Online Data Backup: a Peer-Assisted Approach

Laszlo Toka*†, Matteo Dell'Amico*, Pietro Michiardi*
{laszlo.toka, matteo.dell-amico, pietro.michiardi}@eurecom.fr
* Eurecom, Sophia-Antipolis, France † Budapest University of Technology and Economics, Hungary

***Abstract*—In this work we study the benefits of a peer-assisted approach to online backup applications, in which spare bandwidth and storage space of end-hosts complement that of an online storage service. Via simulations, we analyze the interplay between two key aspects of such applications: *data placement* and *bandwidth allocation*. Our analysis focuses on metrics such as the time required to complete a backup and a restore operation, as well as the storage costs.**

We show that, by using adequate bandwidth allocation policies in which storage space at a cloud provider can be used temporarily, hybrid systems can achieve performance comparable to traditional client-server architectures at a fraction of the costs. Moreover, we explore the impact of mechanisms to impose fairness and conclude that a peer-assisted approach does not discriminate peers in terms of performance, but associates a storage cost to peers contributing with little resources.

I. Introduction

Backup is a well-known nuisance for any computer user. To protect themselves from the risk of losing their valuable data, individuals and organizations have to undergo a tedious, and sometimes expensive, process. Especially for home systems, backup is far from being a solved problem from a practical standpoint. The need for manual operations such as connecting and disconnecting hard drives or burning optical media often results in users neglecting or forgetting to backup their data. For users that are willing to pay an additional cost, there are systems that perform backup without requiring user intervention, e.g. [5], but even these solutions have the shortcoming of residing in the same place the original data is. Hence, an event such as theft, fire or flood will cause the loss of both the original data and its backup.

A convenient solution to these problems is represented by cloud storage systems (e.g., Dropbox [2]) which transparently synchronize, when machines are connected to the Internet, the local copy of data with a remote one residing at a datacenter. The success of this approach is undeniable: Dropbox passed in 2010 the milestone of 4 million registered users [4]. However, even if they are undeniably useful, these applications are not free from shortcomings. The backed up data is outsourced to a single company, raising issues about data confidentiality and risk of data loss (the case of Carbonite is emblematic [3]); indeed, companies offering a storage service do not generally offer formal guarantees about their data availability and reliability.

The most significant limitation of current on-line backup applications, though, is cost: bandwidth and storage are expensive, resulting in companies not being able to offer for free more than few gigabytes of storage space. This trend will reasonably hold in the future since datacenter costs are largely due to energy (power and cooling) and personnel costs rather than hardware costs [7].

To obtain lower costs, it is natural to look at available resources at the edge of the Internet, by exploiting unused disk space and bandwidth at end-users' machines. Several approaches propose to exploit these resources to build peer-to-peer *storage* applications, where the goal is to provide random access to individual files with a small latency and availability guarantees; these systems can also be used to backup data. However, obtaining reliable storage by exploiting resources of unreliable machines is very difficult, if not sometimes practically impossible [10].

In contrast, we explicitly focus on *backup* applications and derive application requirements tailored to this specific setting. A backup application is effective if users are able to complete backup and restore operations within a reasonable time window. Backup data is accessed only in case of end-host failures (e.g. disk crash), which require to recover the whole set of backup files.

In this scenario, it is tempting to think that a pure peer-to-peer architecture would be an ideal solution to eliminate the costs of a cloud-based backup application. However, we show that there are frequent cases in which the resources that peers contribute to the system are simply not sufficient to guarantee that all users will be able to complete their backups in a reasonable amount of time, if ever.

In this work we make the case for an hybrid approach that we call peer-assisted: storage resources contributed by peers and sold by datacenters coexist. We focus on two key elements of such a system, *data placement* and *bandwidth allocation*, and study their impact on performance measured by the time required to complete a backup and a restore operation and the end-users' costs.

The main contributions of this work are summarized in the following.

