happens.

Two important observations must be discussed here about (6.2). 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 (6.2). 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, by employing the mean node availability $\bar{a}$, the *binomial approximation* (Eq. 6.2) does not take into account the heterogeneity of node availabilities within a group. This could potentially introduce estimation errors if we consider heterogeneous friendsets. Recently, a *heterogeneity-aware calculation* has been proposed to calculate data availability with higher accuracy [110].

Considering a set of friends $\mathcal{F} = \{f_1, f_2, \ldots, f_{|\mathcal{F}|}\}$, the *combinadic* $C^{|\mathcal{F}|,i}$ is the lexicographically ordered list with all the $\binom{|\mathcal{F}|}{i}$ possible combinations of $i$ friends. In order to lexicographically sort this list, we consider the nodes in each combination sorted in ascending order using their subindexes. By abuse of notation we denote as $C_j^{|\mathcal{F}|,i}$ the $j^{th}$ element of $C^{|\mathcal{F}|,i}$. For example, for $|\mathcal{F}| = 3$: $C^{|\mathcal{F}|,2} = [[f_1, f_2], [f_1, f_3], [f_2, f_3]]$ and $C_2^{|\mathcal{F}|,2} = [f_1, f_3]$.

Once addressed the concept of combinadic, we have that:

$$\delta = \sum_{i=1}^{|\mathcal{F}|} \sum_{j=1}^{\binom{|\mathcal{F}|}{i}} \left[ \gamma(k, C_j^{|\mathcal{F}|,i}, \vec{\mathbf{b}}) \prod_{f \in C_j^{|\mathcal{F}|,i}} a(f) \prod_{f \in \mathcal{F} \setminus C_j^{|\mathcal{F}|,i}} (1 - a(f)) \right], \tag{6.3}$$

where $a(f)$ is the availability of friend $f$, and the function $\gamma$ selects which combinations from the storage set $\mathcal{L}$ store together at least $k$ data blocks, since they could store distinct amounts of data:

$$\gamma(k, \mathcal{L}, \vec{\mathbf{b}}) = \begin{cases} 1 & (\sum_{i \in \mathcal{L}} b_i) \geq k \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

The heterogeneity-aware approach provides a remarkably more precise notion of data availability than the binomial approximation. Moreover, Eq. 6.3 avoids restrictive assumptions which are not necessarily present in real systems (e.g. every block must be stored in a distinct node).

However, both Eq. 6.2 and Eq. 6.3 assume that hosts are not correlated, since they consider that a mean availability value is an accurate representation of a user's behavior. In our view, this is by far not true in F2F systems and it may imply that these approaches can highly underestimate or overlook the real data availability. This motivates us to present a more accurate approach for representing node availabilities, and even, the collective dynamics of a friendset.

### 6.2.2 The Problem of Scheduling with Availability Correlation

In Chapter 2, we defined the concept of *transfer*, which basically refers to the connection with a remote node that causes the transfer of a single block of data to it. Furthermore, we refer to a *schedule* as the set of transfers concerning the same data object. In this section, we specifically focus on understanding the role of correlated node availabilities on scheduling times (TTS).

Armed with the previous definitions, we are now in position to describe rather informally the problem of data scheduling on a small group of availability-correlated storage nodes by means of a simple formulation.

At this point, we assume that an appropriate $(k, n)$-erasure code has been selected according to the target data availability $\delta$ and the availability correlations for the storage nodes in $\mathcal{F}$, with $|\mathcal{F}| \geq n$. Similar to [67], let $S(i, t)$ denote the event that an encoded block has been transferred to a friend $i$ during time slot $t$. Also, let $\mathbb{1}_{S(i,t)}$ be the indicator variable that notes whether or not the encoded block has been transferred. We say that a schedule $S$ is *complete* if

$$\sum_{i=1}^{n} \sum_{t=1}^{TTS(S)} \mathbb{1}_{S(i,t)} = n, \tag{6.5}$$

where $TTS(S)$ is the TTS for schedule $S$. Let $\mathcal{S}$ denote the set of all *complete schedules*. For simplicity, let us consider the set of schedules where each friend receives at most $m$ fragments

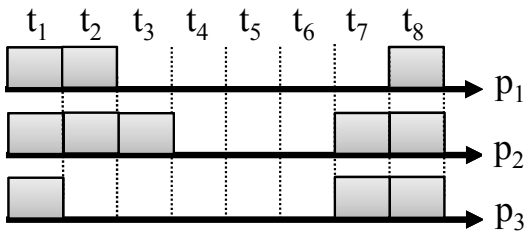



**Figure 6.2:** Example of availability correlation.

$\mathcal{S}_m$, i.e., $\mathcal{S}_m = \left\{S \in \mathcal{S} | \sum_{t=1}^{TTS(S)} \mathbb{1}_{S(i,t)} \leq m, \forall i \in \mathcal{F}\right\}$. Then, the goal of the *scheduling policy* will be to find the schedule $S$ in $\mathcal{S}_m$ with the shortest possible TTS.

For a schedule $S$, its corresponding $TTS(S)$ is equal to $\max\left\{t \in \mathbb{N}^+ | \mathbb{1}_{S(i,t)} = 1, i \in \mathcal{F}\right\}$. This implies that if the order in which transfers are to be executed does not take into account availability correlations, 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. 6.2, where $n = |\mathcal{F}| = 3$ and the set of potential schedules is $S_1$ (exactly one encoded fragment to each friend). We have depicted in gray the time slots where each node is online. This scenario highlights the importance of a good schedule when storage friends have correlated availabilities.

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. However, 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 $t_7$ to complete the upload. If the availability pattern of $p_3$ had been considered, this schedule would have been considered suboptimal and immediately discarded.

In a F2F system, finding the *most optimal scheduling plan* is key to provide an efficient storage service in terms of transfer times, which is not trivial due to the *reduced number of friends* and *the availability correlations among them*.

In our analysis, we evaluate the time needed to download data, both a per-block (BTT) and a per-object (TTS) basis, as a complementary metric to measure data availability. As one can infer, these times are influenced by the chosen *transfer scheduling policy*, that is, 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. As a baseline in our analysis, we make use of the *random policy*: among the pending block transfers the user chooses one of them uniformly at random until gathering at least $k$ blocks. We compare the performance of the random policy with the

*optimal time to schedule* (OTTS) to understand the margin of improvement that more elaborated scheduling policies may achieve.

### 6.2.3 Data Placement

Given a set of candidates, a data placement policy is the algorithm which decides the recipient of a data block. In this Chapter, we evaluate two distinct placements: *round-robin* (RR) and *availability proportional* (AP).

The RR data placement is extremely simple to implement and preserves fairness among friends regarding storage load. In our framework, this placement is used when we estimate the necessary redundancy using the *binomial approximation* (BA) and our *history-based* (HB) algorithm (see Section 6.3.2).

In [147], authors formally demonstrated that, having a group of nodes with heterogeneous availabilities, assigning an amount of redundancy proportional to their availabilities maximizes the resulting data availability. Hence, we employ the AP data placement in the *heterogeneity-aware* (HA) calculation[1].

We believe that preserving load balancing in a reduced set of participants is essential to provide scalable storage and to avoid service bottlenecks. Our objective is to quantify the impact of data placement on storage fairness and reliability provided by a friendset.

## 6.3 Historical Data Availability & Redundancy

In general, the majority of real-world systems express node availabilities with simple averages of their past behavior. With this information, it is simple to estimate the data availability provided by storage nodes with Eq. 6.2 or Eq. 6.3[2].

The relevance of heterogeneity and availability patterns reported in social networks [145] and many P2P systems [102, 103] poses an important evidence: *availabilities cannot be accurately estimated by averaging the fraction of time nodes have been online*. Furthermore, in a F2F scenario, the reduced number of storage nodes makes availability correlations to be even more important [104]. Such a simplification completely hides the correlated dynamics of nodes, which

[1] We thank Matteo Dell'Amico and Lluís Pàmies for developing a dynamic programming solution to perform this calculation.

[2] Other theoretical models to describe node dynamics, such as Markov chains, have reported limited applicability in practice [107, 148].

may, in turn, produce considerable problems to the storage service; for instance, significant errors in *the estimation of data availability*.

In our system, we calculate the data availability provided by a friendset $\mathcal{F}$ in a window of $T$ time units as follows:

$$\delta = \frac{1}{|T|} \sum_{t=0}^{T} \alpha(k, \mathcal{F}, t, \vec{\mathbf{b}}), \forall t \in T, \tag{6.6}$$

where $\alpha(k, \mathcal{F}, t, \vec{\mathbf{b}})$ is an indicator function which evaluates whether a file is available at instant $t$ as follows:

$$\alpha(k, \mathcal{F}, t, \vec{\mathbf{b}}) = \begin{cases} 1 & (\sum_{i \in \mathcal{F}} AV_i[t] \cdot b_i) \geq k. \\ 0 & \text{otherwise} \end{cases} \tag{6.7}$$

Therefore, we calculate the data availability provided by a group of friends based on their historical behavior. Note that the value of $\delta$ in Eq. 6.6 is basically the aggregation of the timeslots of instant data availability, that is, those periods where the number of available blocks is greater than $k$.

This approach provides an exact notion of the data availability provided by a set of friends in a past window of time, even in the presence of high heterogeneity and availability patterns. *Our objective is to benefit from this mechanism to calculate the precise amount of data redundancy $(n/k)$ to be stored in $\mathcal{F}$ for achieving a targeted data availability $(\delta)$.*

### 6.3.1 Historical Optimal Data Redundancy: Complexity

Ideally, the amount of redundancy used should be *minimal to optimize storage space*, provided that the target level of *data availability* is met. Moreover, since friendsets are normally small, we should *maintain load balancing for providing fairness and reliability* in such a limited system.

This problem can be defined as follows (Definition 11):

**Definition 11 (History-based Optimal Redundancy Problem)** *Given a fixed $k$ and a friendset $\mathcal{F}$, our objective is to find the minimal $n$, $n \geq k$, that achieves a targeted data availability $\delta$, where each friend $f \in \mathcal{F}$ stores a number of blocks $b_f$, $b_{min} \leq b_f \leq b_{max}$, where constants $b_{min}, b_{max} \in \{0, \ldots, n\}$.*

Solving this problem requires to *examine all the possible block assignments for each value of $n$*. The reason is that the resulting data availability depends on the availability history of friends, for which no assumption can be made on its exact behavior. Further, it can be easily seen that the optimal block assignment with $n + 1$ blocks will never provide less data availability than

the optimal one with $n$ blocks. Consequently, we will be able to use binary search to optimize the search in $n$ (Algorithm 1).

**Algorithm 1:** Historical Optimal Data Redundancy

```
Input: F, δ, k, b_min, b_max
Output: n
n ← k, t ← |F| · k;
while n ≠ t do
    m = (n+t)/2;
    if maxDA(F, k, m, b_min, b_max) < δ then
        n ← m + 1;
    else
        t ← m;
    end
end
```

The most computationally expensive part of Algorithm 1 lies on the function $maxDA$. This function looks for the block assignment that maximizes data availability under the established redundancy and load balancing constraints. Intuitively, we confront a combinatorial optimization problem. To illustrate its complexity, we formalize the $maxDA$ function as a *bounded knapsack problem* [149].

In our formalization, we assume that each friend has a *weight* $b_f$, that is, the number of blocks it stores. Moreover, to fit our problem into the formal knapsack definition, each friend $f$ has a specific value function $v_f$ defined as follows:

$$v_f(\vec{\mathbf{b}}) = \frac{1}{|T|} \sum_{t=0}^{T} AV_f[t] \cdot \tau(k, \mathcal{F}, t, \vec{\mathbf{b}}) \tag{6.8}$$

$$\tau(k, \mathcal{F}, t, \vec{\mathbf{b}}) = \begin{cases} \frac{1}{\rho(\mathcal{F}, t, \vec{\mathbf{b}})} & \alpha(k, \mathcal{F}, t, \vec{\mathbf{b}}) = 1 \\ 0 & \text{otherwise} \end{cases} \tag{6.9}$$

where the function $\rho(\mathcal{F}, t, \vec{\mathbf{b}}) = \sum_{f \in \mathcal{F}} AV_f[t] \cdot b_i$ represents the total number of available blocks at instant $t$. Therefore, $v_f$ expresses the relative contribution of a friend $f$ to the time periods where the file is available. Then, the optimization problem is:

$$\text{maximize} \quad V = \sum_{f \in \mathcal{F}} v_f(\vec{\mathbf{b}}) \cdot b_f, b_{min} \leq b_f \leq b_{max} \tag{6.10}$$

$$\text{subject to} \quad \sum_{f \in \mathcal{F}} b_f \leq n \tag{6.11}$$

The additional complexity of this problem w.r.t. the classical knapsack problem is that coefficients $(v_f, b_f)$ depend on the assignment $\vec{\mathbf{b}}$, and therefore, they should be recalculated at each execution step. Moreover, one can infer that depending on the assignment of data blocks within a friendset $\mathcal{F}$, the resulting data availability will vary. This means that distinct assignments of equivalent redundancy and load balancing constraints may lead to different $V$ values due to the collective dynamics of nodes. Thus, the complexity of our problem extends beyond the classical non-linear bounded knapsack problem, which is known to be NP hard [149].

For this reason, we propose a heuristic method to take advantage of the historical information in an efficient manner.

### 6.3.2 Estimating Data Redundancy with the History of Friends

As aforementioned, availability correlation in conjunction with a small friendset makes it hard to maintain a high data availability at any moment. This calls for a new notion of data availability in this context:

**Definition 12 (Daily Data Availability)** *The daily data availability metric aims to express the amount of hours per day a file object is available in a storage system.*

That is, we strive to ensure a high data availability during the period of the day where a user's friends are mostly online. The new perspective of data availability presented in Definition 12 specifically *benefits from correlations* to provide an feasible and scalable F2F storage service. More specifically, our aim is to assure a high data availability during at least $\delta$ hours per day, instead of at all times as in traditional large-scale storage systems, where a sufficient number of uncorrelated nodes can be found.

The algorithm for this computation works as follows. The initial number of blocks to be transferred to the friendset is $n = k$. The algorithm then assigns the $n$ blocks to the friends in a *round-robin* style in order to balance storage costs. Using a past time window of $w$ timeslots, it computes the number of timeslots $w_{timeslot}$ within the window where the number of available blocks $n_{avail}$ is equal or greater than $k$. Note that at least few days should be considered in the time window $w$. If $w_{timeslot}$ times the duration of a timeslot $\Delta$ covers $\delta$ hours, the algorithm halts, and the value of $n$ is returned. Otherwise, $n$ is incremented by one block and the entire process is repeated (up to $|\mathcal{F}|$ replicas). This procedure is repeated again and again until the value of $n$ guarantees $\delta$ hours of data availability. The pseudocode for this algorithm is shown in Algorithm 2.

**Algorithm 2:** Historical data redundancy calculation

**Input**: $\mathcal{F}$, $\delta$, $k$
**Output**: $n$
$n \leftarrow k - 1$;
$h_{\text{avail}} \leftarrow 0$
**while** $h_{avail} \leq \delta$ ***and*** $\frac{n}{k} < |\mathcal{F}|$ **do**
  $n = n + 1$;
  $n_{\text{timeslot}} \leftarrow 0$;
  **for** $i$ ***in*** $w$ **do**
    $n_{\text{avail}} \leftarrow 0$;
    **for** $f$ ***in*** $\mathcal{F}$ **do**
      **if** $AV_f[i] \equiv 1$ **then**
        $n_{\text{avail}} = n_{\text{avail}} + n/|\mathcal{F}|$;
      **end**
    **end**
    **if** $n_{avail} \geq k$ **then**
      $n_{\text{timeslot}} = n_{\text{timeslot}} + 1$;
    **end**
  **end**
  $h_{\text{avail}} \leftarrow n_{\text{timeslot}} \cdot \Delta$
**end**

## 6.4 Analysis of storage QoS in F2F systems

### 6.4.1 Setup & Methodology

In our tests, we modeled the alternating ON/OFF behavior of nodes using availability traces. Unfortunately, we could not find real traces of any F2F storage system, simultaneously including social graph and availabilities [99]. Thus, to evaluate the impact of availability correlations we employed real traces from *Kad* [102] and Skype [103]. Additionally, we used synthetically generated [150] traces of an Heterogeneous Yao model[1] [151]. From these traces, we excluded all the nodes whose availability was out of the range $[\bar{a}_{min}, \bar{a}_{max}]$ every 48 hours during the simulation window of $T = 12$ days. This filtering process was necessary to exclude from simulation extreme availability cases (e.g. superpeers, *permanent* churn). As a result, we obtained nodes with regular participation in the system and exhibiting strong correlation (see Fig. 6.3). Friendsets are formed selecting random nodes from these traces. Note that forming random groups from a strongly correlated trace will *generally* result in correlated groups.

[1]The average online session for friends was of 8 hours while the average downtime was of 16 hours ($\bar{a} = 0.33$). Both ON/OFF time durations were drawn from Pareto distribution (heavy-tailed) with shape parameter $\alpha = 3$, which has been reported to provide a tight fit to the real lifetime distribution found in decentralized systems.

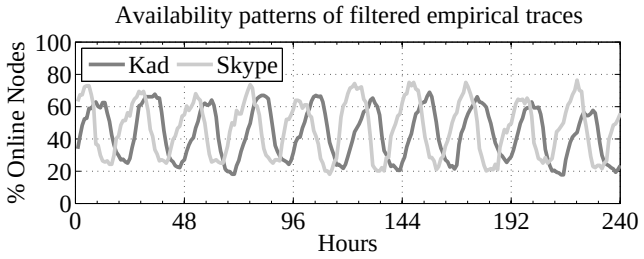



**Figure 6.3:** ON/OFF behavior of nodes in our traces after the filtering process.

| | | |
|---|---|---|
| $N$ | Filtered nodes | 122 (Skype), 192 (Kad)<br>85 (Yao) |
| $[\bar{a}_{min}, \bar{a}_{max}]$ | Node mean availability | Kad, Skype $[0.2, 0.8]$<br>Yao $[0.3, 0.4]$ |
| $T$ | Simulation time | 12 days |
| $\mu_f, d_f$ | Node up/down bandwidth | 30KBps, 120KBps |
| $s_f$ | Node storage capacity | 500GBytes |
| $P_\mu, P_d$ | Parallel up/down connections | 1, 4 |
| $k$ | Original file fragments | 40 |
| $\beta$ | Object size | 500MB, 1GB, 2GB |
| $\lvert\mathcal{F}\rvert$ | Friendset size | $[5, 10, 20]$ |

**Table 6.1:** System parameters and description.

Our simulations are divided into three different phases, each during 4 days:

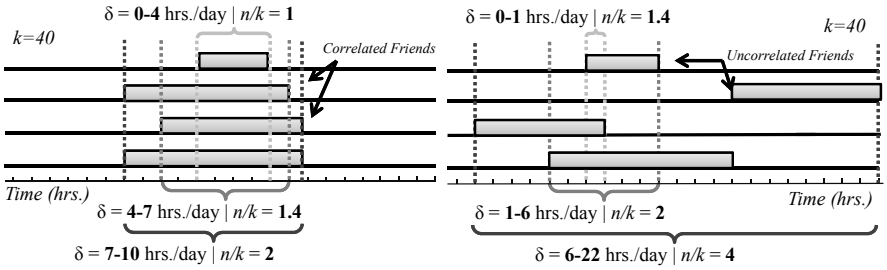
1. *Training phase*. In this phase, the users collect historical information in form of availability vector $AV$ about each of their respective storage friends.

2. *Upload phase*. During this phase, the data owner uploads a single file of $\beta$ bytes, plus the associated redundancy, to the system.

3. *Download phase*. Finally, in the *download phase* a user retrieves $k$ data blocks from the system to reconstruct the uploaded file. During this phase we continuously inspect the *data availability* provided by the group of friends to which the file was uploaded. We measure data availability as the number of time-slots where the number of available blocks is $\geq k$ (Eq. 6.6).

In our F2F application, retrieving a file requires locating a sufficient number of blocks to perform a decoding operation. Thus, a user needs the network information of their storage friends in the system to initiate the downloading process. We do not restrict the way of storing this information; for instance, it can be stored in a social application or a tracker.

The important simulation parameters used in our simulations are depicted in Table 6.1. Simulation results for each trace correspond to a collection of $1,000$ random friendsets.




**(a)** Friendset with high GPM. **(b)** Friendset with low GPM.

**Figure 6.4:** Impact of GPM on the data redundancy ($n/k$) a friendset requires to provide a certain degree of data availability ($\delta$). Correlated friends provide low/moderate data availability values using less redundancy than uncorrelated ones. However, only uncorrelated friendsets can provide high data availability.

### 6.4.2 Availability Correlations and Data Availability

Generally, the presence of availability correlations has been considered in the P2P literature as a flaw which should be avoided to guarantee a high data availability. However, the actual impact of correlated availabilities on the data availability provided by a small group of participants remains unexplored.

To address this issue, we resort to the GPM metric to measure the degree of coincidence among the online sessions of a group of friends. We analyze the resulting data availability when these friends are highly correlated in their online sessions (high GPM) and in the opposite case (low GPM). Both cases are illustrated in Fig. 6.4. This figure depicts the relationship between data availability and redundancy depending on the GPM of a friendset.