- We show that, by using adequate bandwidth allocation policies in which storage space at a cloud provider is only used temporarily, a peer-assisted backup application can achieve performance comparable to traditional client-server architectures with substantial savings.
- We explore the impact of data placement policies on system performance and fairness, and conclude that pure peer-to-peer systems may work only in particular settings and that fairness (in terms of resources obtained and contributed to the system) has a price that we measure by the monetary cost supported by end-users.
- We evaluate the effects of skewed storage demand and resource contribution, and conclude that the system archi-

tecture we propose copes well with peer heterogeneity.
- We evaluate the effects of the system scale and show that a peer-assisted backup application imposes a limited load on storage servers even when the number of peers in the system grows.
- We show that state-of-the art coding techniques used to ensure data availability at any point in a peer's lifetime impose high data redundancy factors, which can be lowered without affecting in a sensible way the ability of peers to restore their data in case of a failure.

The remainder of the paper is organized as follows. In Sec. II we overview the literature on P2P storage applications and emphasize the key differences with respect to our work. In Sec. III we present an overview of our system, the assumptions we made and the techniques we developed in this work. Sec. IV illustrates our methodology and evaluation set-up, and Sec. V summarizes our main results. We conclude in Sec. VI.

II. Related Work

Most of the research in peer-to-peer storage focuses on the design of general-purpose storage systems that provide most of the features usually given by traditional file systems. A significant amount of work has been devoted to implementing systems with low latency, consistency guarantees for multiple readers and writers, elaborate security policies, efficient data look-up, and anonymity for data publishers and readers; for a review of such solutions we refer to [15], [13, Chapter 2].

In a backup system, many of the aforementioned issues can be ignored or solved easily. Access latency is largely not an issue since backup data is only read in case of a restore operation. Additionally, efficient techniques to perform fast look-up of individual backup files are unnecessary: a restore operation can be completed with the simple knowledge of remote locations where backup data has been stored. Data can be assumed to have a single owner authorized to read and write it, hence access control can be achieved with standard cryptographic techniques.

Many works devoted to peer-to-peer backup target the almost Herculean task of backing up the entire contents of a hard drive, including operating system files. These works propose *convergent encryption*, a technique to achieve data summarization that avoids storing multiple times pieces of data that are common to many users [8], [11], [16]. In the context of our work, whereby users specify a subset of their important data to backup, the number of overlapping data fragments that could be summarized is plausibly very little. In addition, convergent encryption is susceptible to various attacks on data privacy [23].

Wuala [6] is a peer-assisted social storage service. In Wuala, a full copy of the data is stored on a central server while (encoded) fragments are "cached" on peers in order to save on server bandwidth costs [14]; conversely, in our architecture, since *long term storage costs* dominate the costs of bandwidth, we aim at reducing the amount of data stored in the central server by using storage capacity of other peers.

FS2You [20] is a peer-assisted system that provides temporary storage and seeding for files in a BitTorrent-like content distribution system. FS2You does not guarantee data persistence; while its goal is to minimize bandwidth costs, we focus instead on minimizing the storage costs that will be dominant in the long run for a storage system.

A lot of attention has been devoted to the study of incentive mechanisms for P2P storage applications [12], [18], [16]. In general, the idea is to impose system fairness in terms of storage resources: any peer should offer an amount of local storage space proportional to the load imposed on the system. Previous works [21], [19], [22] show that it is possible to design embedded incentive mechanisms for peers to contribute resources to the system without requiring additional components such as virtual currency or reputation-based schemes. In this work, we go beyond discussing the feasibility of such approaches and study the effects a symmetric selective data placement policy [22] on system performance and costs.

III. System Overview

In this Section we present the design of our peer-assisted backup application. We discuss the assumptions we make throughout this paper and outline our approach to solve the problem of data availability. We then set off to explore the key issues of a peer-assisted backup system: *i)* how to allocate bandwidth and schedule upload slots and *ii)* how to select remote locations that store data fragments.

In this work we compare our peer-assisted design with pure *peer-to-peer* and *client-server* architectures in which backup data is uploaded, respectively, only to peers and to a storage server. For simplicity, in the remainder of the paper we refer to a single storage server, e.g. operated by an individual storage cloud provider. In practice, our approach accepts multiple storage servers, *i.e.*, backup data can be uploaded to more than one cloud storage provider. We take a trace-driven, simulation-based approach to evaluate and compare the performance of these alternative architectures: our methodology allows experiments with realistic inputs which take into account a fine-grained representation of the mechanisms we designed.