For this analysis, we have synthetically generated 1K friendsets of cardinality 5 for both categories of GPM $\in [0.05, 0.15]$ and GPM $\in [0.3, 0.4]$ from the Yao trace during the download phase. Note that from this trace we excluded all the nodes whose availability was out of the range $[0.3, 0.4]$ every 48 hours during the simulation. This will give us a clear picture of the impact of availability correlations, without the bias induced by high node heterogeneity.

First, in Fig. 6.5 we observe a clear distinction in the growth of data availability as a function of data redundancy depending on the GPM degree of a friendset. For low to moderate amounts of data redundancy, we see that *availability correlations improve the data availability provided by the friendset*. For instance, when two replicas are introduced in the system ($n/k = 2$), correlated friendsets double the data availability provided compared with the uncorrelated friendsets.

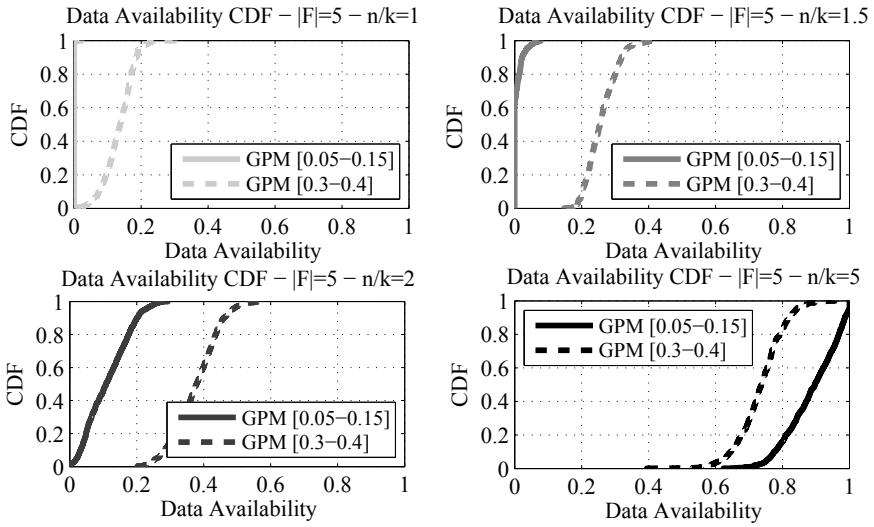
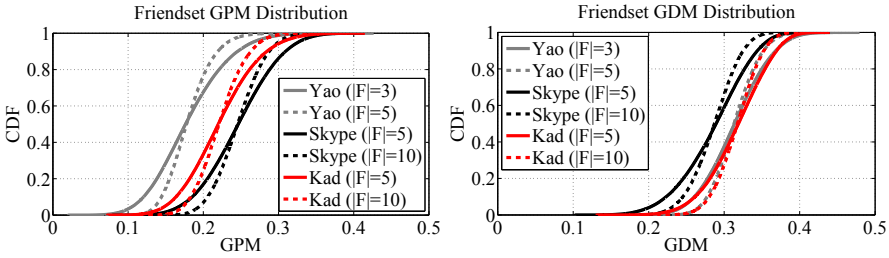



**Figure 6.5:** Evolution of data availability in function of the data redundancy for correlated/uncorrelated friendsets (H. Yao trace, RR placement).

Although this observation may be contrary to conventional wisdom, it is clearly depicted in Fig. 6.4. In Fig. 6.4a, we observe that correlated friends provide moderate levels of data availability at a lower storage cost. The reason is simple: if friends coincide in their online sessions, data availability is maintained by all friends simultaneously. This reduces the amount of data redundancy needed *at each friend*. On the contrary (Fig. 6.4b), friendsets exhibiting low GPM have almost no common online periods among them. This implies that they should support large amounts of redundancy per friend (even a replica) to guarantee a high data availability.

However, when we introduce large amounts of data redundancy we observe that this behavior changes. In case of $n/k = 5$, uncorrelated friendsets provide a significantly higher data availability than correlated ones. That is, *for very large amounts of data redundancy, uncorrelated friends are able to cover the majority of the day time with k blocks or more*. On the other hand, we observed that if friends within a group coincide in their online sessions, they also coincide in their offline sessions (high GDM). This fact lead us to an important conclusion: *in a correlated friendset, the maximum data availability achievable is limited by the degree of coincidence in their offline sessions (GDM)*.

Fig. 6.6 depicts the GPM/GDM distribution of 50K random friendsets selected from availability traces. First, in Fig. 6.6a we observe that the GPM distribution exhibits a wide range ($[0.1, 0.35]$) in all traces due to the randomness of the node selection. Since there are no real




**(a)** GPM Distribution. **(b)** GDM Distribution.

**Figure 6.6:** GPM/GDM distribution of random friendsets of different cardinalities collected from employed traces during the download phase (4 days).

traces of F2F storage systems, we cannot generalize the observed GPM values in our traces to real systems. In our opinion, trusted friends will likely live in the same time-zone, probably exhibiting similar patterns [145]. In that case, the GPM distribution would be remarkably concentrated in high values.

Second, we observe that the GDM distribution of these traces exhibits higher values than the GPM distribution. There are two reasons for this phenomena: i) Low node availabilities make the coincidence of nodes in their offline sessions easier, and ii) the nocturnal patterns of real traces increases the probability for a group of friends to be simultaneously offline.

In line with [104], we found that *it is easier to find high availability correlations among participants as the friendset becomes smaller*. This can be appreciated in Fig. 6.6: For small friendsets ($|\mathcal{F}| = 3$) the GPM/GDM distributions present more extreme values than for larger friendsets ($|\mathcal{F}| = 10$).

In conclusion, in F2F systems, the presence of availability correlations offers a good trade-off between data availability and data redundancy.

### 6.4.3 Data Redundancy Estimation

Next, we study the accuracy of the traditional ways of calculating the necessary amount of data redundancy to achieve a certain data availability. To this end, in Fig. 6.7 we illustrate the data availability experienced by 1K friendsets of distinct cardinalities ($|\mathcal{F}|$) when varying the required data availability ($\delta$).

In Fig. 6.7, we clearly observe that for low $\delta$, the *binomial approximation* (BA) tends to *greatly overestimate the amount of data redundancy required*. Proof of that is that the system exhibits a

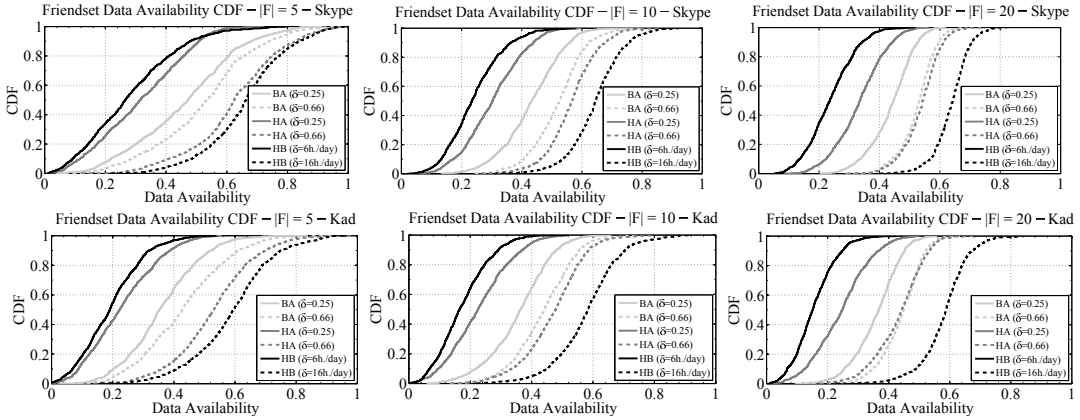



**Figure 6.7:** Data availability obtained by different redundancy calculations approaches varying the targeted data availability ($\delta$) and the friendset size ($|\mathcal{F}|$).

**Table 6.2:** Mean data availability deviation ($\mu - \delta$) and coefficient of variation (CV) of redundancy calculation techniques - Skype

| $\delta = 0.25$\|6 hours/day | BA ($\|\mathcal{F}\| = 5$) | HA ($\|\mathcal{F}\| = 5$) | HB ($\|\mathcal{F}\| = 5$) |
|---|---|---|---|
| $\mu - \delta$ | $+0.21(+84\%)$ | $+0.07(+28\%)$ | $+0.03(+12\%)$ |
| CV ($\sigma/\mu$) | 0.37 | 0.46 | 0.54 |
| $\delta = 0.66$\|16 hours/day | BA ($\|\mathcal{F}\| = 5$) | HA ($\|\mathcal{F}\| = 5$) | HB ($\|\mathcal{F}\| = 5$) |
| $\mu - \delta$ | $-0.14(-21.2\%)$ | $-0.04(-6.1\%)$ | $0.0(0\%)$ |
| CV ($\sigma/\mu$) | 0.32 | 0.25 | 0.21 |
| $\delta = 0.25$\|6 hours/day | BA ($\|\mathcal{F}\| = 20$) | HA ($\|\mathcal{F}\| = 20$) | HB ($\|\mathcal{F}\| = 20$) |
| $\mu - \delta$ | $+0.19(+76\%)$ | $+0.08(+32\%)$ | $-0.01(-4\%)$ |
| CV ($\sigma/\mu$) | 0.17 | 0.25 | 0.34 |
| $\delta = 0.66$\|16 hours/day | BA ($\|\mathcal{F}\| = 20$) | HA ($\|\mathcal{F}\| = 20$) | HB ($\|\mathcal{F}\| = 20$) |
| $\mu - \delta$ | $-0.13(-19.7\%)$ | $-0.12(-18.2\%)$ | $-0.02(-3\%)$ |
| CV ($\sigma/\mu$) | 0.11 | 0.12 | 0.09 |

much higher data availability than expected. In this sense, the *heterogeneity aware* (HA) calculation significantly improves the accuracy of the BA, thereby demonstrating the importance of considering heterogeneous availabilities [110].

On the other hand, both *BA and HA highly underestimate the necessary amount of redundancy needed for high values of $\delta$, providing lower data availabilities than expected*. This is mainly due to strong availability correlations. Further, this phenomenon seems to be more evident as the friendset size grows since most members exhibit nocturnal patterns (Fig. 6.7).

We observe that, *our history-based (HB) redundancy calculation is very accurate in the presence of correlations, irrespective of $\delta$ and $|\mathcal{F}|$*. In Table 6.2, we illustrate the mean data availability deviation ($\mu - \delta$) provided by each method and the resulting coefficient of variation (CV). For instance, for $|\mathcal{F}| = 5$ and $\delta = 0.25$, the HB method has a mean deviation of $+12\%$ from the expected availability, whereas the HA and BA exhibit a deviation of $+28\%$ and $+84\%$

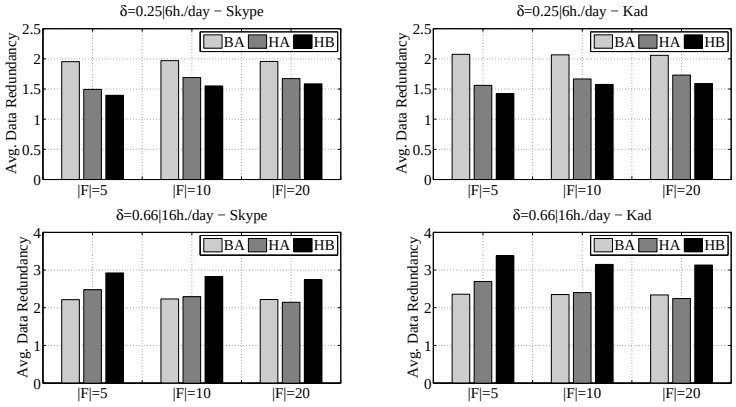



**Figure 6.8:** Average data redundancy factor introduced by BA, HA and HB.

respectively.

In Table 6.2 we also infer that the data availability CV is higher for low $\delta$ values and small friendsets. This is congruent with results in Section 6.4.2, where in case of correlated friendsets and low $\delta$, small increments of redundancy result in significant improvements in data availability. This implies that small variations of redundancy induce high variability (CV).

It is worth mentioning that the HB method provides a slightly lower data availability than expected for $\delta = 0.25$ in Kad. By inspecting the Kad trace, we noted that the selected nodes exhibited lower availabilities in the download phase (0.397) than in the training phase (0.449). This induces an underestimation of the necessary redundancy, as the HB algorithm uses the availability vectors of nodes during the training phase. Therefore, the HB method can lead to incorrect redundancy estimations in case of significant variations between the friendset history and the current friendset availability.

The resulting data availability comes from the generated data redundancy. Fig. 6.8 illustrates the differences in the average data redundancy factor exhibited by the different redundancy calculation approaches. When a user demands high data availability ($\delta = 0.66$), we observe that both BA and HA calculations provide much less redundancy than the HB approach (from $-30\%$ to $-15\%$). Further, the deviation of BA and HA calculations techniques causes friendsets to not meet the required data availability (Fig. 6.7, Table 6.2).

Clearly, for low values of $\delta$ both BA and HA introduce more data redundancy than the HB approach. That is, in Skype for $|\mathcal{F}| = 10$ and $\delta = 0.25$, the BA and HA store 26.97% and 4.13% more redundant data than our proposal, respectively. Furthermore, in that case, this extra redundancy is unnecessary since the BA and HA calculations provide more data availability

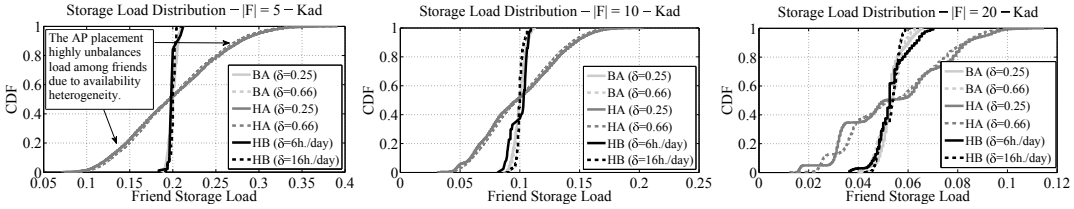



**Figure 6.9:** Storage load balancing supported by nodes. Clearly, the BA and HB redundancy calculation algorithms preserve load balancing due to the use of RR placement whereas the AP placement incurs in high unbalance.

than expected. In a F2F system, *this burden in terms of unnecessary redundancy may pose important drawbacks (e.g. limited scalability, high upload times)*.

In conclusion: i) Due to availability correlations, the BA and HA approaches produce significant deviations for calculating the necessary data redundancy for a targeted data availability, and ii) Our HB method is accurate enough to produce an adjusted amount of data redundancy in the presence of correlations. This provides important benefits to the system in terms of data availability and storage scalability.

### 6.4.4 Storage Load Balancing and Reliability

We investigate next the impact of data placement on the load balancing and reliability of the storage service. To this end, we compare two placement policies described in this Chapter: *round-robin* (RR) and *availability proportional* (AP) policies. As mentioned in Section 6.2.3, we use the RR placement for the BA and HB redundancy calculation algorithms. The AP placement is inherently used by the HA calculation.

Fig. 6.9 depicts the storage load CDF experienced by nodes in our simulations. Noticeably, *the AP assignment induces a high storage unbalance. This is specially evident for very small and heterogeneous groups.* To wit, for $|\mathcal{F}| = 5$, we note that the 20% of nodes store less than 15% of a file, whereas a 4% of nodes suffer a load greater than 30%.

Conversely, irrespective of the redundancy calculation and the values of $\delta$ and $|\mathcal{F}|$, *the RR placement offers high load balancing*. As expected, as the value of $|\mathcal{F}|$ grows, the differences between both placements become less important.

In our view, load balancing is a key property for such a limited storage system. Poor load balancing may in fact produce severe service problems.

In this sense, Table 6.3 presents the file recovery probability in the presence of node failures. In these simulations, we loaded friendsets with a certain amount of data redundancy ($n/k$)

**Table 6.3:** File recovery probability in the presence of failures

| | Friend Random Failures ($f_r$) - $\|\mathcal{F}\| = 5$ (Kad) | | |
|---|---|---|---|
| $n/k = 1.5$ | $f_r = 1$ | $f_r = 2$ | $f_r = 3$ |
| Av. Prop. | 100% | 99.6% | 21.9% |
| Round Robin | 100% | 100% | 0% |
| $n/k = 3$ | $f_r = 3$ | $f_r = 4$ | $f_r = 5$ |
| Av. Prop. | 100% | 85.2% | 1.1% |
| Round Robin | 100% | 100% | 0% |
| | Most Av. Friend Failures ($f_a$) - $\|\mathcal{F}\| = 5$ (Kad) | | |
| $n/k = 1.5$ | $f_a = 1$ | $f_a = 2$ | $f_a = 3$ |
| Av. Prop. | 100% | 97.7% | 0% |
| Round Robin | 100% | 100% | 0% |
| $n/k = 3$ | $f_a = 3$ | $f_a = 4$ | $f_a = 5$ |
| Av. Prop. | 100% | 17.5% | 0% |
| Round Robin | 100% | 100% | 0% |
| | Friend Random Failures ($f_r$) - $\|\mathcal{F}\| = 10$ (Skype) | | |
| $n/k = 1.5$ | $f_r = 3$ | $f_r = 4$ | $f_r = 5$ |
| Av. Prop. | 100% | 80.5% | 0% |
| Round Robin | 100% | 100% | 0% |
| $n/k = 3$ | $f_r = 6$ | $f_r = 7$ | $f_r = 8$ |
| Av. Prop. | 100% | 93.8% | 25.7% |
| Round Robin | 100% | 100% | 0% |
| | Most Av. Friend Failures ($f_a$) - $\|\mathcal{F}\| = 10$ (Skype) | | |
| $n/k = 1.5$ | $f_a = 3$ | $f_a = 4$ | $f_a = 5$ |
| Av. Prop. | 100% | 0% | 0% |
| Round Robin | 100% | 100% | 0% |
| $n/k = 3$ | $f_a = 6$ | $f_a = 7$ | $f_a = 8$ |
| Av. Prop. | 98.9% | 1.4% | 0% |
| Round Robin | 100% | 100% | 0% |

using the AP and RR placements. Moreover, we considered two failure models across a group: random failures and failures occuring to the most available friends.

In general, *random failures within a friendset exhibit a similar impact on both placement strategies*. However, in extreme cases of random failures (rightmost column) we observe that the AP assignment provides better resilience. This is due the fact that random failures can occur to the majority of lowest available (and therefore least loaded) friends, thus providing higher recovery probabilities. In other cases we observe slightly better results from the RR strategy.

Nevertheless, when highest available nodes fail, *the RR placement offers a greater resilience than the AP*. Actually, in the Skype scenario, the RR strategy tolerates one node failure more than the AP placement. This represents a difference of 10% in the recovery probability for the same amount of storage redundancy. Therefore, we conclude that in a F2F scenario it is important to preserve storage load balancing to provide fairness and reliability, as well as to avoid service bottlenecks.

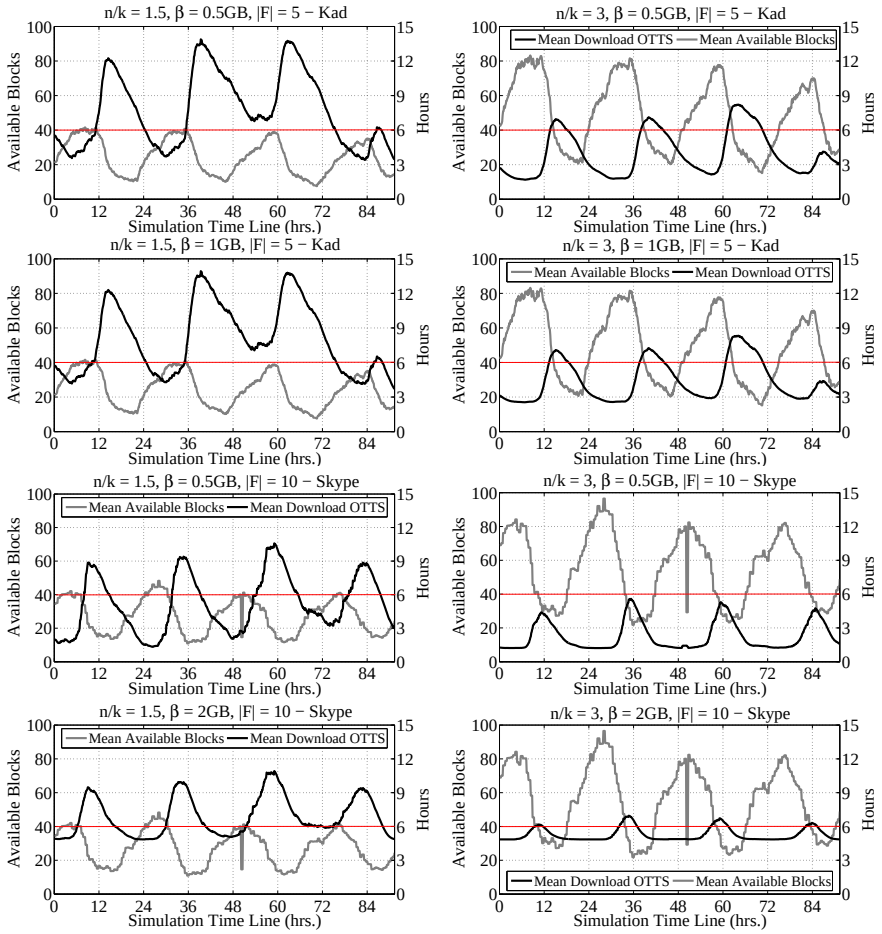
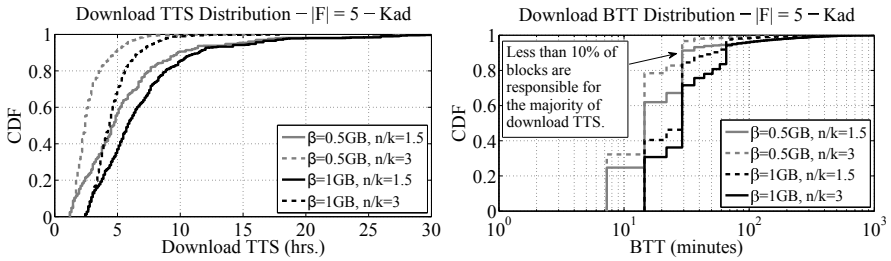



**Figure 6.10:** Relationship between data availability and optimal download times (RR placement).

### 6.4.5 Data Availability vs Download Times

Conversely to large-scale systems, providing a certain degree of data availability in a F2F network does not necessarily imply that data can be retrieved in a short period of time. This is mainly due to the *reduced number of available friends, their bandwidth limitations and their availability patterns*. In this section, we analyze the relationship between availability and download times in a F2F system.

Fig. 6.10 illustrates the behavior of the data availability and download optimal times to schedule (OTTS) during a time series analysis of 4 days for different values of $n/k$, $|\mathcal{F}|$ and $\beta$. First, we can observe in a time-series representation the impact of availability correlations on the mean number of available blocks maintained by friendsets. Clearly, *the strong patterns of nodes produce periods of data redundancy over-provisioning and under-provisioning*. This effect




**(a)** TTS distribution. **(b)** BTT distribution.

**Figure 6.11:** TTS and BTT distributions of 1,000 downloads at the start of the download phase using a random scheduling policy (RR placement).

becomes more significant as the redundancy rate $n/k$ grows.

As can be inferred from Fig. 6.10, the availability correlations of nodes also impact on the download OTTS. As expected, the download times present a near-opposite behavior than data availability; the more redundancy is available, the shorter the download time a user can achieve. However, we also see that the download OTTS values are highly dependent on the redundancy introduced in the system. That is, in Kad for $\beta = 1$GB and $n/k = 3$, the mean download OTTS values range from 2.54h. to 8.32h. ($\mu = 4.39$h.), whereas when $n/k = 1.5$ they range from 3.67h. to 13.94h. ($\mu = 7.98$h.). Further, results for Skype suggest that *larger friendsets notably improve the time to retrieve a file from the system*.

It is interesting to note that both availability and download OTTS plots are not completely opposite to each other. To wit, *we can find periods without availability and low download times, being the contrary also true*. Hence, start downloading a file in the evening may take several hours, due to the simultaneous disconnection of nodes before the download is completed. On the contrary, retrieving a file in early morning exhibits low download times even though the availability at the start of the download indicates that the file is unavailable.

In Fig. 6.10 we observe that the lowest TTS values are clearly dependent on the file size ($\beta$). However, the largest download times seem to be less affected by the file size, specially in Kad. To explore this issue in depth, in Fig. 6.11 we present the TTS and BTT distributions obtained by a *random scheduling policy* for different values of $\beta$ and $n/k$.

As can be observed in Fig. 6.11a, we see that for low TTS values the file size plays an important role. The dominance of low TTS values in Fig. 6.11a is because these downloads are performed at the beginning of the day. However, we observe that *as the download times become higher, TTS distributions become similar irrespective of the value of $\beta$*. The reason of that is that the order in which transfers are executed do not take into account availability correlations. Hence,

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 BTT distribution.

In Fig. 6.11b we present the BTT distribution of the schedules depicted in Fig. 6.11a. Note that while the majority of blocks are transferred in a reasonable time, there is a reduced number of blocks ($< 10\%$) exhibiting very large transfer times. Thus, the presence of availability patterns cause the unavailability of these blocks, which importantly increase the final TTS. Therefore, irrespective of the file size, *the largest TTS values are dictated by a minimal fraction of blocks whose schedule is affected by availability correlations*.

## 6.5 F2Box: Cloudifying F2F Storage

In the previous analysis, we observed that the achievable storage QoS for a purely decentralized F2F system is negatively affected by the small groups of friends storing data and their correlated availabilities. From our point of view, there is little margin to improve these systems with novel data management techniques, since their limitations are inherent to their decentralized nature. For this reason, in this section we present a novel hybrid architecture where storage resources from users are blended with cloud storage to improve the storage QoS.

### 6.5.1 System Design

To improve the performance of F2F storage, we propose F2Box, a hybrid architecture that exploits the higher availability of cloud storage services. As pure F2F systems, F2Box nodes use their social links to set up symmetric storage relationships among them. Furthermore, each F2Box node incorporates 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 two-fold manner:

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

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 TTS by letting F2Box users to upload blocks to the cloud instead of waiting for the disconnected friends to come back on-line again.

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

### 6.5.2 Historical Data Availability in a Hybrid Environment

F2Box benefits from the concept of daily data availability (Definition 12) for providing $\delta$ hours per day of data availability mixing resources from friends and the cloud. In fact, we extend Algorithm 2 to deal with a hybrid environment.

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 maintained by the friendset. More specifically, we denote by $F_C$ the fraction of the files to be *permanently* stored at the cloud. A high value of $F_C$ translates into a high data availability, since any cloud service has been designed to reach several nines of uptime availability. Notice 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.

The remaining proportion of the file $1 - F_C$ is maintained by the friendset. Given a chosen $k$, this requires a new method to calculate the minimum redundancy rate $\frac{n}{k}$ according to the specific availability patterns of friends, so that data availability can be maintained at least $\delta$ hours per day. Recall that a file is available if at least $k$ blocks are available for download.

The algorithm for this computation works as follows. Given the number of blocks $k$, the algorithm computes the number of blocks to be uploaded to the cloud $n_{\text{cloud}}$ as $n_{\text{cloud}} = \lceil k \cdot F_C \rceil$. The initial number of blocks to be stored at friends is $n_{\text{friend}} = \lfloor k(1 - F_C) \rfloor$. The algorithm then assigns the $n_{\text{friend}}$ blocks to the friends in a *round-robin* style in order to balance storage costs. Using a past time window of $w$ timeslots, it computes the number of timeslots $n_{timeslot}$ within the window where the number of available blocks $n_{\text{cloud}} + n_{avail}$ is equal or greater than $k$. Note that at least few days should be considered in the time window $w$. If $n_{timeslot}$ times the duration of a timeslot $\Delta$ covers $\delta$ hours, the algorithm halts, and the values $n_{\text{cloud}}$ and $n_{\text{friend}}$ are returned. Otherwise, $n_{\text{friend}}$ is incremented by one block and the entire process is repeated. This procedure is repeated again and again until the value of $n_{\text{friend}}$ guarantees $\delta$ hours of data availability, whenever $n_{\text{friend}}$ does not exceed $|\mathcal{F}_v|$ replicas. Then, the redundancy rate is simply $(n_{\text{cloud}} + n_{\text{friend}})/k$. The pseudocode for this algorithm is shown below.

Note that Algorithm 3 gives flexibility to the user regarding the amount of data he wishes to store in the cloud. Moreover, depending on the targeted data availability $\delta$, the mone-

tary cost of cloud storage will be higher or lower, and the disk capacity required to friends will be on the opposite situation. We study this trade-off in the next section.

**Algorithm 3:** Hybrid historical redundancy calculation

**Input**: $\mathcal{F}_v$, $\delta$, $F_C$, $k$
**Output**: $n_{\text{cloud}}$, $n_{\text{friend}}$
$n_{\text{cloud}} \leftarrow \lfloor k \cdot F_C \rfloor$;
$n_{\text{friend}} \leftarrow \lceil k \cdot (1 - F_C) \rceil - 1$;
$h_{\text{avail}} \leftarrow 0$
**while** $h_{avail} \leq \delta$ ***and*** $\frac{n_{friend}}{k} < |\mathcal{F}_v|$ **do**
  $n_{\text{friend}} = n_{\text{friend}} + 1$;
  $n_{\text{timeslot}} \leftarrow 0$;
  **for** $i$ ***in*** $w$ **do**
    $n_{\text{avail}} \leftarrow 0$;
    **for** $u$ ***in*** $\mathcal{F}_v$ **do**
      **if** $AV_u[i] \equiv 1$ **then**
        $n_{\text{avail}} = n_{\text{avail}} + n_{\text{friend}}/|\mathcal{F}_v|$;
      **end**
    **end**
    **if** $n_{cloud} + n_{avail} \geq k$ **then**
      $n_{\text{timeslot}} = n_{\text{timeslot}} + 1$;
    **end**
  **end**
  $h_{\text{avail}} \leftarrow n_{\text{timeslot}} \cdot \Delta$
**end**

### 6.5.3 Improving Scheduling Times

Given determined the redundancy rate in terms of $n_{\text{cloud}}$ and $n_{\text{friend}}$, we resort to the cloud in order to decrease scheduling times. As described above, the cloud provider, in addition to store $n_{\text{cloud}}$ blocks, acts as a temporary repository to store the blocks assigned to nodes that are currently offline. This policy can lead to an important reduction of scheduling times.

To this aim, we propose the *Bandwidth Maximizing Friend-to-Cloud* policy[1]. With this policy a user seeks to minimize scheduling times as much as possible by fully utilizing his own bandwidth. Thus, if a node responsible for a block is not online, this policy automatically pushes this block to the cloud. This ensures that a node achieves the MTTS. In this sense, this scheduling policy can be viewed as a pure F2F scheduling policy that immediately uploads a block to the cloud when the corresponding friend to which transfer that block is offline.

[1] In this piece of research we designed other Friend-to-Cloud policies. However, the differences in performance among them were not significant.

In any case, the extra fragments allocated to the cloud will have to be downloaded afterwards by the friends to whom were initially assigned. Clearly, this yields extra cloud costs in outgoing traffic, which are analyzed in Section 6.6.

Finally, apart from the random scheduling policy presented in Section 6.2.2, we want to extend the performance analysis of data transfer scheduling policies introducing two new ones:

- *Least-Available First (LAF)*. This scheduling strategy is based on the assumption that nodes that have been online in the past will continue to do so in a near future. Therefore, it prioritizes transfers towards the nodes that have been less available within a past time window of $w$ timeslots [67].

- *Previous Optimal Schedule (POS)*. We propose a novel policy which takes advantage from the max-flow based calculation of the optimal time to schedule. Essentially, it works as follows. If a scheduling were to be started at time $t$ of day $d$, this policy would reproduce exactly the optimal schedule that would be obtained from the max-flow based approach but starting at time $t$ of day $d$ in the preceding week. This policy thus tries to take advantage of the regular participation of nodes over the week.

Our objective is to understand the effects of more elaborated transfer scheduling policies on transfer times and cost of the F2Box storage service.

## 6.6 Evaluation of F2Box

### 6.6.1 Setup & Methodology

Similarly to our previous analysis, in our tests we modeled the alternating ON/OFF behavior of nodes using real availability traces. Since a capital aspect of our evaluation lies on studying the impact of availability correlation, we employed traces from *Kad* [102] and Skype [103]. From these traces, we excluded all the nodes whose availability was out of the range $[0.2, 0.6]$ during the simulation window of $T = 12$ days. From this subset, we filtered out the nodes that were not online at least once every 48 hours. This filtering process was necessary to exclude from simulation extreme availability cases, such as superpeers or *permanent* churn.

Since we do not target backup scenarios, where each storage operation involves several Gigabytes of content, we consider that the data owner stays connected during the whole storage operation. This behavior is natural in file-sharing and storage applications. In addition,

| Parameter Description and Values | | |
|---|---|---|
| $N$ | Nodes in the system | 60 (Skype), 110 (Kad) |
| $\bar{a}$ | Node mean availability | $[0.2, 0.6]$ |
| $T$ | Simulation time | 12 days |
| $\mu, d$ | Node upload/download bandwidth | 30KBps, 120KBps |
| $s$ | Node storage capacity | 500GBytes |
| $P_\mu, P_d$ | Parallel upload/download connections | 1, 4 |
| $k$ | Original file fragments | 40 |
| $\beta$ | Object size | 2GB |
| $\|\mathcal{F}\|$ | Friendset size | 10 |

**Table 6.4:** System parameters and description.

each friendset served a single storage request. The impact of concurrent schedules within a friendset was addressed in the experimental evaluation.

Our simulations are divided into three different phases, each during 4 days:

1. *Training phase*. In this phase, the users collect historical information in form of availability vector $AV$ about each of their respective storage friends. This knowledge base is vital to calculate the value of $n_{\text{friend}}$, or to initialize the scheduling policies LAF and POS, among other matters.

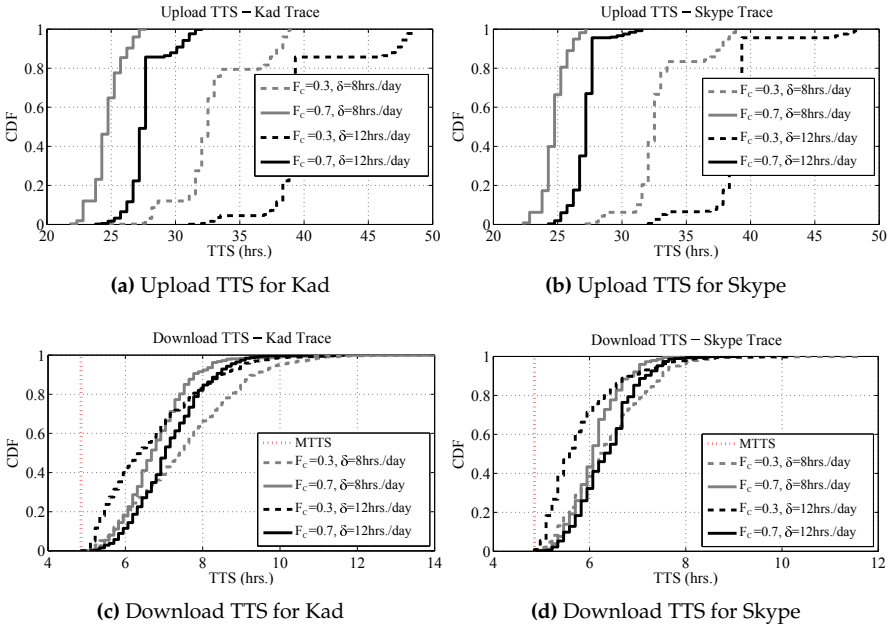
2. *Upload phase*. During this phase, the data owner uploads a single file of $\beta$ bytes, plus the associated redundancy, to the system.

3. *Download phase*. Finally, in the *download phase* a user retrieves $k$ data blocks from the system to reconstruct the uploaded file.

All the results presented below were obtained by repeating this process for $1,000$ random friendsets in each configuration.

Our analysis showed that applying scheduling policies in this scenario has little or no effect on *download* TTS. For this reason, download schedules are performed randomly, giving a greater priority to the available blocks stored at friends in order to save costs due to data transferred out of the cloud. Only in case of having idle connections, a node retrieves fragments from the cloud.

*Monetary Cost*. We adopt for our evaluation the same pricing model of Amazon S3, which is a well representative for cloud storage. Accordingly, we consider outgoing data transfers and storage to be charged by the cloud service. At February 2012, Amazon S3 pricing was of \$0.120 per GB of data transfer out of the cloud and \$0.140 per GB/month of storage. Transfers into the cloud are free of charge. Finally, it must be noted that we did not account for the cost




**(a)** Upload TTS for Kad

**(b)** Upload TTS for Skype

**(c)** Download TTS for Kad

**(d)** Download TTS for Skype

**Figure 6.12:** Scheduling times as a function of $F_C$ and the targeted data availability $\delta$ for friendsets of size $|\mathcal{F}| = 10$.

incurred by transactions (GET, PUT, COPY, POST, and LIST requests) due to the small charge per transaction: \$0.01 for every 10,000 transactions.

We compare the costs of F2Box with the same service provided by Amazon S3. We analyze the costs by *file operation*, which corresponds to upload and download a file of $\beta$ bytes, as well as keeping it stored for one month in the system.

### 6.6.2 Results

**Scheduling times**. In first place, we comment on the scheduling times obtained by F2Box and depicted in Fig. 6.12a for Kad, and in Fig. 6.12b for Skype, respectively. As can be seen in both figures, the higher the amount of data transferred to the friends, the longer the upload *time to schedule*. This is because a higher value of $1 - F_C$ translates into a greater number of blocks $n_{\text{friend}}$ to be uploaded in order to assure the same level of data availability. This fact is clearly visible in the case of demanding a *high data availability* like a $\delta$ of 12 hours/day.

It is worth to note that the steeper slope that is observed in Fig. 6.12a and 6.12b decreases as more blocks are stored at friends. The reason of this lies on the redundancy calculation we make to ensure data availability in the presence of high availability correlation. However, the

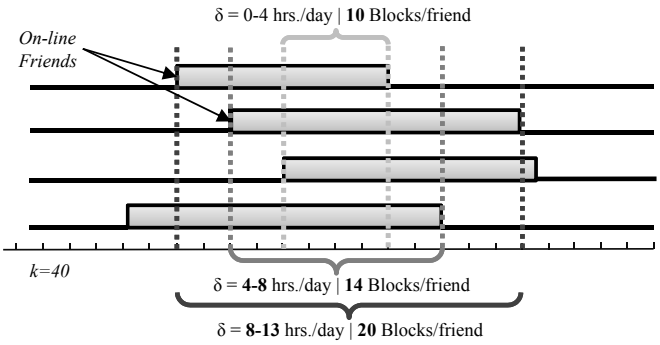



**Figure 6.13:** Relationship between daily data availability ($\delta$) and the amount of redundancy (blocks/friend). Clearly, this has a tremendous impact on the TTS.

availability traces we use in our tests are only *moderately* correlated, which very often results in an increase in the redundancy assigned to friends, so as *the time to schedule*. This is clearly visible in the "flattened" region of the curves for $F_C = 0.3$. Such an increase of redundancy is not *inversely linear* with the level of availability correlation, which is especially apparent in friendsets of small cardinality. In fact, we observed that this flattened region disappears completely in friendset of 30 or more members.

We illustrate the relationship between *data availability* and *redundancy* at storage nodes in Fig. 6.13. By simple inspection of this figure, it is easy to see that a targeted daily data availability of 8 hours requires much lower redundancy than a little higher data availability of 9 hours.

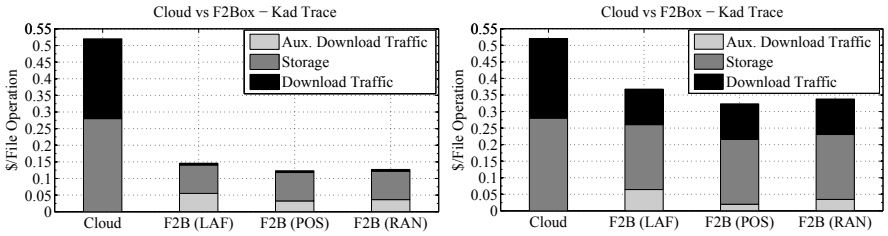
The download scheduling times are close to the MTTS as shown in Fig. 6.12c and Fig. 6.12d. Even in the worst case, between 65% to 90% of downloads are completed in less than 8 hours when the MTTS is of 4,85 hours. It is interesting to note that assigning more redundancy to friends contributes definitely to the smaller download TTS compared with upload times. On the other hand, the *relatively* long tail of the download TTS distribution, particularly for Kad, suggests that uploading more blocks to the cloud significantly reduces the chances to exceed greatly the average download TTS. This was expected since a user can always resort to the cloud when their friends are unavailable or present a high GDM.

**Data availability**. In Table 6.5, the data availability obtained by our system is reported for different values of $F_C$.

First, we observe that our redundancy calculation algorithm is very accurate in the presence of availability correlation. This means that past availability information is suitable to

**Table 6.5:** Data Availability

| $\delta = 8$ hours/day | Kad ($\lvert\mathcal{F}\rvert = 10$) | | | Skype ($\lvert\mathcal{F}\rvert = 10$) | | |
|---|---|---|---|---|---|---|
| *Percentile* | *0.25* | *0.5* | *0.75* | *0.25* | *0.5* | *0.75* |
| $F_C = 0.3$ | 5.09 | 6.67 | 8.16 | 6.59 | 8.10 | 9.40 |
| $F_C = 0.7$ | 4.76 | 6.49 | 7.97 | 6.52 | 7.96 | 9.37 |
| $\delta = 12$ hours/day | Kad ($\lvert\mathcal{F}\rvert = 10$) | | | Skype ($\lvert\mathcal{F}\rvert = 10$) | | |
| *Percentile* | *0.25* | *0.5* | *0.75* | *0.25* | *0.5* | *0.75* |
| $F_C = 0.3$ | 8.84 | 10.54 | 11.89 | 10.28 | 11.59 | 12.77 |
| $F_C = 0.7$ | 8.88 | 10.52 | 11.89 | 10.29 | 11.55 | 12.77 |




**(a)** $F_C = 0.3$. **(b)** $F_C = 0.7$.

**Figure 6.14:** Comparison of monetary costs of schedules between Amazon S3 and F2Box for one month storage in Kad with $\lvert\mathcal{F}\rvert = 10$ and $\delta = 12$ hours/day.

assure $\delta$ hours of data data availability for small, correlated friendsets. Proof of that is that the *median* data availability is quite close to the targeted data availability $\delta$, particularly for Skype.