A. System and Application Assumptions

In this work we build upon the approach taken by Dropbox [2], and assume users to specify one or more local folders containing important data to backup. We also assume that data selected for backup is available locally to a peer. This is an important trait that distinguishes backup from storage applications, in which data is only stored remotely. As a consequence, *data maintenance*, *i.e.*, making sure that a sufficient number of data fragments are available at any point in time and reacting by generating new fragments when remote peers fail or leave, is greatly simplified.

We assume peers to contribute with non-negligible storage capacity to the system, with ADSL-like bandwidth capacity, and several hours of continuous uptime per day. As we show in our results, nodes contributing with too little resources either exact a high toll in terms of storage capacity of other peers or, when incentive mechanisms are in place, they are not able to sustain by themselves a working system and require the presence of server-based storage.

In this work, we assume the datacenter hosting the storage service to offer ideal reliability and availability guarantees and to charge end-users for bandwidth and storage.

Furthermore, we assume the presence of a centralized component, similar in nature to that of the "tracker" in the BitTorrent terminology. The *Tracker*[1] is in charge of membership management, *i.e.*, it maintains a list of peers subscribed to the backup application. Hence, the tracker can bootstrap a new peer with a list of other peers susceptible to store her backup data. The Tracker also implements an additional component used to *monitor* the online behavior of a peer: the list of peers in the system is enriched by a measure of the fraction of time a given peer is online.

B. Data Availability

The problem of storing data on remote peers with an intermittent online behavior has received ample attention in the past [9]. When data is lost, due to an unpredictable event such as a disk crash, it is important for a peer to be able to restore it, and that the time to complete this operation to be short. For this reason, data availability, *i.e.*, the probability to recover enough fragments to reconstruct the original data from remote peers at any point in time, needs to be achieved and maintained throughout a peer's lifetime.

In our work, peers store their data in encrypted backup objects of a fixed size S. Backup objects are divided in k fragments of size S/k, that we label *original blocks*. Before uploading data fragments to remote peers, the original k blocks are encoded using erasure coding: we call these new fragments *encoded blocks*. A peer may upload s original blocks to the storage server, which is assumed to be always online.

Hence, the number of encoded blocks to upload to remote peers can be derived as follows. We consider the probability of each peer being online as an independent event with probability $\overline{a}$ (termed *peer availability*), and we aim for a data availability target value t. Therefore, we store on remote peers the number of fragments $p(s)$ defined as:

$$p(s) = \min\left\{ x \in \mathbb{N} \;\middle|\; \left(\sum_{i=k-s}^{x} \binom{x}{i} \overline{a}^i (1-\overline{a})^{x-i} \right) \geq t \right\} \quad (1)$$

The *redundancy rate* $r = \frac{s+p(s)}{k}$ represents the ratio between the quantity of data stored in the system (remote peers and storage server) and the original size of unencoded data. Suppose that a peer decides to store backup fragments on a remote server only: in this case, $p(s) = 0$ and $s = k$, thus the redundancy factor $r = 1$. Instead, assume a peer to store backup data on remote peers only: in this case $s = 0$ and p can be derived by the normal approximation to the binomial in Eq. 1 (see for example [9]). Our bandwidth allocation policy, described in the next Section, allows to explore the space between the two extreme cases illustrated above.

We note that, for Eq. 1 to hold, encoded fragments must be stored on *distinct remote peers*, because otherwise the failure of a single machine would imply the contemporary loss of many fragments, thereby resulting in a higher probability of data loss. On the other hand, since the datacenter is considered to be always online, any number of fragments can be stored on the data center.

Since we assume every peer in the system to hold a local copy of the data that is backed up, we simplify *data maintenance* by letting peers take care of regenerating and uploading a new encoded block whenever a remote peer holding their data crashes.

C. Bandwidth Allocation

In this work, we target typical users that connect to the Internet through ADSL: upload bandwidth is a scarce resource that calls for bandwidth allocation policies to optimize its usage. Upload capacity is used to back up local data, for data maintenance and for serving remote requests for data restore.