Further, we do not observe significant differences between the data availability obtained depending on the fraction of data stored in the cloud ($F_C$). This means that our algorithms performs well irrespective of the storage distribution between friends and the cloud.

Second, as $F_C$ grows, the amount of data redundancy needed to achieve a certain $\delta$ is clearly lower, as can be inferred from the upload times (figures 6.12a and 6.12b). Therefore, by making use of cloud storage, F2Box *enables achieving equal or higher levels of data availability than a F2F system with much less data redundancy*. This is key to limit scheduling times and improve the storage capacity of a F2F system.

**Cost-QoS trade-off of F2Box**. First, it is clear that there exists a trade-off between cloud monetary costs and upload scheduling times as shown in Fig. 6.14a and Fig. 6.14b. More specifically, if a user prefers to minimize the time spent in storing large files, it is generally better uploading more data to the cloud. This will reduce the number of blocks to be transferred to friends $n_{\text{friend}}$, thereby drastically reducing the upload time to schedule.

The *inverse* trade-off arises from downloads. Storing more data at friends reduces monetary costs because less fragments must be downloaded from the cloud. Therefore, if some file is retrieved very often, it would pay off to decrease $F_C$ and upload more data to friends.

Finally, it worth to mention that the specific F2F scheduling policy chosen as basis for *Bandwidth Maximizing Friend-to-Cloud* policy has little impact on monetary costs, so the use of the cloud as a temporary buffer contributes minimally to the total cost. This cost is termed *auxiliary download bandwidth* in Fig. 6.14. However, it must be noted that POS outperforms LAF and RAN, mainly for two reasons: i) The nodes in the traces are *autocorrelated* enough to exploit their previous behavior, which yields that nodes exhibit a quite regular participation over time; and ii) the transfer of blocks into the cloud *indirectly* increases the autocorrelation of the schedule (i.e., similarity with itself), as those blocks will always be uploaded in the specified timeslot by POS.

In conclusion, `F2Box` *is flexible enough to cover all user needs by trading daily data availability and scheduling times for monetary cost*. This opens the door to a real adoption of F2F systems thanks to the greater reliability of cloud storage.

## 6.7 Discussion and Conclusions

In this section, we discuss the main observations of our analysis of F2F storage systems, putting special emphasis on their QoS limitations that motivated us to propose a novel data management techniques and a hybrid storage architecture.

**Data availability and redundancy**. We observed that, in general, traditional approaches to estimate data availability are not suitable in a decentralized F2F system. The cause of their lack of accuracy is the assumption that online sessions are not correlated, which translates into an over-provisioning of redundancy during diurnal hours and into an under-provisioning during night hours. We showed that such a treatment of redundancy is inadequate, and strongly hinders the viability of a pure F2F solution. For this reason, we proposed a new notion of data availability, called *daily data availability*, and a history-based data availability estimation algorithm to accurately calculate the level of data redundancy in the presence of strong correlations. Moreover, contrary to conventional wisdom, we found that the presence of availability correlations offers a good trade-off between data availability and data redundancy.

**Load Balancing**. Load balancing becomes a critical aspect to evaluate the effectiveness of a data placement policy. In our analysis, we discovered that storing more data blocks at highly available nodes may achieve higher data availability requiring less data redundancy than a simple round-robin data placement. However, considering small groups of friends to store data, this type of placement makes highly available nodes to be overloaded, inducing poor

load balancing. Combined with our history-based data availability calculation, we demonstrated that a round-robin placement can be used to maintain load balancing and to obtain an adequate degree of data availability in a F2F scenario.

**Transfer Performance**. The correlated availabilities of nodes in a F2F system makes it necessary to differentiate between if a file is available at a certain instant (i.e., there are enough blocks stored at online nodes) and if it is retrievable in a reasonable amount of time. This is mainly due to the presence of nocturnal patterns, which force the interruption of transfers performed before nights until blocks stored at disconnected nodes become available again. Further, although we detected slight differences in the performance of various data transfer scheduling policies, there is not a "clear winner" and normally these policies perform significantly worse than the optimal time to schedule (OTTS) —this result agrees very well with the results obtained in previous works. We also found that in most cases a small number of blocks is responsible for large delays of file transfers.

From our analysis, we conclude that due to the characteristics of F2F systems, it is difficult to provide an acceptable storage QoS to end-users. However, although these problems seem hard to overcome in a pure decentralized setting, we believe that a wise involvement of a cloud storage service in a F2F system may improve many of these limitations. Therefore, we propose to resort to the cloud to provide realistic service guarantees.

Essentially, our hybrid architecture, called `F2Box`, aims at helping a purely decentralized F2F storage system in the following aspects:

- *Reducing upload times*: The cloud helps users to reduce upload times by temporarily buffering data blocks that should be stored at friends that are unavailable at the moment of the upload.

- *Reducing download times*: The cloud improves download times of users by providing the missing data blocks in the cases where a group of friends cannot serve enough blocks to reconstruct the original file.

- *Limiting the amount of data redundancy*: To achieve a certain degree of data availability, storing data blocks of a file across friends and the cloud requires less data redundancy that storing that file only at friends. Naturally, this is due to the superior availability of the cloud that avoids generating additional redundancy for the blocks stored on it.

- *Users keep the control of their data*: In `F2Box` users are able to decide up to which extent a cloud is involved in their storage service. In turn, this decision is also reflected in the *cost-storage QoS trade-off* that our architecture provides.

In our view, `F2Box` improves critical aspects of the storage QoS of purely decentralized F2F systems, which may represent a significant step towards the adoption of these systems.

**Conclusions**. In this Chapter, we illustrated that F2F systems have specific characteristics (reduced friendsets, availability correlations) which need a special treatment. In this sense, we explored the storage QoS of F2F storage system analyzing severals aspects that are fundamental for their adoption, such as data availability, load balancing and transfer performance.

We evaluated traditional data management techniques used in large-scale systems (e.g., data availability, redundancy) and we concluded that they are not suitable in a F2F scenario. To solve this problem, we proposed to use historical information on the availability of friends to accurately calculate data redundancy. In our simulations, our technique obtained significant improvements compared with traditional redundancy calculation approaches.

Finally, our analysis also showed that it is difficult to provide an adequate QoS in a pure F2F system due to the presence of availability correlations and small groups of friends. To retain the main advantages of F2F systems, we have proposed a hybrid architecture that takes advantage of the superior availability of cloud storage services to improve their QoS. To this aim, we have developed novel scheduling strategies, and a new algorithm that let users adjust redundancy according to the availability correlation exhibited by friends. Our results certify the benefits of combining the best of both worlds.

# 7

# Empirical Analysis of Social Cloud Storage

**Summary**

Digital relationships between individuals are becoming capital for turning to one another for communication and collaboration and create new opportunities to define socially oriented computing models. In this Chapter, we propose to leverage these relationships to form a dynamic "social cloud" for storage. While at first glance, the concept of social cloud looks very appealing, a deeper analysis brings out many problems in terms of storage QoS due to its decentralized nature. To overcome this problem, in addition to digital friends, we propose to the members of the social cloud the use of online storage services like Amazon S3 to store data and improve the service performance. Through a real deployment in our campus, we analyze the role that factors like the social network graph play on storage QoS to determine the feasibility of the social cloud as storage media.

*The papers with the results of this chapter appeared in [25, 28, 31].*

## 7.1 Introduction

The "social cloud" facilitates resource sharing by utilizing the relationships established between members of a social network [22, 23]. Therefore, compared with cloud storage, the information is made only available to *trustable* members of the social network, thus significantly reducing the risk that personal data might be sold on, and without raising suspicions about how commercial storage services are monetized.

However, despite the increasing popularity of this computing paradigm, the social cloud also carries important deficiencies due to its decentralized nature. The most critical one is that, contrary to commercial clouds, it is not feasible to establish a formal Service Level Agreement (SLA) within a social cloud system. Its operational feasibility is based on the premise that participants are socially motivated and subject to the personal repercussions outside the functional scope of the social cloud. This is primarily due to the existing level of trust that already exists between members. In this context, SLAs or "contracts" should be viewed as a best effort agreement between the social links. This weaker form of agreement translates into a limited availability of resources and capabilities. Although a social cloud system is built upon social incentives, peer pressure, etc., the discontinuous participation of social contacts, or even the abandon of the social cloud, is intrinsic to the nature of social relationships.

In terms of storage, this means that the data stored within the social cloud may be subject to recurrent periods of unavailability. In a social cloud, the percentage of time that data is available is a function of the number of friends contributing their storage space over time. And such a dependence has deep implications for the correct operation of a social cloud, mainly in terms of data availability, understood as the probability to access a data item when needed.

First, as pointed out in Chapter 6, while there may be a sizable number of individuals in a social network, typically only an insignificant number can be utilized as a destination for personal data. To inform this argument, over 63% of Facebook users have less than 100 friends, and the majority of social interactions occur only across a small subset of them. More specifically, it has recently been observed that only 20% of the social links capitalize 70% of all social interactions [65]. This means that in practice the number of users willing to contribute their storage resources to sustain the social cloud will be small. If in addition to this we add the problem of the temporal correlation in the connection habits of users, the loss of data availability is inevitable. Real measurements from online social networks have detected the presence of strong daily and weekly interaction patterns [99, 100]. Very succinctly, this means that the probability of finding simultaneously offline all the social links of a user is high, particularly

during night hours, which makes it impossible to maintain data availability even under full replication where a replica is allocated to every member of the social cloud.

Second, the topology of the social network graph plays a central role. As such, it delineates the interaction events that may occur across social links and hence, the amount of resources to be contributed by a member. Although users with many friends have a greater opportunity to store their data with higher availability, they may possibly have to donate more disk space to reciprocate a larger number of friends. Real measurements of social networks [64, 65] show that while clustering is very high, the existence of a few users with a large number of friends is characteristic of social interaction. For these users with abnormally high degrees, usually called *hubs* in the graph literature, the contribution of their storage resources may be high for little or no personal gain. In this sense, poor storage fairness may motivate the need for economic or non-economic mechanisms to regulate sharing within a social cloud.

Overall, understanding these factors is a necessary step in determining whether the vision of social cloud is realizable, and therefore, it can really emerge as an alternative to commercial cloud providers. In this Chapter, we aim to answer questions like: *"What is the role of social graph in the obligation to trade storage space? Is there any significant asymmetry in the level of contribution by users such that an altruistic model is infeasible? Is the availability of a user indicative of its real contribution level to the social cloud?"* Questions that have not been raised in the existing literature. We believe that answering these questions is vital to appraise to what extent the social cloud can emerge as true alternative to existing commercial and non-profit storage systems.

In summary, the main contributions of this Chapter are the following:

- We contribute to the state of the art by quantifying the influence of the above factors, putting special emphasis on the topological effects, while outlining some of the challenges to make the concept of social cloud storage a reality.

- To conduct this study, we have instantiated this model into `Friendbox` [25], a social cloud storage application embedded into Facebook. A distinctive feature of `Friendbox` is that lets a user add an external cloud storage service like Amazon S3 to its social cloud in order to improve the availability of its data.

- Through a real deployment in our campus, we spot evidence of the bearing of factors like the social topology on the definition of social cloud storage. The fact that our results has been obtained through experimentation gives the additional advantage of measuring the real impact that these factors and design choices may have on performance and cost.

The remainder of this Chapter is structured as follows. In Section 7.2 we describe `FriendBox`, our social storage application to conduct our experimental assessment, which is included in Section 7.3. The results of our empirical analysis appear in Section 7.4. Finally, we provide some discussion about our empirical insights and our conclusions in Section 7.5.

## 7.2 Social Storage with `FriendBox`

To give form to the definition of "social cloud storage" and determine what aspects should be integrated into its definition, we have employed our social cloud storage application, called `FriendBox` [25], which has been developed and deployed as a Facebook application. We chose Facebook for its popularity, development environment and API, and very importantly, because Facebook identification allows users to define policies regarding who can store and access their personal data. For example, a user could limit the sharing of their data with close friends only, or users in the same group. This gives individuals high control over their data, engendering trust and some level of accountability, properties that are hard to find in a cloud environment. From a privacy standpoint, while Facebook learns the interactions between the members of the social cloud, personal information is never revealed to this online service, as it is stored and shared through peer-to-peer exchanges.

A distinctive feature of `FriendBox` is that lets a user add an external cloud storage service like Amazon S3 to its social cloud in order to improve the availability of its data. By no means this signifies that all data will be stored in the cloud. Following the spirit of the social cloud approximation, `FriendBox` lets the user decide the amount of data to be stored in the cloud, which can be zero if the user wishes so. This feature is particularly useful, as it allows to trade data availability for monetary costs and adapt the storage service to the user needs. In fact, `FriendBox` is based on the design of `F2Box` and our insights in Chapter 6.

Further, the use of the cloud requires another layer of preprocessing the data in order to protect it from unauthorized access, disclosure and theft. This could be accomplished in many ways. A simplistic approach could be to encrypt each sensitive piece of data and share the key with the authorized users. Instead of this simple encryption scheme, we use Reed-Solomon codes [69] for that purpose, blending storage efficiency [68] and privacy in a single scheme. Other approaches would be equally possible with no significant changes in the proposed method. However, we do not want to involve ourselves in this question here, since our focus is on analyzing the feasibility of this new storage model.

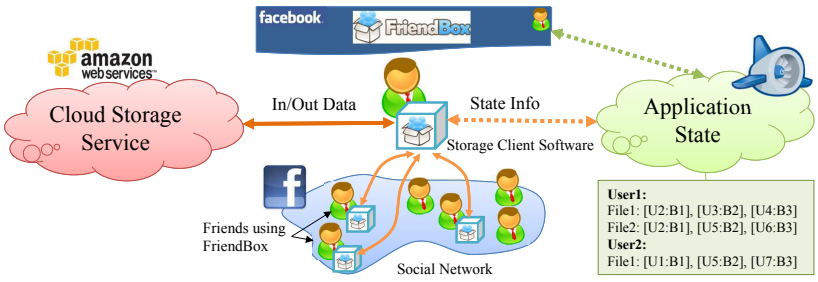



**Figure 7.1:** A user maintains storage links with some of his friends in Facebook. Moreover, this user is able to store a fraction of his data in a cloud storage service. The state information of a user's data is stored in the `FriendBox` Application State. Finally, users manage their storage relationships and check the state of their storage service in the `FriendBox` Social Front-end.

In what follows, we will describe the components of `FriendBox`, whose general architecture is illustrated in Fig. 7.1. We will give the essential details to make our results understandable and refer the reader to [25] for full details[1].

### 7.2.1 Social front-end: Facebook Application

In our social cloud, the storage overlay is bootstrapped by the underlying social structure. Accordingly, every node in the friendship graph acts as a storage service to their adjacent neighbors. In practice, the friendship graph can include members of the family, close friends only, or even friends of friends, which can be viewed as directly connected to each user that selects them as storage servers.

As social substrate, `FriendBox` uses Facebook for user management, because Facebook exposes access to their social graph through a simple API, called the Graph API[2]. This API exposed through a REST service gives access to many objects, including friends, profile information, groups and photos. To control access to the Graph API, Facebook utilizes the OAuth protocol [51] to authenticate both users and applications. This authorization model allowed `FriendBox` to delegate access control to Facebook, simplifying considerably user management and accountability.

The integration of `FriendBox` with the Facebook look and feel was by means of the Facebook Markup Language (FBML). FBML includes a subset of HTML with proprietary extensions that enables the creation of applications that follow the Facebook style. Code written in FBML is retrieved by the Facebook server, parsed, and then inserted into their surrounding

[1] `FriendBox` webpage: http://ast-deim.urv.cat/friendbox/
[2] http://developers.facebook.com/docs/reference/api/.

code. This facilitated the creation of a familiar and intuitive GUI for `FriendBox`. Through this GUI, the user can keep track of its monthly storage consumption in the cloud provider of its choice and the distribution of its data within the social cloud, among other operations. Such state information is maintained in a separate component called Application State, which we discuss in the following section.

### 7.2.2 Application State

Essentially, the Application State maintains up to date the data management information about any file stored in the system. This information includes the specific set of friends that store each data object along with the network address of each one. Without this information, the clients would be unable to perform the necessary peer-to-peer storage operations to store and retrieve any data file from the social cloud. The logic of keeping the Application State current lies on the desktop clients themselves. The clients update the Application State via a REST API.

The role of the Application State is depicted in Fig. 7.1. In this figure, we show how a user communicates with the 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. As shown in the figure, Application State stores this information using mappings that relate data blocks with the friends who are responsible for them.

The Facebook application code for `FriendBox` along with the Application State is nowadays hosted in Google App Engine[1]. The reason for this choice was that this PaaS for developing web applications offers elasticity in the service. Note that if we wanted to protect the metadata from possible threats such as theft, unauthorized access, copying, etc., an additional layer of protection would be indeed necessary. One simple way of doing this would be to encrypt the metadata before storing it in Google App Engine. This issue is, however, beyond the scope of this thesis.

### 7.2.3 Desktop Client

In addition to the integration with Facebook, a social cloud storage application needs a desktop client to store and access remote data. To efficiently achieve the desired level of data availability, `FriendBox` lets users select the set of friends to where store each content and decide which part of the data should go to the cloud. In the current version of `FriendBox`, the desktop client

[1] http://code.google.com/intl/en/appengine/.

only permits to store data in Amazon S3, though other cloud storage services like `Windows Azure` and `Google Drive` can be easily supported.

To achieve high availability, the best strategy would be to store all data in the cloud to guarantee 24/7/365 access availability. However, at $0.120 per GB of data transferred out of the cloud, these costs might quickly add up. To decrease monetary costs, `FriendBox` uses the friends in the social cloud to store data but at the expense of a lower data availability. The fundamental idea behind `FriendBox` is to provide data availability during the hours of the day where friends are mostly logged in to benefit from availability correlations. We introduced this new notion of data availability, termed *daily data availability* in Chapter 6, for we refer the reader to for further details. Going back to our formulation in Section 2.2.2, a user may want to achieve a daily data availability of $\delta$ time units for its data. By viewing daily data availability $D$ as a subset of $T_{day}$, i.e., the set including all the time units of one day according to a particular quanta, $D$ contains those time units of $T_{day}$ being covered by at least one friend, and preferably those with a greater number of friends. The reason is that a greater number of friends supplies more flexibility to allocate data for load balancing.

### 7.2.4 Data Redundancy and Privacy

In Chapter 2 we explained the advantages of Reed-Solomon (RS$(n,k)$) codes and our data redundancy model. Thus, we are now ready to discuss how we distribute the encoded data objects across the social ties and the cloud service. Concretely, after applying the RS coding scheme, a fraction $F_C$ of the original $k$ fragments is allocated to the cloud. Recalling that $n = k+h$, the remaining $\lceil (1-F_C)\cdot k \rceil + h$ blocks are allocated to the social friends in a round robin fashion to achieve an even use of their disk capacity. Compared with replication, one of the most valuable assets of RS codes is that the amount of data assigned to a friend is typically only a fraction of the original file size, saving significant storage space.

It is important to mention here that the exact value for $F_C$ depends on the parameter $\delta$ and the connection pattern of the friends in the social cloud. For instance, let us consider we want to cover $\delta$ time units of data availability. Depending on the number of online friends at each of these time units, the storage requirements and the appropriate value for $F_C$ will vary. To illustrate this, we consider two extreme cases. At one extreme lies the case where one of the $\delta$ time units is covered by a single friend. In this case, in order to ensure the reconstruction of the object, this single friend will be forced to store at least $k - \lfloor F_C \cdot k \rfloor$ out of the $n$ blocks. And here the chosen value for $F_C$ makes a big difference. The reason is that the value of $F_C$

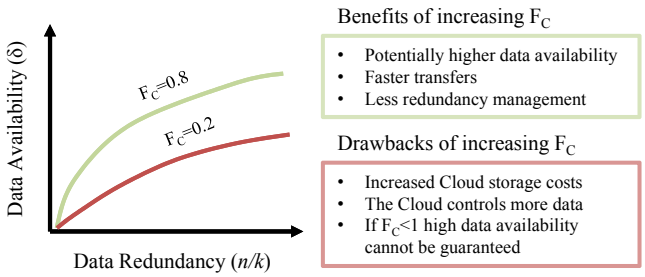



**Figure 7.2:** Implicit trade-offs between data availability, redundancy and cloud costs in `FriendBox`.

determines the storage requirements for this friend, which will be maximal and lead to the storage of a complete replica of the data file for $F_C = 0$. At the other extreme is the case that all $\delta$ time units are covered by at least $k - 1$ friends, requiring to store only one block in the cloud. In this case, however, a small value of $F_C$ will be not so problematic, because the storage capacity contributed by each friend will be significantly smaller: Just one block. Hence, a high level of correlation in availability patterns can help to reduce the fraction $F_C$ for a fixed $\delta$.