In our system, a bandwidth scheduler is triggered at regular intervals of time. Restore slots are given the highest priority to ensure that crashed peers are able to recover their data as soon as possible. Backup slots are treated as follows. By default, we employ an *opportunistic* allocation that prioritizes uploads to online peers rather to the storage server, with the goal of saving on storage cost. When multiple slots that satisfy this constraint are available, we prioritize pending fragment uploads that are closest to completion. As an alternative, we also study the effects of a *pessimistic* allocation aiming at minimizing the time to backup data: in this case, all the upload slots are devoted to send backup fragments to a storage server.

Since remote peers exhibit an intermittent online behavior, our bandwidth allocation aims at completing as soon as possible the transfer of data fragments to remote peers (both in restore and backup operations). Hence, we dedicate the whole[2] capacity to a single upload slot; if a single data transfer does not saturate the upload bandwidth and the backup operation is not finished, the surplus is used to transfer backup fragments to the storage server. A similar technique has also been proved effective in P2P content distribution applications [17].

A data backup operation is successful when s fragments have been uploaded to the storage server and $p(s)$ fragments, as defined in Eq. 1, have been uploaded to remote peers, ensuring the required target data availability. For example, assume $k = 32$ original blocks of which $s = 15$ are stored on the storage server, and $x = 20$ encoded blocks are currently stored on remote peers. If $x \geq p(s)$, the backup operation is considered successful. Otherwise a new backup fragment is uploaded to a remote peer or to the storage server: in the first case, the number of encoded blocks becomes $x = 21$; in the second case, the number of original blocks stored on the server becomes $s = 16$, resulting in a lower value for $p(s)$. This process continues until x reaches $p(s)$.

Since long-term storage on a server is costly, we introduce an *optimization phase* that begins after a successful backup: peers attempt to offload the storage server by continuing to upload additional encoded blocks to remote peers. Storage servers are used as a *temporary storage* to meet the availability

[1] In practice, a tracker can be easily distributed using a DHT approach.

[2] In practice, to use the full nominal rate of the uplink, one must also consider some under-utilization introduced by TCP's congestion control mechanism.

target as soon as possible. Once the number of additional encoded blocks stored on remote peers reaches the $p(s)$ value of Eq. 1, the used storage space on the server gets gradually reclaimed. In practice, during the optimization phase a random backup fragment stored on the server is flagged for deletion; subsequently, a peer uploads one encoded block to a remote peer and checks if the number of remotely-stored fragments x is at least $p(s-1)$. If this condition is satisfied, the original block marked for deletion can be safely removed from the server, otherwise the upload of encoded blocks to remote peers continues until x reaches $p(s-1)$.

Data maintenance operations work as follows: once the local peer detects a remote peer failure, the number of remotely stored fragments x has become lower than $p(s)$. This is equivalent to a situation where the backup is again not completed, and handled according to the upload bandwidth allocation policy. When using the opportunistic strategy, a new backup fragment is generated locally and is re-scheduled to be uploaded to a (possibly different) remote peer, or to the remote server if no remote peer is available. With the pessimistic strategy, a new fragment gets transferred immediately to the storage center, and possibly reclaimed afterward during the optimization phase.

Our download bandwidth allocation policy prioritizes data restore operations on crashed nodes to storing backup fragment for other peers. Indeed, a restore operation is critical as data fragments on remote peers can get lost in case of failures.

Download bandwidth allocation depends on whether data fragments are downloaded from the storage server (*i.e.*, during restore operations) or from remote peers. In our work we assume that restoring one or more fragments from a storage server saturates the downlink of a peer. When data fragments are downloaded from remote peers we avoid over-partitioning the downlink of a peer by imposing a limit on the number of parallel connections: as a consequence the risk of very slow data transfers is mitigated.

D. Data Placement

Data placement amounts to the problem of selecting the remote location that will store backup fragments.

In many P2P storage systems, data fragments are randomly placed on remote peers using a DHT-based mechanism. Since our focus is on backup applications, a look-up infrastructure is not needed. Indeed, a restore operation requires locating a sufficient number of fragments to obtain the original data, hence the only information a peer needs is the list of remote peers currently storing backup fragments. This information is provided by the *Tracker*.

In our work, we study the impact on system performance of two data placement policies:

- *Random*: backup fragments are placed on random remote peers with enough available space and that are not already storing another backup fragment for the same peer.
- *Symmetric Selective*: peers adopt a "tit-for-tat"-like policy and accept to store a (single) fragment for a remote peer only if reciprocity is satisfied. In addition, peers are partitioned into "clusters" depending on their online behavior. Peers upload backup fragments exclusively to remote peers within the same cluster.

As discussed in Sec. II, the quest against "free-riders" can be pursued with complex mechanisms based on reputation and virtual currency. Previous works [21], [19], [22], have studied alternative incentive schemes for P2P storage applications with selfish peers and have shown, with game theoretic tools, the existence of equilibrium strategies motivating users to increase their online time and local storage dedicated to the system.

Inspired by those works, we implement the Symmetric Selective data placement policy for its simplicity and because it does not require any additional layer to enforce peer cooperation. When the Symmetric Selective policy is used, data availability is computed using a modified Eq. 1: the average online availability of peers is computed for each cluster. Clustering based on the online availability of a peer is performed by the *Tracker*, which constantly monitors the intermittent behavior of the subscribers of the backup application.

IV. Evaluation Methodology

In this Section we describe the methodology we used to evaluate our peer-assisted backup system and compare it to alternative schemes based on purely centralized and P2P designs respectively. For brevity, in the remainder of this paper we label the three backup applications as follows: *DC* stands for the centralized approach, *P2P* indicates the peer-to-peer application and *PA* stands for the peer-assisted design.

We implemented a custom, flow-level simulator based on a fluid model of TCP in which different (eventually) competing flows receive equal shares of a link's capacity. Network connectivity is modeled at the access level only: an end-to-end link between two peers has uplink and downlink bottlenecks while no network bottlenecks are simulated.

Next, we describe the two main datasets we used to characterize the peers in our system: the bandwidth distribution and the online behavior datasets.

Bandwidth distribution: uplink and downlink capacities of peers in our system are obtained by sampling a real bandwidth distribution collected over more than 300,000 unique Internet hosts for a 48 hour period from roughly 3,500 distinct ASes across 160 countries[3]. Fig. 1(a) depicts the cumulative distribution of uplink capacity we randomly assigned to peers; downlink capacity is derived by the uplink distribution scaled by a constant multiplicative factor equal to 4, which is representative of a typical asymmetric residential Internet access.

Online behavior: monitoring the raw online activity of Internet hosts is a difficult operation. In this work we focus on a specific application that we deem relevant for our context. We used traces collected over a period of more than 6 months for roughly 10,000 users of a popular Internet messaging application in Italy[4]. Fig. 1(b) illustrates the cumulative distribution of peer availability computed as the number of hours per day a user is online, where we filtered sporadic users with low

[3]Raw measurement data has been collected using an opportunistic technique that exploits control messages of the BitTorrent protocol. We thank M. Piatek for providing the original trace.

[4]One of the authors of this work is an administrator of the messaging server and has access to user activity logs.

availability values. For example, a user online for 6 hours per day will have 0.25 availability.

In this work, the availability a assigned to a peer is obtained by randomly sampling the distribution in Fig. 1(b). Online and offline periods of a peer with availability a are instead taken randomly from exponential distributions with expected lengths respectively $24a$ and $24(1-a)$ hours, yielding an average of one connect/disconnect cycle per day; at the beginning of the simulation each peer is considered to be offline.

Peer deaths: we simulate peer deaths as follows. In our system a peer is considered dead when her hard-disk crashes. Such peers never leave definitively the system: the underlying assumption is that those peers will come back online to recover their data. Disk crash events happen within a period of time defined by an exponential distribution with expected time equals to the simulation duration; after a disk crash, peers stay online until the restore operation is completed, then they return to their usual online behavior. The rate of peer deaths is chosen to be extremely stressful for our system: in average a peer will crash at least once during a simulation run.

The storage server: in this work, a storage server is assumed to be an element of a datacenter, exposed to Internet users through a simple (e.g. HTTP) interface allowing to put, get and delete data objects. This is the case for a cloud storage service such as Amazon S3 [1]. In our simulations we model an ideal storage server hosted in an over-provisioned datacenter: storage space and uplink/downlink bandwidth are unlimited. Availability and durability of data stored in such servers are also ideal: the storage server is never offline and never fails.