Another important advantage of Reed-Solomon codes is that the generator matrix of the code can be chosen to be non-systematic. If a code is non-systematic, then the original data fragments will not appear in the code, preserving data confidentiality. Note that this statement is valid provided that no subset of blocks of cardinality greater than $k - 1$ is in the hands of a non-authorized party —for instance, a cloud storage provider. We must note, however, that while a non-systematic RS code is a $(k, n)$ threshold scheme, and can be interpreted in terms of Shamir's secret sharing [152], its security guarantees are less than Shamir. The reason is the lack of randomness in the generator matrix of RS codes. So, attackers looking for known or patterned data can find it more easily without reconstructing the original data [153]. In `FriendBox`, this level of protection is sufficient. A higher protection level can be simply achieved by first encrypting the data and then encoding it, or by using more elaborated dispersal schemes such as AONT-RS [153]. Since all of these variants also transform a file into $n$ distinct blocks, our analysis is equally valid for all of them.

For completenesses, Fig. 7.2 illustrates the relationship between data availability $\delta$, the redundancy ratio $n/k$, and the fraction of data allocated to the cloud $F_C$, which are the three parameters of our storage model. Let us first consider that the redundancy ratio $n/k$ is kept fixed. In that case, the result of increasing $F_C$ by pushing more blocks to the cloud is that data availability increases. This suggests that by choosing the right $F_C$, one can achieve the same data availability with less redundancy. Consequently, a user will experience shorter

transfer times and he will require less resources from his friends. However, increasing $F_C$ may present some drawbacks, specially related with a higher cost of the storage service and the amount of data control relinquished to the cloud operator. Furthermore, even in the case of storing $k-1$ blocks in the cloud, 100% availability cannot be guaranteed: *If all storage friends are simultaneously unavailable, the missing block will not be reachable* [27]. `FriendBox` gives to the user the opportunity to decide the most adequate storage service depending on his needs.

### 7.2.5 Data Transfer

As introduced in Chapter 6, once a file has been encoded, it is necessary to transfer the encoded blocks to the corresponding social ties and to the cloud storage service. In `FriendBox`, we differentiate between two distinct types of transfer scheduling policies: *Friend-to-friend transfer scheduling policies*, which select the blocks that should be transferred to storage friends in a certain order based on a criterion, and (ii) *friend-to-cloud transfer scheduling policies* that decide if blocks should be transferred first from the cloud or friends.

First, we describe the *friend-to-cloud* transfer scheduling policies implemented in `FriendBox`. To minimize transfer time and fully utilize the upstream bandwidth, `FriendBox` uses the cloud storage service as a *temporary repository* to store the blocks for those social links that were offline when the transfer of their blocks was scheduled. In any case, the extra blocks pushed to the cloud are downloaded afterwards by the friends to whom they were initially allocated.

For downloading a file, `FriendBox` prioritizes the download of the corresponding blocks from friends to incur the minimal monetary costs due to the data transfers out of the cloud. Only in the case that there are less than $k$ blocks, the remaining up to $k$ are downloaded from the cloud storage service.

Regarding *friend-to-friend* transfer scheduling policies, to simplify the development, we implemented in `FriendBox` a *random* transfer scheduling policy. That is, given a set of data block assigned to a group of available nodes, `FriendBox` selects at random the order in which block transfer will occur. This applies to both uploads and downloads.

## 7.3 Evaluation Framework

In this section, we empirically study *the fundamental problems and challenges involved in the social cloud storage paradigm*. Indeed, what the incipient social cloud literature misses is a deep analysis of the implications that environmental factors such as user availability and topology

have on the storage service. As a central contribution of this work, we identify and quantify the main underlying factors that influence the storage service provided by a social cloud.

## Objectives and metrics

Through experimentation, we aim to shed some light on the following aspects that we believe capital to provide an adequate storage service in a social cloud:

- **Daily Data Availability**: The probability to access a data object during the day, which depends on parameters such as the amount of redundancy $\frac{n}{k}$ and the fraction of the data allocated to Amazon S3. Of course, correlation in availabilities plays a key role on the achievable daily data availability.

- **Clustering Coefficient**: In addition to the graph degree, we make use of the clustering coefficient ($CC$) to measure to what extent the social links in the friendship graph tend to cluster together. The local $CC$ of a user $v$ is defined as:

$$CC_v = \frac{2 \cdot E_v}{D_v(D_v - 1)}, \tag{7.1}$$

  where $E_v$ is the number of edges between neighbors of $v$ and $D_v$ is the degree of user $v$. Loosely speaking, the $CC_v$ quantities to what extent the neighbors of $v$ are linked to one another. In our tests, we will mainly use this metric to study the contribution level of hubs.

- **Load Balancing**: Load balancing is critical to the feasibility of a distributed storage system [27]. For this reason, we analyze the interplay of the social graph topology and user availability on the resulting storage load supported by users.

  We quantify load balancing in two ways. At the user level, we account for the number of storage operations processed by each user, i.e., data block `PUT`s and `GET`s. At the global level, we utilize the Gini coefficient and the Lorentz curve to examine the distribution of served storage operations in the social graph. Specifically, the Lorenz curve depicts the proportion of the total income of the population ($y$ axis) that is cumulatively earned by the bottom $x$% of the population[1]. The diagonal line represents perfect equality of incomes. The Gini coefficient, denoted by $G$, is the ratio of the area that lies between the line of equality and the Lorenz curve ($A$) over the total area under the line of equality ($A + B$):

$$G = A/(A + B). \tag{7.2}$$

[1] For a technical description of Gini coefficient and Lorentz curve see http://en.wikipedia.org/wiki/Gini_coefficient.

- **Transfer Time**: An important performance metric for social cloud storage is data transfer speed. In particular, we study two aspects: the congestion caused by the topology of the social network and the impact of correlated user availabilities on the time to download a file from the system.

- **Fairness**: Typically, a social cloud adds regulatory protocols to enforce resource fairness. However, there is no analysis on the extent of the potential asymmetry that may arise in a social cloud along with what elements may originate it. As a simple measure of fairness, we utilize the ratio between the amount of resources contributed to the social cloud and those consumed by a user:

$$FR = \frac{R_p}{R_c}, \tag{7.3}$$

  where $R_p$ represents the amount of resources a user provides to the system, and $R_c$ the amount of resources that a user consumes from his social ties. A value of $FR$ equals to 1 represents perfect equilibrium between resource consumption and contribution. $FR > 1$, however, means that a user is contributing more resources to the system than what is actually consuming. Finally, $FR < 1$ signals that a user may be abusing its social ties, because it consumes more than it donates.

- **Cloud Contribution**: As we use cloud storage, i.e. Amazon S3, as a pivotal element to the feasibility of a storage service in a social cloud [25], its role in the system deserves special attention. Indeed, we measure the consumption of cloud resources that the members of the social cloud incur in their `PUT` and `GET` storage operations, depending on their availability and position in the social graph. We use the number of data blocks transferred in and out of the cloud because this simple metric can be immediately turned into monetary metrics like the "dollars per storage operation".

## Scenario and Setup

Once elaborated on the objectives of our evaluation, we are ready to describe the setup of our experiments.

**Topology**. We deployed a group of 20 `FriendBox` desktop clients in our university laboratories. The 20 `FriendBox` clients were organized according to two real graphs from Friendster [154] in order to assess the influence of the friendship topology. One topology shows a high clustering or local transitivity, i.e., if user $a$ knows $b$ and $c$, then $b$ and $c$ are likely to know

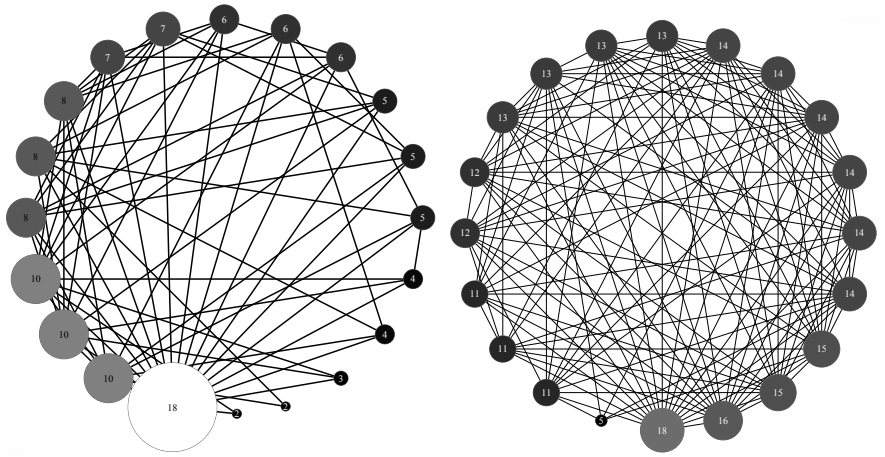


**Figure 7.3:** Input social graphs for our experiments. The graph on the left exhibits a low average clustering coefficient of $CC = 0.3$, whereas the $CC$ of the graph on the right is 0.7. Node labels correspond to their degree.

each other, while the other is weakly clustered. To identify each topology, we will use the value of the clustering coefficient at the hub. Both topologies are illustrated in Fig. 7.3. Accordingly, their degree distributions are shown in Fig. 7.4 (right).

**Availability**. To incorporate availability correlations into our experiments, we instrumented the alternating ON-OFF behavior of users by means of an availability trace from Skype [103], which exhibits strong diurnal patterns and high heterogeneity in user availabilities. Both properties are clearly visible in Fig. 7.4 (left). The CDF of user availabilities ranges from 0.18 to 0.75, which evidences high heterogeneity. Furthermore, the time-series representation in the inner plot illustrates that friends are mostly connected during the central part of the day and disconnected during night hours.

To study the impact of availability in the social hub, we assigned two different availabilities to the highest degree user in the social graph: A high availability of 0.594 and a low availability of 0.278.

We also conducted simulations where users were always online as baseline to understand the effect of availability correlations. We will refer to this scenario as "no churn" throughout the evaluation.

**Workload**. The workload model of our experiments is homogeneous. All nodes alternatively perform file downloads and uploads while being logged in. Hence, file transfers are concurrently executed throughout the experiment to capture the effects of network congestion. File transfers are randomly performed every period of $[600\text{-}1,200]$ seconds over synthetic files

| Parameter Description and Values | |
|---|---|
| Nodes in the system | 20 |
| Experiment duration | 24 hours |
| Node storage capacity | 40 GB |
| Parallel upload/download connections | 2, 2 |
| Erasure codes original file fragments ($k$) | 40 |
| Cloud file fraction ($F_C$) | 0.5 |
| Object size ($\beta$) | 400 MB |
| Data redundancy ($n/k$) | 2.0 |
| Cloud back-end | Amazon S3 |

**Table 7.1:** Parameter configuration in our experimental scenario.

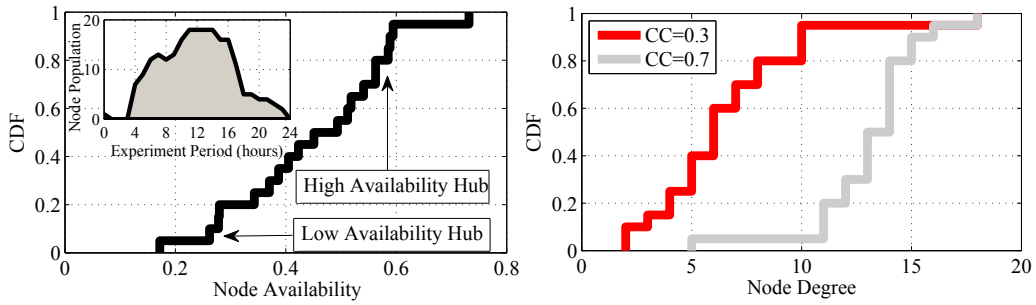



**Figure 7.4:** Nodes present high availability heterogeneity and diurnal patterns (left). The node degree distribution varies significantly depending on the $CC$ (right).

of size $\beta = 400$MB. Unless otherwise stated, we fixed $F_C = 0.5$ and the redundancy ratio to $\frac{n}{k} = 2$.

**Hardware**. `FriendBox` clients were hosted in desktop computers (Intel Core2 Duo and AMD Athlon X2 processors) equipped with 4GB DDR2 RAM. The OS was Debian Linux[1]. The clients were connected via a 100 Mbps switched Ethernet links. For the collection of physical network information, we utilized `vnstat`, a tool that keeps a log of network traffic for a selected interface. The rest of information presented in this section was gathered by the `FriendBox` log system. Other important parameters in this experimental scenario are depicted in Table 7.1.

## 7.4 Experimental Results

Here we present the experimental results and describe the main insights that follow from our analysis of the social cloud storage.

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

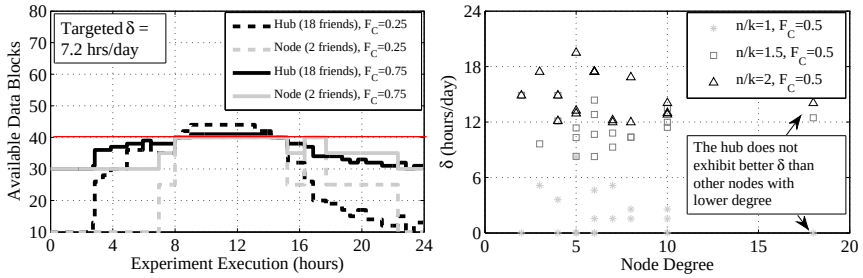



**Figure 7.5:** Time series plot of the available blocks for the hub and the least connected node to achieve $\delta = 7.2$ hrs./day (left). Impact of increasing $n/k$ on $\delta$ depending on the node degree (right).

### 7.4.1 Data Availability

In this section, we study the factors that influence the daily data availability. For this reason, we fix the target daily data availability $\delta$ to 7.2 hours and vary the fraction $F_C$ of data to be allocated to the cloud. For clarity, we only report the results for the topology with small clustering. Also, we only consider the case where the social hub is highly available.

The effect that availability correlation induces on daily data availability can be clearly seen in Fig. 7.5 (left). Surprisingly, the least connected user achieves the target 7.2 hours of data availability by making use of less redundancy than the social hub. This can be easily inferred by tracking over time the number of data blocks available for each user. The cause of this counterintuitive behavior is availability correlation: The two friends of the least connected user are simultaneously online for $\approx 8.5$ hours. Because they cover by far the target 7.2 hours of daily data availability, no extra redundancy is necessary. In general, however, it is difficult to have a sufficient number of online friends for $\delta$ hours, which requires the introduction of extra redundancy to meet the target level of data availability.

Further, Fig. 7.5 (left) gives an interesting result: The allocation of a larger proportion of data to the cloud makes it possible to achieve the target 7.2 hours of data availability with less redundancy. This is because a larger $F_C$ reduces the number of data blocks to be given to friends. Since friendsets exhibit poor availability compared with Amazon S3, the necessary redundancy to meet a certain $\delta$ may become smaller. This occurs to the social hub whose redundancy ratio $\frac{n}{k}$ decreases by a 14% when increasing $F_C$ from 0.25 to 0.75. These savings become more significant for higher $\delta$s.

The dispersion graph in Fig. 7.5 (right) relates the number of social links ($x$ axis) with the achievable $\delta$ ($y$ axis) for different amounts of redundancy $\frac{n}{k}$. As expected, the higher the redundancy is, the higher the data availability is. However, the increase in data availability is

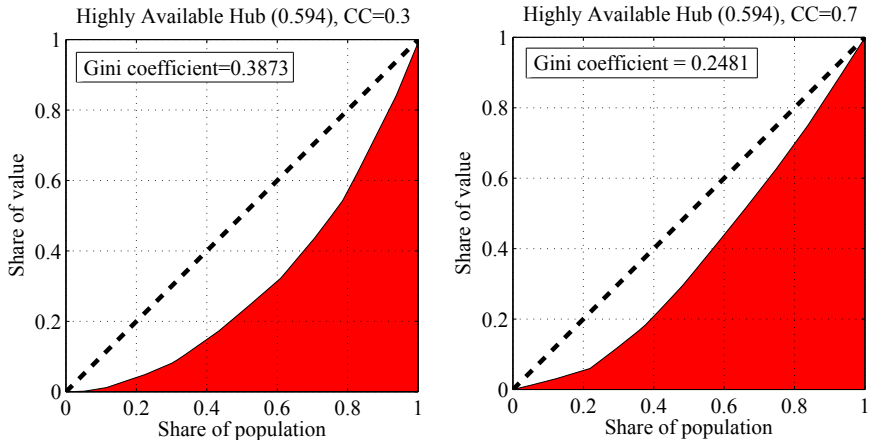



**Figure 7.6:** Distribution of served download block requests (`GETs`) in a churn scenario depending on *CC*.

not linear and may be abrupt or even zero for a higher $\frac{n}{k}$. Concretely, the final data availability depends more on the availability pattern of users than on the number of friendship links a user has. This is evidenced by the lack of correlation between the node degree and the achievable $\delta$. In fact, some users with a smaller number of friends present a higher $\delta$ than those users with a larger friendset.

We can summarize the main findings of this section as follows: (i) A larger number of friends *helps but does not necessarily improve* daily data availability; (ii) The *degree of coincidence in the online periods* of friends is crucial to understand the relationship between data availability and redundancy; (iii) Storing a fraction of data in the cloud may *reduce the overall redundancy* needed in a social cloud system.

### 7.4.2 Load as a Function of Social Graph Topology

Here we examine the influence of the graph topology on the load experienced by users. In Fig. 7.7, we report the number of data blocks that a user stored (`PUT`) and served (`GET`) as a function of its degree. The figure contains four subplots, each of which corresponds to a distinct combination of topology and availability model. Interestingly, all four dispersion graphs show that the load of users varies significantly depending on the clustering of the social graph topology. For high clustering, load is more evenly spread across all users, irrespective of the availability model.

For low clustering topologies, however, the degree strongly determines the load of a user. This conclusion comes from the visible linear growth on the number of storage operations with

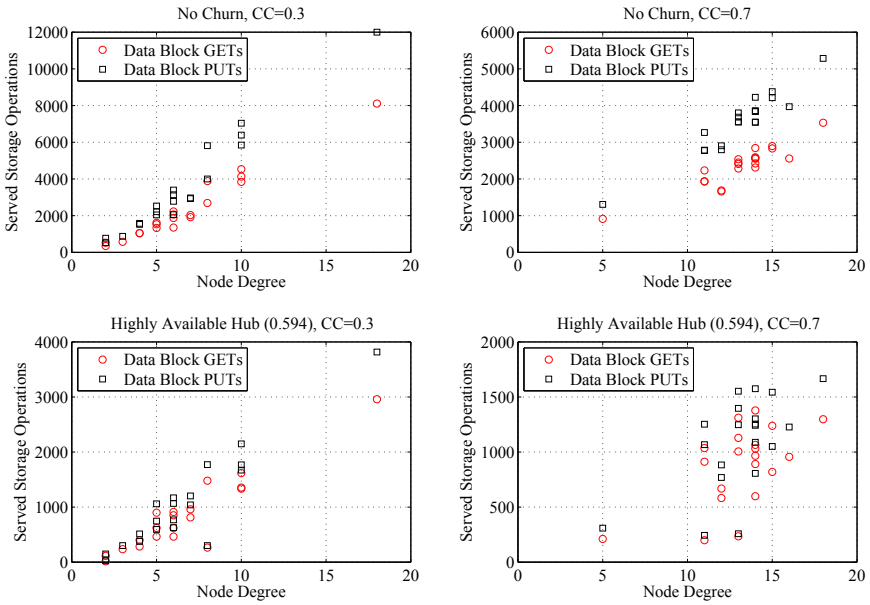



**Figure 7.7:** Relationship between a node's degree and the storage load caused by its friends. We illustrate a churn scenario (high available hub) and a stable scenario.

increasing user degree. Such a behavior may compromise the scalability of a social cloud. Social hubs, which interact with most of their social links [65], may become eventually saturated, and socially-based incentives may be even insufficient to enforce cooperation in the social cloud. This may pose the need for more sophisticated trading and sharing strategies like auctions and formal SLAs.

To examine load balancing from a global view, we calculate the Gini coefficient to measure the inequality in serving `GET` operations. The corresponding Lorentz curves are shown in Fig. 7.6. As shown in this figure, the Gini coefficient is much smaller and the Lorenz curve much closer to the diagonal in the topology with high clustering, which indicates that a higher connectivity facilitates the balancing of load among the members of the social cloud. But more importantly, and contrary to conventional wisdom, there exists no correlation between the load and the user degree in the presence of availability correlations. This phenomenon can be easily seen in the lower right subplot of Fig. 7.7, where users of similar degree present very disparate loads. We explore this issue in the next section.

We summarize the main results of this section as follows: (i) For low clustering, the *degree strongly determines the load* of a user; (ii) In general, a *high clustering* coefficient results in a *better load balancing* within the social cloud.

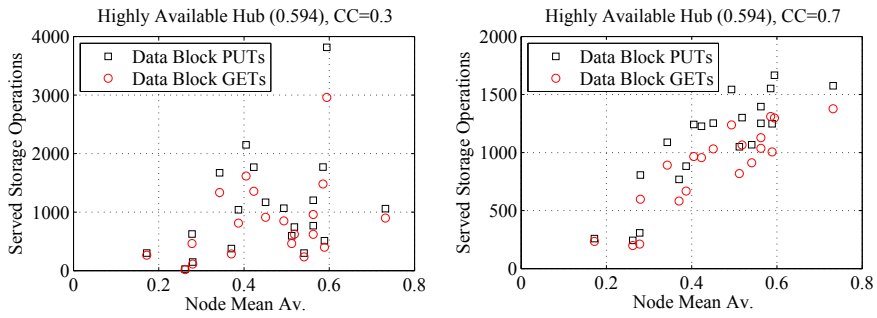



**Figure 7.8:** Relationship between storage load and node availability depending on the clustering coefficient.

### 7.4.3 Load as a Function of User Availability

Let us now consider the traffic load a user encounters as a function of its availability. The dispersion graphs in Fig. 7.8 relate these metrics for both stored and served blocks in a dynamic scenario with different clustering values.