A. Simulation parameters

We now define the simulation parameters we used in our evaluation. It is outside of the scope of this paper to examine the whole parameter space, hence we include here a set of parameters yielding the most insightful results. Simulation results correspond to a collection of 10 simulation runs.

Our simulations cover a period of 97 days. We simulate a system composed of 870 peers, each requesting to backup a total of 10 GB of data. Each backup object is divided into $k = 32$ fragments before applying redundancy, resulting in a fragment size of 320 MB. In this work we assume that peers dedicate 50 GB of local space to host backup fragments of remote peers.

The parameters that govern the behavior of *bandwidth allocation* is set as follows: scheduling decisions are taken at regular intervals of 5 minutes in order to always have decisions based on a recent sample of peer availabilities. The number of parallel download slots that are activated by a peer is computed according to the following expression: $\max\left(4, \left\lfloor d_i/\overline{d} \right\rfloor\right)$, where d_i is the downlink capacity of a peer and $\overline{d}$ is the average download capacity of the system.

With regard to *data placement*, the symmetric selective policy requires the definition of cluster granularity. When this placement strategy is used, peers are clustered according to their availability, *i.e.*, the fraction of time they spend online. The granularity of each cluster is obtained by rounding the value of peer availability by multiples of 0.1.

Availability	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Red. rate	4.87	3.55	2.76	2.22	1.83	1.53	1.28
Cluster size	554	108	66	36	28	22	56

TABLE I
REDUNDANCY RATES FOR SYMMETRIC SELECTIVE DATA PLACEMENT.

Finally, *redundancy rates* are computed as follows. The data availability target is set to $t = 0.99$. When the random data placement policy is used, the redundancy rate is computed based on the system-wide average peer availability. In this case, the redundancy rate is set to $r = 3.47$. Instead, the symmetric selective data placement requires to compute a per-cluster redundancy rate, which is summarized in Tab. I when $s = 0$. Additionally, we include information on the number of peers that belong to each cluster, which is computed according to the distribution of peer availabilities illustrated in Fig. 1(b).

B. Performance Metrics

Prior works on P2P storage usually evaluate system performance in terms of access latency of individual files. In this work we set apart from such metrics and measure the overall time required to complete a backup and a restore operation on the whole user data, and the costs for long term storage.

Formally, we define:

- *Time To Backup* (TTB) is the time needed to complete the backup of a backup object. A backup is complete when the number of (possibly encoded) fragments uploaded to remote peers x and to the storage server s is sufficient to satisfy the inequality $x \geq p(s)$.
- *Time To Restore* (TTR) is the time required to download at least k backup fragments that are sufficient to reconstruct the original backup object. We measure the TTR only for peers that completed the backup operation.
- *Costs* accounts for the resources used by a peer when uploading and storing backup fragments in the storage server. Storage dominates bandwidth costs for long-term storage; moreover, our system only uses the inbound bandwidth of the storage server, which is often at low cost or free [1]. For these reasons, in the following we will neglect bandwidth costs.

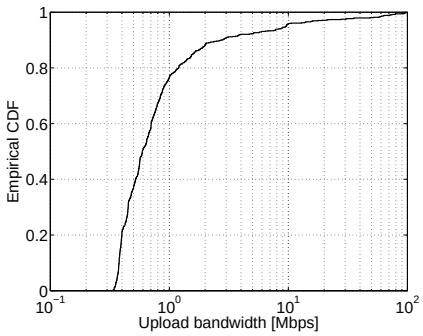
V. RESULTS

A. General Overview of Online Backup Systems

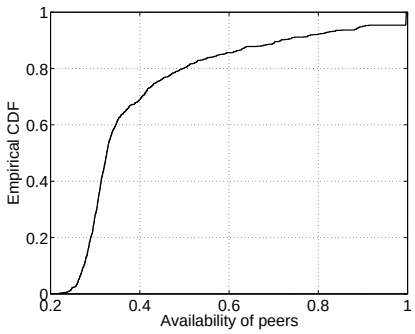
Fig. 2 overviews the performance of three alternative approaches to online backup applications through the lenses of the performance metrics defined in Sec. IV-B.