The first main observation is that user availability does not positively correlate with storage load when the degree of clustering is low. This result is important because conventional wisdom assumes that high user availability is synonym of a higher burden. However, we observe that load in a social cloud system depends on other factors like the specific topology of the social graph. Concretely, we find that for low clustering, the number of friends that a user has is what determines its storage load.

On the contrary, when the social graph is highly interconnected, availability is what mainly determines the storage load experienced by users. This conclusion is evidenced by the linear increase in the number of data block transfers with increasing user availability. This result is not surprising. In the ideal case that all the members of the social cloud were fully connected, the burden experienced by each individual would be proportional to its availability: The higher availability the greater the odds of undertaking a storage `PUT` and `GET` operation.

To summarize, in a social cloud with *high clustering*, the *availability of a user determines the load* it will receive.

### 7.4.4 Data Transfer Time

First, we assess transfer speed as a function of the social graph topology. To avoid any interference caused by availability correlations, Fig. 7.9 depicts the distribution of transfer time when all the users in the social cloud are online, i.e., when there is no churn. For clarity, we only plot

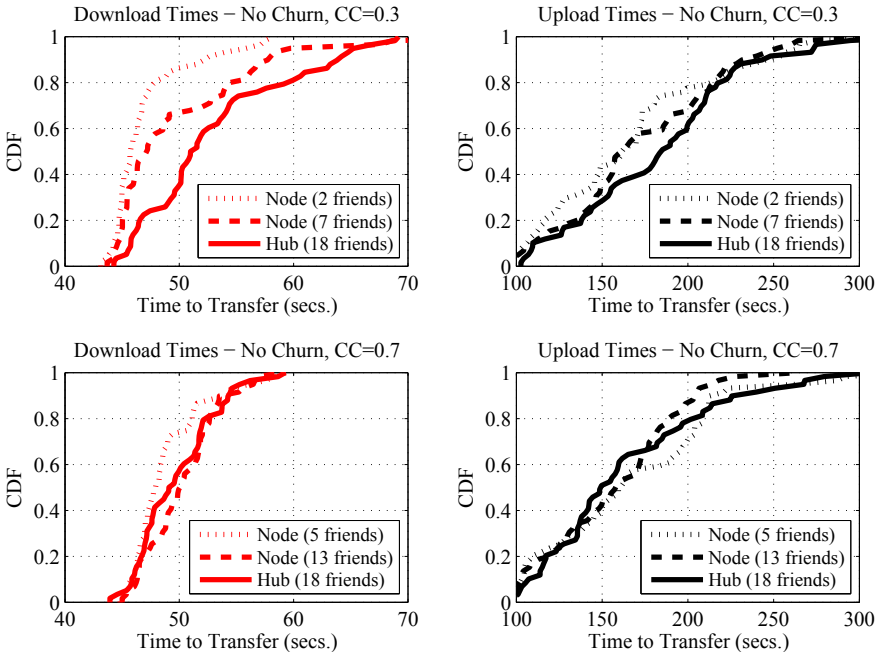



**Figure 7.9:** Effects of *CC* on transfer times and congestion.

the transfer time distribution for three users: the social hub who is linked to 18 friends, a user with the average network degree, and the least connected user in the social cloud.

For the low clustering topology, two observations are specially interesting. First, the least connected user achieves a lower transfer time than its higher degree friends, particularly for downloads. This is explained by the fact that for low clustering topologies, the users with many social links support a higher storage load and suffer from congestion. Second, the differences in the upload time are less significant. This is mainly due to two factors. First, local data block transfers among friends are much faster because of our Fast Ethernet LAN than accessing Amazon S3. Second, uploading in `FriendBox` involves the transfer of a fraction of the data to Amazon S3 while downloads retrieve as much as possible data from friends only accessing the cloud if there are not enough blocks available at friends.

For the high clustering topology, however, there are no important differences in file transfer times neither for uploads nor downloads. This means that a higher clustering coefficient introduces less congestion.

Now we study the effects of availability correlations on download times. For such a purpose, Fig. 7.10 plots the download time given as a time series for the social hub and one of the users whose degree coincides with the average degree of the social graph. For the social hub,

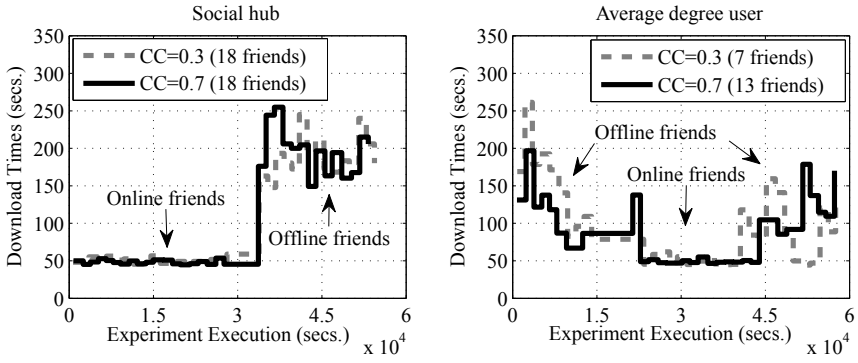



**Figure 7.10:** Time series analysis of download times of two different nodes. We clearly observe the consequences of availability correlations on download times.

Fig. 7.10 (left) reports that the download time is short when most of its friends are logged in. However, this time increases significantly during night hours. This is because the hub needs to resort to the cloud in order to complete the file download, which makes downloading to be slower in our campus scenario.

For the average-degree user, Fig. 7.10 (right) reports a larger download time than for the social hub, which indicates that the download time diminishes with the number of friends since blocks transfers from friends are faster than accessing the cloud. This is supported by the fact that for the same node, in most cases, a higher degree induces shorter download times.

Finally, it is worth mentioning that in some cases, specially at the end of the regular node execution, a few file downloads when that node has 13 friends are slower than when it has only 5 friends. As in the case of data availability, *a higher degree reduces download times if friends are simultaneously online at the moment of downloading the content*. Otherwise, a higher degree will have little or no positive effect for the storage service a node receives.

We summarize this section as follows: (i) For *low network clustering*, users with *high degrees exhibit larger transfer times* due to network congestion, which can be critical for hubs; (ii) A *higher clustering* coefficient inherently reduces congestion and *improves transfer times*; (iii) Although having more friends may in general improve download times, the actual number of *online friends when the download occurs* is fundamental.

### 7.4.5 Fairness

Now we study the resource fairness among the members of the social cloud. We use the fairness ratio ($FR$) as defined in Eq. 7.3 to measure the asymmetry in resource contribution. To

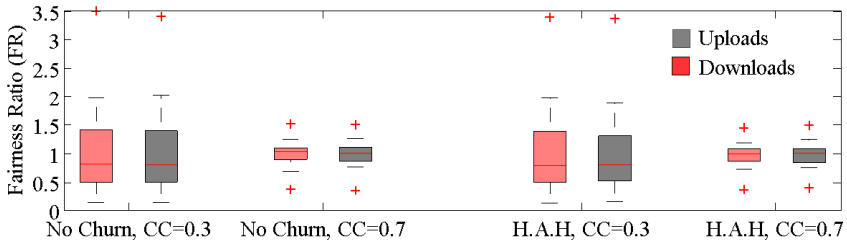



**Figure 7.11:** Fairness ratios of up./down. transfers depending on the network's *CC* for stable/churn scenarios.

start with, we focus on the fairness in bandwidth contribution. As a boxplot allows to assess the dispersion of a given distribution, Fig. 7.11 shows the boxplots of the distribution of fairness ratio when the resource under consideration is the upstream and downstream bandwidth[1]. As can be seen in the figure, a high clustering is crucial to promote fairness. For the topology with small clustering, around 70% of the users consume more resources than they contribute. This forces the remaining 30% to correct this deficit and contribute the missing resources for little or no personal gain. Some users even present a $FR$ superior to 2, which may be a powerful disincentive for many users to remain in the system.

For the topology with high clustering, however, the boxplots resemble a normal distribution centered at the equilibrium point of $FR = 1$. This is very positive for the system, as it means that most users consume an amount of resources that is equal to their individual contribution.

Next, we investigate the influence of user degree on the fairness ratio. More specifically, Fig. 7.12 correlates the fairness ratio with user degree by means of several dispersion graphs. As before, this figure contains four subplots, each corresponding to a single combination of topology and availability model.

As can be seen in the figure, and contrarily to our prior observations, the user degree is the dominant factor controlling local fairness: The higher the degree is, the higher the asymmetry is, because the number of storage operations is proportional to the number of friends. Interestingly, perfect fairness is only achieved for those users whose degree is close to the average degree of the social graph, which is 6.7 and 13.1 for the low and high clustered graphs, respectively. This gives a clue about the intricate relationship between topology and fairness, whose analytical study is object of future work.

[1]In our experiments, the application workload is homogeneous, which means that asymmetry arises as a result of topological variations

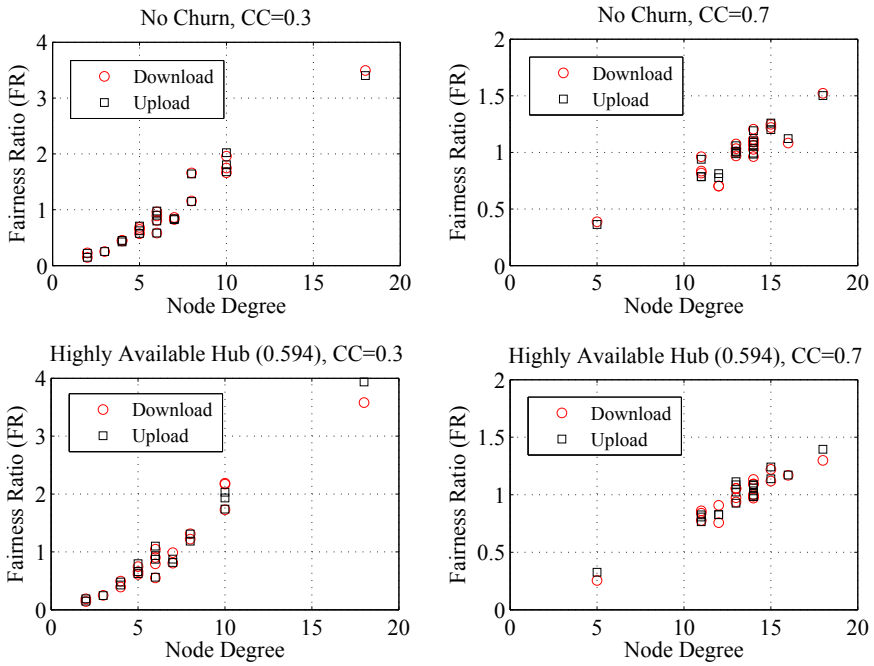



**Figure 7.12:** Relationship between up./down. fairness ratios and node degree for churn/stable scenarios.

Furthermore, the availability of friends does not affect fairness, which can be verified by comparing the subplots of Fig. 7.12 where users are always logged in, labeled "no churn", with those subplots where users join and disconnect from the social cloud. The main reason is that while a user is offline, no data block can be stored in the hard disk of a friend, and vice versa.

Our observations may have important implications on the behavior of users in a social cloud. For instance, given that users with a low degree tend to abuse the system, their friends may, in turn, reject to transact with them until they increase their degree. This could lead to a cold-start situation, where newcomers cannot easily be part of the social cloud. Therefore, further research is needed to guarantee resource fairness in a social cloud by taking into account the underlying system characteristics.

The main insights of this section can be summarized as follows: (i) A *high clustering* coefficient is critical to maintain *fairness* in the system; (ii) The *degree* of a user greatly *determines the fairness* it establishes with the system.

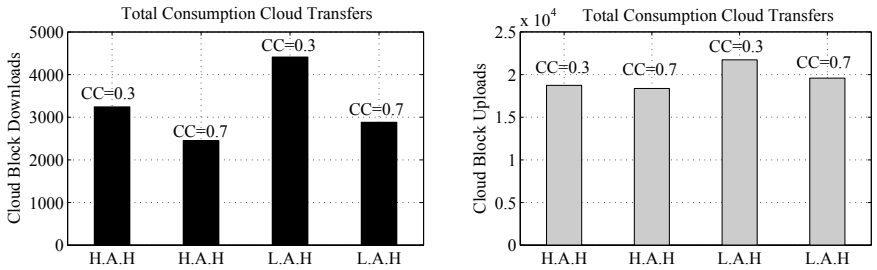



**Figure 7.13:** Cloud block transfers depending on the hub's availability and the clustering coefficient.

### 7.4.6 Cloud Usage & Monetary Costs

Finally, we study the use of storage cloud resources by `FriendBox` clients. Concretely, we illustrate the total number of data block transfers in and out of the cloud when the social "hub" is highly available, abbreviated H.A.H, and low available, abbreviated L.A.H, for our two topologies with clustering coefficients of 0.3 and 0.7, respectively. To avoid biasing the results, the data blocks transferred by the hub were excluded from the final count. The reason was that a highly available hub conducts more block transfers than a hub with a lower availability, which may seriously bias results towards the H.A.H configuration. Results are shown in Fig. 7.13.

For the same degree of clustering, this figure shows that overall the users resort to the online cloud storage service substantially more times when the availability of the hub is low, effect that is more significant for file downloads. This behavior is aggravated for social graphs for which the clustering is small. For instance, for the topology with $CC = 0.3$, the number of data transfers out of the cloud increases a 26.5% when the availability of the social hub decreases from 0.594 to 0.278. The reason is that for social graphs with small clustering, users have fewer chances of downloading data blocks from their friends, thus making the system more dependent on the availability of the hub.

For the same hub availability, a higher $CC$ reduces significantly the number of data block transfers out of the cloud. To give some numbers: If the hub has low availability, the number of transfers is comparatively a 34.6% smaller in the high clustering graph than for the social topology with small clustering. This can be explained by the fact that a higher clustering degree is accompanied by a greater number of links between users, which in general increases the number of data blocks retrievable from friends at any time [27, 99]. This reduces the number of accesses to the cloud.

| | | Storage ($/month) | Down. Traffic ($) | Storage Buffering ($/month, only 1st month) | Down. Buffering Traffic ($) | FriendBox vs. Cloud (1st month) | FriendBox vs. Cloud (permanent) |
|---|---|---|---|---|---|---|---|
| H.A.H. | $CC = 0.3$ | 9.234 | 3.891 | 8.571 | 10.826 | $-21.19\%$ | $-68.19\%$ |
| | $CC = 0.7$ | 9.234 | 2.941 | 8.227 | 10.392 | $-25.37\%$ | $-68.83\%$ |
| L.A.H. | $CC = 0.3$ | 9.234 | 5.294 | 11.417 | 14.421 | $-2.18\%$ | $-64.79\%$ |
| | $CC = 0.7$ | 9.234 | 3.459 | 9.373 | 11.839 | $-17.84\%$ | $-69.24\%$ |
| Amazon S3 | | 18.465 | 22.8 | - | - | - | - |

**Table 7.2:** Costs estimation of FriendBox compared with Amazon S3 for the experiment workload.

Further, we observe that uploads consume a higher amount of cloud resources than downloads. This is because FriendBox minimizes the number of cloud transfers by giving priority to friends in the download schedule, only accessing the cloud in those situations where available friends cannot supply the necessary blocks to complete the file retrieval. However, uploads always require transferring a fraction of the data to the cloud, which increases its overall usage. It should be noted that alternatively uploading and downloading distinct files makes it difficult for offline nodes to download buffered blocks and serve download requests when they become online again. This means that less aggressive workloads would greatly reduce the number of cloud downloaded blocks, since there would be enough blocks available at friends.

Therefore, we see that a higher *CC* alleviates the consumption of cloud resources when the social hub is poorly available. This implies that when the hub is disconnected, Amazon S3 is used to temporarily buffer a smaller number of blocks per file storage operation than when the degree of clustering is low.

The previous observations are reflected in the economic cost of the FriendBox service as visible in Table 7.2[1]. At first glance, we observe that low network *CC* and poor hub availability induce high economic expenses in cloud resources. This particularly impacts on the number of extra blocks buffered in the cloud due to the unavailability of friends at the moment of storing a file. However, we should note that FriendBox greatly reduces the long term cloud costs. For example, configuring FriendBox with $F_C = 0.5$ and $n/k = 2$, users save up 50% of permanent storage costs and $87\% - 77\%$ of download traffic costs compared with Amazon S3. Thus, we conclude that FriendBox is feasible in economic terms.

We summarize this section as follows: (i) The *availability of social hubs* plays an important role in the consumption of *cloud resources*, specially for low clustering topologies; (ii) In general, a *high clustering* degree *reduces* the overall amount of consumed *cloud resources*; (iii) FriendBox provides an *attractive trade-off* between storage service and economic cost.

[1] According to Amazon's S3 at December 2013 we assume 0, 12 per GB of outgoing traffic and 0, 095 per GB/month of storage. Incoming traffic is free of charge.

## 7.5 Discussion and Conclusions

An important conclusion drawn from our evaluation is that *the degree of clustering plays a critical role on how storage resources are exchanged among social links*. More concretely, we have seen that resource fairness, simply understood as a cost-benefit ratio, can exhibit a large imbalance when the cluster coefficient is low.

We envisage two different strategies to address this situation:

- Apply a different placement policy to balance the contributed resources by each user; and
- Increase the cluster coefficient through incentives.

Regarding the first solution, the idea is to replace the round robin allocation policy used in `FriendBox` by a fairer policy. For instance, a better policy would be to allocate much more data to the members with a small number of social ties, because, in general, those users are prone to consume more resources than they contribute. This would free the hubs from donating too much resources to the social cloud.

However, this policy might introduce undesirable effects. For example, a large number of blocks might be allocated to a single friend in an attempt to reduce contribution asymmetry. Once this friend went offline, data availability could be highly affected because the data owner might unable to retrieve a sufficient number of redundant blocks from the remaining set of logged-in friends and the cloud.

Regarding the second solution, we have seen that the best fairness ratio is achieved in the social graphs with high clustering, mainly because the data is better spread among friends without overloading hubs. This leads to the question of which type of incentive mechanism would be appropriate to increase the clustering coefficient and improve the overall fairness.

An appealing way of regulating sharing, providing incentives to users and mitigating the risk of an unfair distribution of resources in a social context is the use of market metaphors as shown in [23]. Although using market-based mechanisms is not a new idea to solve the resource allocation problem in computer systems (see, for example, [155][156]), leveraging digitized social relationships provides benefits in terms of increased trust and lowers the barrier to share spare resources. The key idea would be to provide incentives for users to create new social interactions to increase the cluster coefficient up to the necessary level upon which a fair distribution of work among the whole social network could be achieved.

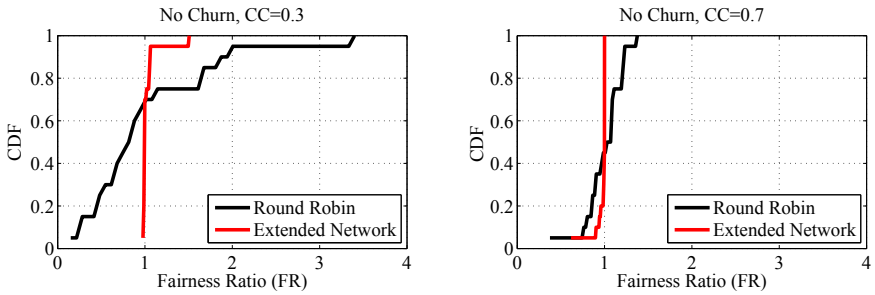



**Figure 7.14:** Fairness ratio for different *CC* using round robin placement or widening storage links to the extended network.

More technically, given a user $v$ of the social cloud, let us consider the ratio of fairness at $v$, namely $FR_v$ and calculated by (Equation 7.3), as the objective metric we want to equalize among all participants. Let us now denote by $d(v_1, v_2)$ the shortest distance between node $v_1$ and $v_2$ in the social graph, and by

$$X_v = \{v_i : d(v, v_i) \geq 1 \wedge FR_{v_i} < 1\}$$

the extended network of $v$ that includes the friends and friends of friends that have a fairness metric less than 1 (contribute less than consume). If we consider the excess of contribution $\mathcal{M} = R_p - R_c$ as a currency in the social market, participants with a $FR_{v_i} > 1$ and $\mathcal{M} > 0$ could be allowed to use its extended network $X_{v_i}$ to discover new social contacts where store new content, increase their $FR_{v_j}$ while decreasing its own $FR_{v_i}$.

To give a sense of the efficacy of this solution, an initial simulation was run on the topologies of Fig. 7.3 using round robin scheduling for exactly 10 rounds of simulation. In each simulation round, storage requests were repeatedly made by all the members of the social cloud using the same setup as in the experiments of the preceding section. Results are depicted in Fig. 7.14, where it is easy to appreciate the high imbalance when the social graph is sparse ($CC = 0.3$) compared when it is highly connected ($CC = 0.7$).

If we turn our attention to the new mechanism, the members of the social network with an initial excess of contribution ($FR > 1$) after the first round of storage requests are allowed to use the extended network on subsequent rounds as explained above until they run out of storage currency. Contrary to the simple round robin policy, Fig. 7.14 clearly verifies how the fairness index at each member is close to the target value of 1: $R_p = R_c \Rightarrow FR = R_p/R_c = 1$, thanks to the use of the extended network and very importantly, irrespective of the clustering degree. The extended network serves to artificially increase the degree of clustering by creating new social links (transient social ties), thereby leading to a better balanced system.

As a side effect, we are also improving data availability by adding more social contacts to the ego-centric graph of each user. The new social acquaintances might even belong to other time zones, which would alleviate the effects of availability correlations.

Although the use of market metaphors in the social cloud is a promising line of work, their final adoption is yet uncertain as it remains to be studied how factors like the topology, availability correlations, etc., shape the form of utility functions. Regarding the implications on trust and privacy of adding new social ties, this decision potentially could be made using reputation measures to leverage the level of trust among direct links and the extended network, addressing to some degree the trust and privacy concerns of users [122].

**Conclusions**. In this Chapter, we have shown how to leverage social relationships to form a dynamic social cloud for storage. To this end, we presented `FriendBox`: A social cloud storage application embedded into Facebook. Moreover, a salient feature of `FriendBox` is that lets a user add an external cloud storage service like Amazon S3 to its social cloud in order to improve data availability while keeping the control of his data.

Although the social cloud model builds upon the unique environment in which users are motivated by social incentives, we have seen that there exist some difficulties and subtleties that prevent the realization of this concept in the real world, such as the *availability correlation* between social contacts and the *asymmetry in contribution levels*. Through a real deployment of `FriendBox` in our campus, we have studied to what extent these factors affect the feasibility of socially oriented storage, paying special attention to the role that the *social graph* plays in the system's performance. Our analysis has revealed new insights on how to design a social storage cloud, in particular, when the storage resources contributed by each member are augmented with an external storage service like Amazon S3. We believe that our analysis provides useful guidelines to improve the design and performance of social cloud systems in the future.

# 8

# Conclusions and Future Directions

## 8.1 Conclusions

Increasingly, end-users demand larger amounts of online storage space to store their personal data. This challenge motivates researchers to devise and study novel personal storage infrastructures. In this thesis, we focused on two popular personal storage architectures: *Personal Clouds* and *social storage systems*. In our view, despite their growing popularity among users and researchers, there still remain some critical aspects to address regarding these systems.

On the one hand, Personal Clouds are centralized systems built on top of dedicated cloud resources for providing a high performance storage service, but their internal infrastructure and behavior remains unknown in many senses since they are proprietary services. On the other hand, social storage systems aim at leveraging the synergy between social networks and storage systems to build a private and secure online storage service. Unluckily, it is unclear if they will be widely adopted by end-users, specially because of their QoS limitations due to their decentralized nature. These two issues are the main topics of this thesis.

### Measurement and Analysis of Personal Clouds

In Part I of this dissertation, we focused on Personal Cloud systems. We examined various aspects of their *internal operation* (metadata back-end) and *external service access* (REST APIs).

- **We contribute the first study of a global-scale Personal Cloud back-end**.

In Chapter 4, we illustrated the internals of a global-scale Personal Cloud service (UbuntuOne, U1). One contribution of this Chapter is the technical description of the U1 back-end, including its *architecture*, *core components* involved in the metadata service hosted in the datacenter of Canonical, as well as the *interactions* of U1 with Amazon S3 to outsource data storage. In fact, Chapter 4 is the first study to depict the metadata store of a real-world vendor.

We also analyzed in depth the *performance issues of the metadata store*. Among our insights, we found that U1 API servers present high tail latencies, that metadata servers exhibit poor

load balancing in the short term, and that a sharded database cluster is an effective way of storing metadata in U1. In our view, this knowledge may be useful for researchers and practitioners in order to comprehend how these systems work and foster research in this field.

- **We provide an extensive analysis of the U1 activity for one month**.

Apart from the performance analysis of the *metadata store*, Chapter 4 also includes the study of the *storage workload* and the *user behavior* in U1. It must be remarked that our study encompasses both the storage and metadata activity of the entire U1 user population (1.17M users), which provides a more general view of U1 than existing measurements on similar systems.

In addition to reconfirm observations of prior works [11, 24] to generalize aspects of these systems, we observed that the distribution of activity across U1 users is *more skewed* than in Dropbox [11] (1% of active users generate 65% of the traffic) and that user operations are *bursty*; users transition between long, idle periods and short, very active ones. We also found that integrating the adequate data management mechanisms in desktop clients may report *significant savings* in storage resources (e.g., delta updates, file-based deduplication) and that *DDoS attacks* against U1 are frequent. These insights can be used to optimize systems like U1.

An important conclusion of our study is that *understanding the behavior of users* is essential to adapt the system to its actual demands and reduce costs. As a result, we suggested improvements to U1 that can also benefit similar Personal Cloud systems in terms of *storage optimizations*, *user behavior detection* and *security*. Actually, our experience in this Chapter points out that there are still technical and research challenges to face for optimizing these services.

- **Characterization and exploitation of Personal Cloud REST API services**.

In Chapter 5, we conducted an active measurement of the REST API service of various vendors to generate a public dataset as a basis for our analysis. We studied several aspects of the transfer QoS of these services, including their *transfer speed*, *variability* and *failure behavior*.

For instance, we found that the transfer speed of these services *greatly varies* from one provider to another and also depending on the client's geographic location, which is a valuable insight for users and developers to choose the best vendor for their needs. In this sense, we observed that *uploads are more variable than downloads* and that such variability depends on various aspects, like the hour of the day. Moreover, we noticed that SugarSync *changed its freemium QoS* unexpectedly, which emphasizes the relevance of the *data lock-in* problem.

We also contributed statistical insights on the transfer QoS of these services (e.g., Poissonity of failures) to help researchers on building simulation environments for Personal Clouds, as well as for providing solid assumptions to their analytical modeling.

Apart from their characterization, we detected that these open REST APIs may be a vector for abuse of Personal Clouds, given the *freemium* business model that most vendors adopt —in line with the attacks reported in Chapter 4. In this thesis, we termed the automated and fraudulent resource consumption that malicious parties may perpetrate on Personal Clouds by exploiting the REST API access to free accounts as *storage leeching problem*. We demonstrated the practicality of abusing Personal Clouds by building a proof-of-concept file-sharing application that benefits from storage leeching to share illicit content, even exploiting storage diversity across multiple vendors. To conclude this Chapter, we discussed the main causes that make storage leeching possible, as well as various alternatives to mitigate this vulnerability.

To summarize, Part I of this dissertation provides a holistic view of the behavior of Personal Clouds, which extends the state-of-the-art knowledge on these systems.

### Exploring QoS in Social Storage Systems

As an alternative to Personal Clouds, social storage systems are emerging as a mean of providing private and secure online storage to end-users. In Part II of this thesis, we studied the storage QoS of social storage systems in terms of *data availability*, *load balancing* and *transfer times*. Our main interest was to understand the way intrinsic phenomena, such as the dynamics of users and the structure of their social relationships, limit the storage QoS of these systems, as well as to research novel mechanisms to ameliorate these limitations.

- **We analyzed the role of data management mechanisms in the QoS of F2F systems**.

In Chapter 6, we focused on the performance of friend-to-friend (F2F) storage systems. First, we noted that these systems differ from large-scale storage systems in the sense that they are highly affected by *small groups of nodes* to store data and *high availability correlations*. In our analysis, we found that these particularities should be seriously considered to implement effective data management techniques.

Concretely, we illustrated that traditional mechanisms, such as estimating *data availability and redundancy*, are *not suitable in this scenario*; they exhibit significant estimation errors that may lead to additional overheads. In this sense, we believed it necessary to coin a new notion

of data availability adjusted to the daily-patterned dynamics of users, called *daily data availability*. Moreover, we presented a *history-based* data availability estimation tailored to this new notion of data availability that accurately calculates the level of data redundancy.

With respect to *load balancing*, we discovered that storing more data blocks at highly available nodes may achieve higher data availability requiring less data redundancy than a simple round-robin data placement. However, considering small groups of friends to store data, this type of placement makes highly available nodes to be overloaded, inducing very poor load balancing. To solve this problem, we demonstrated that the combination of a round-robin placement and our history-based data availability calculation obtains an adequate degree of data availability without compromising load balancing in a F2F scenario.

Regarding *transfer performance*, the correlated availabilities of nodes in a F2F system makes it necessary to differentiate between if a file is available at a certain instant and if it is retrievable in a reasonable amount of time. This effect dominates the performance of transfer scheduling policies, among which we found *no clear winner*.

- **A novel hybrid storage architecture to improve the QoS of social storage systems**.

Unfortunately, one of the main conclusions of our analysis in Chapter 6 was that it is *difficult* to provide a high-quality storage service in a *purely decentralized* F2F system. Thus, we contributed with a new hybrid architecture design, namely `F2Box`, that blends user resources with cloud storage services in a F2F system to enhance storage QoS.

In `F2Box`, the cloud reduces transfer times and limits the amount of redundancy needed to achieve a targeted data availability, which makes the system more scalable. Moreover, `F2Box` is equipped with a battery of data management techniques that enable users to decide the best trade-off between data control, and service QoS/cost. Our results certify that our architecture leverages the benefits of combining the best of both worlds, which may represent a step towards the wide adoption of F2F storage systems by end-users.

- **Understanding the role of network topology in the Social Cloud**.

In adition to F2F storage systems, the "social cloud" is also becoming a popular socially-motivated computing paradigm that enables resource sharing across users. Our main interest in Chapter 7 was to understand, from an empirical perspective, the role that *social network topology plays in the storage QoS* provided by a social cloud.

To conduct this analysis, we implemented `FriendBox`, the first social cloud storage application that materializes the architecture presented in Chapter 6. We studied how the *correlated*

*user availabilities*, the *friendset size* and the *degree of connectivity among friends* impact on the storage QoS of the system. Moreover, we analyzed how the combinations of these elements influence the consumption of cloud resources. In summary, our analysis has revealed new insights on how to design a social storage cloud, in particular, when the storage resources contributed by each member are augmented with an external storage service. In our view, understanding the underlying infrastructural issues in a social cloud, such as the availabilities of users and the network topology, is critical to avoid undesirable effects like overloading highly available users. Thus, our insights in this Chapter may guide the design of sophisticated social cloud markets that also take into account the storage infrastructure when allocating and trading resources to enhance the performance of the system.

In summary, the Part II of this thesis contributes by providing new insights on the performance of social storage systems as well as alternative architectural designs. Our contributions may help to understand and enhance these systems, which is fundamental to their eventual adoption by end-users.

## 8.2 Future Directions

Optimizing and designing novel personal storage systems is, and will probably be, an active research topic given the increasing needs and new requirements that users exhibit. In the course of this thesis, we found several research lines that may be interesting to develop, among which we highlight the following ones:

- *Hot/Cold Personal Data Identification*: In many cases, Personal Clouds resort to third-party cloud storage providers for outsourcing data storage. This means that Personal Clouds are highly motivated to cut down costs in storage resources in order to maximize their profit. From our analysis in Chapter 4, we realized that almost 80% of the new files stored in U1 are read only once. Furthermore, 60% of reads over files occur within 3 days, which means that in the long term files are rarely accessed. We believe that these empirical observations can be exploited to build intelligent algorithms that decide when a file can be considered *warm or cold*. This may enable Personal Clouds to migrate these files to cold storage services, such as Amazon Glacier[1], and save a significant fraction of their monthly expenses in storage resources.

[1] `http://aws.amazon.com/es/glacier/`

- *Benchmarking On-Premise Personal Clouds*: On-premise Personal Clouds are an attractive solution to enterprises since they provide Dropbox-like functionality (file storage, synchronization and sharing) to employees without the need of outsourcing sensitive data to a public vendor. However, there are no tailored tools for accurately evaluating and right-sizing an on-premise Personal Cloud, which is critical to the performance and cost of the system being deployed. Intuitively, the specific user activity within an enterprise may play a key role on choosing the correct vendor, given that Personal Clouds integrate distinct data reduction techniques. Our idea is to first identify the different types of user behavior commonly found in a Personal Cloud, namely stereotypes. The second step is to develop a benchmarking framework for modeling and reproducing user behavior in a Personal Cloud at scale.

- *Topology-level Incentives in Social Cloud Storage*: In Chapter 7, we discussed that resource fairness in a social cloud, simply understood as a cost-benefit ratio, can exhibit a large imbalance when the degree of connectivity or cluster coefficient among users is low. We presented two mechanisms to ameliorate the impact of such unfairness in this context: a *topology-aware data placement strategy* and *the provisioning of incentives to modify the underlying social network topology*. An interesting research line may be to continue this discussion with a practical implementation of these mechanisms in a real social cloud market to evaluate their implications, also considering the effect of user dynamics.

# A
# U1 Upload Management

The management of file uploads is one of the most complex parts in the U1 architecture[1]. Specifically, U1 resorts to the multipart upload API offered by Amazon S3[2]. The lifecycle of an upload is closely related to this API, where several U1 RPC calls are involved (see Table A.1).

| | |
|---|---|
| `dal.add_part_to_uploadjob` | Continues a multipart upload by adding a new chunk. |
| `dal.delete_uploadjob` | Garbage-collects the server-side state for a multipart upload, either because of commit or cancellation. |
| `dal.get_reusable_content` | Check whether the server already has the content that is being uploaded. |
| `dal.get_uploadjob` | Get the server-side state for a multipart upload. |
| `dal.make_content` | Make a file entry in the metadata store (the equivalent of an inode). |
| `dal.make_uploadjob` | Set up the server-side structure for multipart upload. |
| `dal.set_uploadjob_multipart_id` | Set the requested Amazon S3 multipart upload id to the `uploadjob`. |
| `dal.touch_uploadjob` | Check if the client has canceled the multipart upload (garbage collection after a week). |

**Table A.1:** Upload related RPC operations that interact with the metadata store.

Internally, U1 uses a persistent data structure called `uploadjob` that keeps the state of a multipart file transfer between the client and Amazon S3. The main objective of multipart uploads in U1 is to provide user with a way of interrupting/resuming large upload data transfers. `uploadjob` data structures are stored in the metadata store during their life-cycle. RPC operations during the multipart upload process guide the lifecycle of `uploadjobs` (see Fig. A.1).

Upon the reception of an upload request, U1 first checks if the file content is already stored in the service, by means of a SHA-1 hash sent by the user. If deduplication is not applicable to the new file, a new upload begins. The API server that handles the upload sends an RPC to create an entry for the new file in the metadata store.

In the case of a multipart upload, the API server creates a new `uploadjob` data structure to track the process. Subsequently, the API process requests a multipart id to Amazon S3 that will identify the current upload until its termination. Once the id is assigned to the `uploadjob`,

[1] Downloads are simpler: API servers only perform a single request to Amazon S3 for forwarding the data to the client.

[2] `http://docs.aws.amazon.com/AmazonS3/latest/dev/UsingRESTAPImpUpload.html`

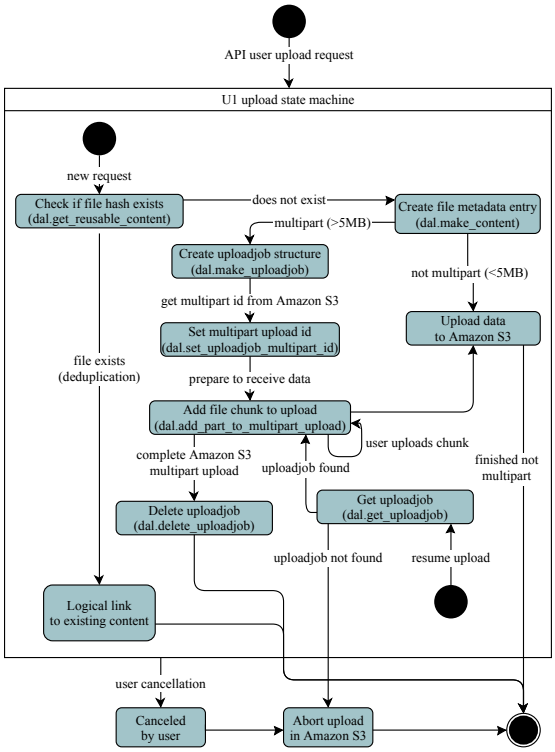



**Figure A.1:** Upload state machine in U1.

the API server uploads to Amazon S3 the chunks of the file transferred by the user (5MB), updating the state of the `uploadjob`.

When the upload finishes, the API server deletes the `uploadjob` data structure from the metadata store and notifies Amazon S3 about the completion of the transfer.

Finally, U1 also executes a periodic garbage-collection process on `uploadjob` data structures. U1 checks if an `uploadjob` is older than one week (`dal.touch_uploadjob`). In the affirmative case, U1 assumes that the user has canceled this multipart upload permanently and proceeds to delete the associated `uploadjob` from the metadata store.

# B

# Glossary

**API**: Application Programing Interface.

**BTT**: Block Transfer Time.

**CC**: Clustering Coefficient.

**DOSN**: Distributed Online Social Network.

**F2F**: Friend-to-Friend.

**GDM**: Group Disconnection Matching.

**GPM**: Group Presence Matching.

**IaaS**: Infrastructure-as-a-Service.

**MTTS**: Minimum Time To Schedule.

**NAS**: Network Attached Storage.

**OS**: Operating System.

**OSN**: Online Social Network.

**OTTS**: Optimal Time To Schedule.

**P2P**: Peer-to-Peer.

**QoS**: Quality of Service.

**REST**: REpresentational State Transfer.

**RPC**: Remote Procedure Call.

**SaaS**: Software-as-a-Service.

**SLA**: Service Level Agreement.

**SME**: Small and Medium Enterprise.

**TCP**: Transmission Control Protocol.

**TTS**: Time To Schedule.

# Bibliography


[1] John Gantz and David Reinsel. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the Future*, 2012.

[2] Jay J Wylie, Michael W Bigrigg, John D Strunk, Gregory R Ganger, Han Kiliccote, and Pradeep K Khosla. Survivable information storage systems. *IEEE Computer*, 33(8):61–68, 2000.

[3] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *USENIX HotOS'01*, 2001.

[4] Robert J.T. Morris and Brian J. Truskowski. The evolution of storage systems. *IBM systems Journal*, 42(2):205–217, 2003.

[5] Ragib Hasan, Zahid Anwar, William Yurcik, Larry Brumbaugh, and Roy Campbell. A survey of peer-to-peer storage techniques for distributed file systems. In *IEEE ITCC'05*, volume 2, pages 205–213, 2005.

[6] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *ACM SOSP'01*, 2001.

[7] Gartner, Inc. Forecast: Consumer digital storage needs, 2010-2016. `http://www.gartner.com/newsroom/id/2060215`, 2012.

[8] Techcrunch. Dropbox hits 200m users, unveils new "for business" client combining work and personal files. `http://techcrunch.com`, 2013.

[9] A. Li, X. Yang, S. Kandula, and M. Zhang. Cloudcmp: comparing public cloud providers. In *ACM IMC'10*, pages 1–14, 2010.

[10] Hussam Abu-Libdeh, Lonnie Princehouse, and Hakim Weatherspoon. Racs: a case for cloud storage diversity. In *ACM SoCC '10*, pages 229–240, 2010.

[11] Idilio Drago, Marco Mellia, Maurizio M Munafo, Anna Sperotto, Ramin Sadre, and Aiko Pras. Inside dropbox: understanding personal cloud storage services. In *ACM IMC'12*, pages 481–494, 2012.

[12] Siani Pearson and Azzedine Benameur. Privacy, security and trust issues arising from cloud computing. In *IEEE CloudCom'10*, pages 693–702, 2010.

[13] A survey on security issues in service delivery models of cloud computing. *Journal of Network and Computer Applications*, 34(1):1–11, 2011.

[14] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Designing Privacy Enhancing Technologies*, pages 46–66, 2001.

[15] Ranjita Bhagwan Kiran, Kiran Tati, Yu-chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: System support for automated availability management. In *USENIX NSDI'04*, 2004.

[16] 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. *ACM SIGOPS Operating Systems Review*, 36(SI):1–14, 2002.

[17] WuaLa. Wuala. `http://www.wuala.com`, 2010.

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

[19] Landon P. Cox and Brian D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *ACM SOSP '03*, pages 120–132, 2003.

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

[21] CrashPlan. Crashplan. `http://www.crashplan.com`, 2015.

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

[23] K. Chard, K. Bubendorfer, S. Caton, and O.F. Rana. Social cloud computing: A vision for socially motivated resource sharing. *IEEE Transactions on Services Computing*, (4):551–563, 2012.

[24] Zhenhua Li, Cheng Jin, Tianyin Xu, Christo Wilson, Yao Liu, Linsong Cheng, Yunhao Liu, Yafei Dai, and Zhi-Li Zhang. Towards network-level efficiency for cloud storage services. In *ACM IMC'14*, 2014.

[25] Raúl Gracia-Tinedo, Marc Sánchez-Artigas, Adrián Moreno-Martínez, and Pedro García-López. FriendBox: A hybrid f2f personal storage application. In *IEEE CLOUD'12*, pages 131–138, 2012.

[26] Raúl Gracia-Tinedo, Marc Sánchez-Artigas, and Pedro García-López. F2box: Cloudifying f2f storage systems with high availability correlation. In *IEEE CLOUD'12*, pages 123–130, 2012.

[27] Raúl Gracia-Tinedo, Marc Sánchez-Artigas, and Pedro García-López. Analysis of data availability in f2f storage systems: When correlations matter. In *IEEE P2P'12*, pages 225–236, 2012.

[28] Adrián Moreno-Martínez, Raúl Gracia-Tinedo, Marc Sánchez-Artigas, and Pedro García-López. Friendbox: A cloudified f2f storage application. In *IEEE P2P'12*, pages 75–76, 2012.

[29] Raúl Gracia-Tinedo, Marc Sánchez Artigas, Adrián Moreno-Martınez, Cristian Cotes, and Pedro Garcıa López. Actively measuring personal cloud storage. In *IEEE CLOUD'13*, pages 301–308, 2013.

[30] Raúl Gracia-Tinedo, Marc Sánchez Artigas, and Pedro Garcıa López. Cloud-as-a-gift: Effectively exploiting personal cloud free accounts via REST APIs. In *IEEE CLOUD'13*, pages 621–628, 2013.

[31] Raúl Gracia-Tinedo, Marc Sánchez Artigas, Aleix Ramírez, Adrián Moreno-Martínez, Xavier León, and Pedro García López. Giving form to social cloud storage through experimentation: Issues and insights. *Elsevier Future Generation Computer Systems*, 40:1–16, 2014.

[32] Raúl Gracia-Tinedo, Marc Sánchez-Artigas, and Pedro García-López. eWave: Leveraging energy-awareness for in-line deduplication clusters. In *ACM SYSTOR'14*, pages 6:1–6:11, 2014.

[33] Raúl Gracia-Tinedo, Danny Harnik, Dalit Naor, Dmitry Sotnikov, Sivan Toledo, and Aviad Zuck. SDGen: Mimicking Datasets for Content Generation in Storage Benchmarks. In *USENIX FAST'15*, 2015.

[34] P. García-López, S. Toda-Flores, C. Cotes-González, M. Sánchez-Artigas, and J. Lenton. Stacksync: Bringing elasticity to dropbox-like file synchronization. In *ACM/IFIP/USENIX Middleware'14*, page In press, 2014.

[35] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A view of cloud computing. *Communications of the ACM*, 53(4), 2010.

[36] Alain Azagury, Vladimir Dreizin, Michael Factor, Ealan Henis, Dalit Naor, Noam Rinetzky, Ohad Rodeh, Julian Satran, Ami Tavory, and Lena Yerushalmi. Towards an object store. In *IEEE MSST'03*, pages 165–176, 2003.

[37] Idilio Drago, Enrico Bocchi, Marco Mellia, Herman Slatman, and Aiko Pras. Benchmarking personal cloud storage. In *ACM IMC'13*, pages 205–212, 2013.

[38] Zhenhua Li, Christo Wilson, Zhefu Jiang, Yao Liu, Ben Y Zhao, Cheng Jin, Zhi-Li Zhang, and Yafei Dai. Efficient batched synchronization in dropbox-like cloud storage services. In *ACM/IFIP/USENIX Middleware'13*, pages 307–327. 2013.

[39] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23(3):337–343, 1977.

[40] J. Ziv and A. Lempel. Compression of Individual Sequences via Variable-Rate Coding. *IEEE Transactions on Information Theory*, 24(5):530–536, September 1978.

[41] D. A. Huffman. A Method for the Construction of Minimum-Redundancy Codes. *Proceedings of the Institute of Radio Engineers*, 40(9):1098–1101, September 1952.

[42] P. Deutsch and J. L. Gailly. Zlib compressed dataformat specification version 3.3. Technical Report Technical ReportRFC 1950, Network Working Group, 1966.

[43] Oracle. What is ZFS? http://docs.oracle.com/cd/E19253-01/819-5461/zfsover-2/.

[44] Nimble Storage. Nimble storage: Engineered for efficiency. Technical report, WP-EFE-0812, Nimble Storage, 2012.

[45] J. Quintal E. Traitel J. Tate, B. Tuv-El and B. Whyte. Real-time compression in SAN volume controller and Storwize V7000. Technical report, REDP-4859-00. IBM, 2012.

[46] Benjamin Zhu, Kai Li, and R Hugo Patterson. Avoiding the disk bottleneck in the data domain deduplication file system. In *USENIX FAST'08*, volume 8, pages 1–14, 2008.

[47] Dutch T Meyer and William J Bolosky. A study of practical deduplication. *ACM Transactions on Storage (TOS)*, 7(4):14, 2012.

[48] Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Side channels in cloud services, the case of deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.

[49] Libsync. https://github.com/librsync/librsync.

[50] Andrew Tridgell, Paul Mackerras, et al. The rsync algorithm, 1996.

[51] Barry Leiba. Oauth web authorization protocol. *IEEE Internet Computing*, 16(1):74–77, 2012.

[52] Pedro García-López, Adrián Moreno-Martínez, Marc Sánchez-Artigas, Marko Vukolic, Hamza Harkous, Thanasis Papaioannou, Hao Zhuang, and Stuart Langridge. Roadmap of outcomes. http://cloudspaces.eu/deliverables/doc_download/2-d2-1-roadmap-of-outcomes, 2013.

[53] Cloud or on-premise? Egnyte lets you choose, one file at a time. http://diginomica.com/2015/01/22/cloud-premise-egnyte-lets-choose-one-file-time/.

[54] M. Mulazzani, S. Schrittwieser, M. Leithner, M. Huber, and E.Weippl. Dark clouds on the horizon: Using cloud storage as attack vector and online slack space. In *USENIX Security*, pages 5–8, 2011.

[55] Lluís Pàmies-Juárez. *On the Design and Optimization of Heterogeneous Distributed Storage Systems*. PhD thesis, Departament d'Enginyeria Informàtica i Matemàtiques, Universitat Rovira i Virgili, 2011.

[56] J. Kubiatowicz et. al. Oceanstore: An architecture for global-scale persistent storage. *ACM SIGPLAN'00*, 35:190–201, 2000.

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

[58] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *ACM SIGCOMM '06*, pages 147–158, 2006.

[59] A.G. Dimakis, P.B. Godfrey, M.J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *IEEE INFOCOM'07*, 2007.

[60] Anwitaman Datta and Karl Aberer. Internet-scale storage systems under churn–a study of the steady-state using markov models. In *IEEE P2P'06*, pages 133–144, 2006.

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

[62] Anhai Doan, Raghu Ramakrishnan, and Alon Y Halevy. Crowdsourcing systems on the world-wide web. *Communications of the ACM*, 54(4):86–96, 2011.

[63] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM Intl. Workshop on Grid Computing (GRID)*, 2004.

[64] Atif Nazir, Saqib Raza, and Chen-Nee Chuah. Unveiling facebook: a measurement study of social network based applications. In *ACM IMC'08*, pages 43–56, 2008.

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

[66] Simon Caton, Christian Haas, Kyle Chard, Kris Bubendorfer, and Omer F Rana. A social compute cloud: Allocating and sharing infrastructure resources via social networks. *IEEE Transactions on Services Computing*, (3):359–372, 2014.

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

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

[69] Robert J. McEliece and Mark Kac. *The theory of information and coding : a mathematical framework for communication*. Encyclopedia of mathematics and its applications. Addison-Wesley Pub. Co., Reading, Massachusetts, 1977.

[70] Bogdan Popescu. Safe and private data sharing with turtle: friends team-up and beat the system (transcript of discussion). In *Security Protocols*, pages 221–230. Springer, 2006.

[71] Tom N Jagatic, Nathaniel A Johnson, Markus Jakobsson, and Filippo Menczer. Social phishing. *Communications of the ACM*, 50(10):94–100, 2007.

[72] Leyla Bilge, Thorsten Strufe, Davide Balzarotti, and Engin Kirda. All your contacts are belong to us: automated identity theft attacks on social networks. In *ACM WWW'09*, pages 551–560, 2009.

[73] Randy Baden, Adam Bender, Neil Spring, Bobby Bhattacharjee, and Daniel Starin. Persona: an online social network with user-defined privacy. In *ACM SIGCOMM Computer Communication Review*, volume 39, pages 135–146, 2009.

[74] Diaspora. https://joindiaspora.com.

[75] Sonja Buchegger and Anwitaman Datta. A case for p2p infrastructure for social networks-opportunities & challenges. In *IEEE WONS'09*, pages 161–168, 2009.

[76] Leucio Antonio Cutillo, Refik Molva, and Thorsten Strufe. Safebook: A privacy-preserving online social network leveraging on real-life trust. *IEEE Communications Magazine*, 47(12):94–101, 2009.

[77] Sonja Buchegger, Doris Schiöberg, Le-Hung Vu, and Anwitaman Datta. Peerson: P2p social networking: early experiences and insights. In *ACM EuroSys Workshop on Social Network Systems*, pages 46–52, 2009.

[78] Rajesh Sharma and Anwitaman Datta. Supernova: Super-peers based architecture for decentralized online social networks. In *IEEE COMSNETS'12*, pages 1–10, 2012.

[79] A. Shakimov, H. Lim, R. Caceres, L.P. Cox, K. Li, Dongtao Liu, and A. Varshavsky. Vis-à-vis: Privacy-preserving online social networking via virtual individual servers. In *IEEE COMSNETS'11*, pages 1–10, 2011.

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

[81] Dropbox. A dropbox for business guide. https://www.dropbox.com/static/business/ resources/dfb_security_whitepaper.pdf, 2015.

[82] Box. Scaling mysql for the web. https://www.percona.com/live/mysql-conference-2013/sessions/scaling-mysql-web, 2013.

[83] Glauber Gonçalves, Idilio Drago, Ana Paula Couto da Silva, Alex Borges Vieira, and Jussara M Almeida. Modeling the dropbox client behavior. In *IEEE ICC'14*, volume 14, 2014.

[84] Yupu Zhang, Chris Dragga, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. *-box: towards reliability and consistency in dropbox-like file synchronization services. In *USENIX HotStorage'13*, pages 2–2, 2013.

[85] Herman Slatman. Opening up the sky: a comparison of performance-enhancing features in skydrive and dropbox. In *Proceedings of the 18th Twente Student Conference on IT*, 2013.

[86] Songbin Liu, Xiaomeng Huang, Haohuan Fu, and Guangwen Yang. Understanding data characteristics and access patterns in a cloud storage system. In *IEEE/ACM CCGrid'13*, pages 327–334, 2013.

[87] Zach Hill, Jie Li, Ming Mao, Arkaitz Ruiz-Alvarez, and Marty Humphrey. Early observations on the performance of windows azure. In *ACM HPDC '10*, pages 367–376, 2010.

[88] Mayur R. Palankar, Adriana Iamnitchi, Matei Ripeanu, and Simson Garfinkel. Amazon s3 for science grids: a viable solution? In *ACM DADC'08*, pages 55–64, 2008.

[89] A. Bergen, Y. Coady, and R. McGeer. Client bandwidth: The forgotten metric of online storage providers. In *IEEE Pacific Rim Conference on Communications, Computers and Signal Processing*, pages 543–548, 2011.

[90] W. Hu, T. Yang, and J.N. Matthews. The good, the bad and the ugly of consumer cloud storage. *ACM SIGOPS Operating Systems Review*, 44(3):110–115, 2010.

[91] Meiko Jensen, Nils Gruschka, and Ralph Herkenhöner. A survey of attacks on web services. *Computer Science - Research and Development*, 24:185–197, 2009.

[92] Juraj Somorovsky, Mario Heiderich, Meiko Jensen, Jörg Schwenk, Nils Gruschka, and Luigi Lo Iacono. All your clouds are belong to us: security analysis of cloud management interfaces. In *ACM CCSW'11*, pages 3–14, 2011.

[93] Luis Vaquero, Luis Rodero-Merino, and Daniel Morán. Locking the sky: a survey on iaas cloud security. *Computing*, 91:93–118, 2011.

[94] J. Idziorek and M. Tannian. Exploiting cloud utility models for profit and ruin. In *IEEE CLOUD'11*, pages 33–40, july 2011.

[95] J. Idziorek, M. Tannian, and D. Jacobson. Attribution of fraudulent resource consumption in the cloud. In *IEEE CLOUD'12*, pages 99–106, 2012.

[96] J. Srinivasan, Wei Wei, Xiaosong Ma, and Ting Yu. Emfs: Email-based personal cloud storage. In *IEEE NAS'11*, pages 248–257, 2011.

[97] Avishay Traeger, Nikolai Joukov, Josef Sipek, and Erez Zadok. Using free web storage for data backup. In *ACM StorageSS'06*, pages 73–78, 2006.

[98] Hsiang-Ching Chao, Tzong-Jye Liu, Kuong-Ho Chen, and Chyi-Ren Dow. A seamless and reliable distributed network file system utilizing webspace. In *IEEE WSE'08*, pages 65–68, 2008.

[99] 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.

[100] 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.

[101] Nguyen Tran, Frank Chiang, and Jinyang Li. Efficient cooperative backup with decentralized trust management. *ACM Transactions on Storage (TOS)*, 8(3):8, 2012.

[102] M. Steiner, T. En-Najjary, and E.W. Biersack. A global view of kad. In *ACM IMC'07*, 2007.

[103] S. Guha, N. Daswani, and R. Jain. An experimental study of the skype peer-to-peer voip system. In *IPTPS'06*, 2006.

[104] Marc Sánchez-Artigas and Enrique Férnandez-Casado. Evaluation of p2p systems under different churn models: Why we should bother. In *Euro-Par Parallel Processing*, pages 541–553, 2011.

[105] R.J. Dunn, J. Zahorjan, S.D. Gribble, and H.M. Levy. Presence-based availability and p2p systems. In *IEEE P2P'05*, pages 209–216, 2005.

[106] Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Finding good partners in availability-aware p2p networks. In *Stabilization, Safety, and Security of Distributed Systems*, volume 5873, pages 472–484. 2009.

[107] A Kermarrec, Erwan Le Merrer, Gilles Straub, and Alexandre Van Kempen. Availability-based methods for distributed storage systems. In *IEEE SRDS'12*, pages 151–160, 2012.

[108] James W. Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *USENIX NSDI'06*, pages 6–6, 2006.

[109] Krzysztof Rzadca, Anwitaman Datta, and Sonja Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. In *IEEE ICDCS'10*, pages 599–609, 2010.

[110] Lluis Pàmies-Juárez, Pedro García-López, and Marc Sánchez-Artigas. Heterogeneity-aware erasure codes for peer-to-peer storage systems. In *IEEE ICPP'09*, 2009.

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

[112] T. Mager, E. Biersack, and P. Michiardi. A measurement study of the wuala on-line storage service. In *IEEE P2P'12*, pages 237–248, 2012.

[113] Zhi Yang, Ben Y. Zhao, Yuanjian Xing, Song Ding, Feng Xiao, and Yafei Dai. Amazingstore: available, low-cost online storage service using cloudlets. In *IPTPS'10*, pages 2–2, 2010.

[114] Ye Sun, Fangming Liu, Bo Li, Baochun Li, and Xinyan Zhang. Fs2you: Peer-assisted semi-persistent online storage at a large scale. In *IEEE INFOCOM'09*, pages 873–881, 2009.

[115] Ctera. http://www.ctera.com/, 2011.

[116] Cleversafe. `http://www.cleversafe.com`, 2010.

[117] R. Curry, C. Kiddle, N. Markatchev, R. Simmonds, Tingxi Tan, M. Arlitt, and B. Walker. Facebook meets the virtualized enterprise. In *IEEE EDOC'08*, pages 286–292, 2008.

[118] Zhenhua Guo, Raminderjeet Singh, and Marlon Pierce. Building the polargrid portal using web 2.0 and opensocial. In *GCE'09*, pages 5:1–5:8, 2009.

[119] K. John, K. Bubendorfer, and K. Chard. A social cloud for public eresearch. In *IEEE e-Science'11*, pages 363–370, 2011.

[120] A.M. Thaufeeg, K. Bubendorfer, and K. Chard. Collaborative eresearch in a social cloud. In *IEEE e-Science'11*, pages 224–231, 2011.

[121] P. Druschel and A. Rowstron. Past: a large-scale, persistent peer-to-peer storage utility. In *HotOS'01*, pages 75–80, 2001.

[122] Xiang Zuo, J. Blackburn, N. Kourtellis, J. Skvoretz, and A. Iamnitchi. The power of indirect ties in friend-to-friend storage systems. In *IEEE P2P'14*, pages 1–5, 2014.

[123] F. Research. The personal cloud: Transforming personal computing, mobile, and web markets, 2011.

[124] Russell Sears, Catharine Van Ingen, and Jim Gray. To blob or not to blob: Large object storage in a database or a filesystem? Technical report, Microsoft Research, 2007.

[125] Jialin Li, Naveen Kr Sharma, Dan RK Ports, and Steven D Gribble. Tales of the tail: Hardware, os, and application-level sources of tail latency. In *ACM SoCC'14*, 2014.

[126] Mary G Baker, John H Hartman, Michael D Kupfer, Ken W Shirriff, and John K Ousterhout. Measurements of a distributed file system. In *ACM SIGOPS Operating Systems Review*, volume 25, pages 198–212, 1991.

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

[128] Andrew W Leung, Shankar Pasupathy, Garth R Goodson, and Ethan L Miller. Measurement and analysis of large-scale network file system workloads. In *USENIX ATC'08*, volume 1, pages 5–2, 2008.

[129] W.W. Hsu and A.J. Smith. Characteristics of i/o traffic in personal computer and server workloads. *IBM Systems Journal*, 42(2):347–372, 2003.

[130] Subramanian Muralidhar, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Viswanath Sivakumar, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm blob storage system. In *USENIX OSDI'14*, pages 383–398, 2014.

[131] Jelena Mirkovic and Peter Reiher. A taxonomy of DDoS attack and DDoS defense mechanisms. *ACM SIGCOMM Computer Communication Review*, 34(2):39–53, 2004.

[132] Albert-Laszlo Barabasi. The origin of bursts and heavy tails in human dynamics. *Nature*, 435(7039):207–211, 2005.

[133] Mark E Crovella and Azer Bestavros. Self-similarity in world wide web traffic: evidence and possible causes. *IEEE/ACM Transactions on Networking*, 5(6):835–846, 1997.

[134] Seppo Hätönen, Aki Nyrhinen, Lars Eggert, Stephen Strowes, Pasi Sarolahti, and Markku Kojo. An experimental study of home gateway characteristics. In *ACM IMC'10*, pages 260–266, 2010.

[135] Pavan Deolasee, Amol Katkar, Ankur Panchbudhe, Krithi Ramamritham, and Prashant Shenoy. Adaptive push-pull: disseminating dynamic web data. In *ACM WWW'01*, pages 265–274, 2001.

[136] Jane Silber. Shutting down Ubuntu One file services. http://blog.canonical.com/2014/04/02/shutting-down-ubuntu-one-file-services/, April 2014.

[137] Wikipedia. http://en.wikipedia.org/wiki/Dropbox_(service).

[138] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. Planetlab: an overlay testbed for broad-coverage services. *SIGCOMM Computer Communications Review*, 33(3):3–12, 2003.

[139] Hari Balakrishnan, Mark Stemm, Srinivasan Seshan, and Randy H. Katz. Analyzing stability in wide-area network performance. *ACM SIGMETRICS'97*, 25(1):2–12, 1997.

[140] M. Allman, V. Paxson, W. Stevens, et al. Tcp congestion control. 1999.

[141] A. Iosup, N. Yigitbasi, and D. Epema. On the performance variability of production cloud services. In *IEEE/ACM CCGRID'11*, pages 104–113, 2011.

[142] John R. Douceur. The sybil attack. In *IPTPS'01*, pages 251–260, 2002.

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

[144] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. In *ACM STOC'97*, pages 654–663, 1997.

[145] 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.

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

[147] Lluis Pàmies-Juárez, Pedro García-López, Marc Sánchez-Artigas, and Blas Herrera. Towards the design of optimal data redundancy schemes for heterogeneous cloud storage infrastructures. *Computer Networks*, 55(5):1100–1113, 2011.

[148] Kevin M. Greenan, James S. Plank, and Jay J. Wylie. Mean time to meaningless: Mttdl, markov models, and storage system reliability. In *USENIX HotStorage'10*, 2010.

[149] H. Kellerer, U. Pferschy, and D. Pisinger. *Knapsack Problems*. Springer Verlag, 2004.

[150] E. Fernández-Casado, M. Sánchez-Artigas, and P. García-López. Affluenza: Towards universal churn generation. In *P2P'10*, pages 1–2, 2010.

[151] Z. Yao, D. Leonard, X. Derek, X. Wang, and D. Loguinov. Modeling Heterogeneous User Churn and Local Resilience of Unstructured P2P Networks. In *IEEE ICNP'06*, 2006.

[152] R. J. McEliece and D. V. Sarwate. On sharing secrets and reed-solomon codes. *Communications of the ACM*, 24(9):583–584, 1981.

[153] J. K. Resch and J. S. Plank. AONT-RS: blending security and performance in dispersed storage systems. In *USENIX FAST'11*, pages 191–202, 2011.

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

[155] IE Sutherland. A futures market in computer time. *Communications of the ACM*, 11(6):449–451, 1968.

[156] Michal Feldman, Kevin Lai, and Li Zhang. The Proportional-Share Allocation Market for Computational Resources. *IEEE Transactions on Parallel Distributed Systems*, 20(8):1075–1088, 2009.