We compare a legacy client-server application where users store their data solely on a storage server (labeled DC, which stands for datacenter), a P2P application in which the only storage resources available are those contributed by the peers (labeled P2P), and the peer-assisted backup system we discuss in this work (labeled PA). We include both the opportunistic and the pessimistic allocation policies illustrated in Sec. III-C, and use the *random data placement* policy.

Fig. 2(a) illustrates the cumulative distribution function (CDF) of TTB values. Clearly, the minimum TTB is achieved

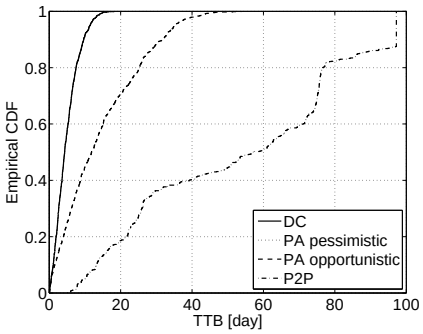


(a) Upload bandwidth distribution.

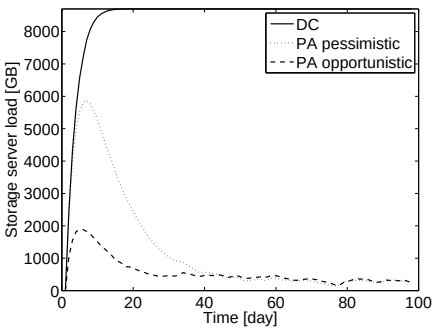


(b) Peer availability distribution.

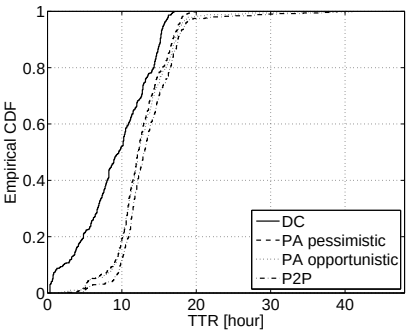
Fig. 1. Overview of the datasets that describe peer characteristics in terms of access bandwidth and online behavior.



(a) TTB.



(b) Storage costs.



(c) TTR.

Fig. 2. General overview of the performance of three approaches to online backup applications. A peer-assisted design approaches the performance of a server-based solution and achieves small long-term storage costs.

by the DC application. Indeed, the redundancy factor is $r = 1$, hence only k backup fragments are uploaded to a central server, which is always available and has infinite inbound bandwidth. Note that also the peer-assisted application with pessimistic allocation (PA-pessimistic) achieves a minimum TTB for the simple reason that, initially, all original backup fragments are uploaded to the storage server (the line for the two cases coincide in Fig. 2(a)).

The peer-assisted application with opportunistic allocation (PA-opportunistic) achieves longer times to backup. Indeed, $s+x(s) > k$ backup fragments are uploaded to remote peers to cope with their online behavior. Recall that s is the number of original blocks initially uploaded to the central server, before the optimization phase begins.

The P2P application obtains the worst results in terms of TTB. More than 15% of peers cannot complete the backup operation, according to the definition we give in Sec. IV-B, within the simulation time. The very large TTB values are due to two reasons. Firstly, the redundancy factor that meets the target data availability is large and so is the amount of (redundant) backup data to be uploaded to remote peers. Secondly, the intermittent online behavior of peers may interrupt data uploads: the bandwidth scheduler enters a time-out phase when no online remote peer is available to receive data.

Fig. 2(b) depicts the storage costs expressed as a time-series of the aggregate amount of data located in the storage server[5]. The area underlying the curves, multiplied by monetary costs, and normalized by the simulation time, is an indication of the aggregate monetary costs end-users must support.

We observe that in the DC application, the amount of storage requested on the datacenter quickly reaches the maximum value. In the PA application, the cost grows as peers upload their fragments to the storage server to complete their backups, and gradually diminishes during the optimization phase. Lower TTB values in the PA-pessimistic case are counterbalanced by higher aggregate costs on the datacenter. On the long run, however, the storage load on the server is very low and settles to the same value for both opportunistic and pessimistic allocation. Storage costs do not settle to zero because of the maintenance operations due to peer deaths.

Finally, Fig. 2(c) shows the CDF of the TTR metric. We observe that the TTR is much lower than the TTB in all different backup applications. Again, the minimum TTB is achieved in the DC case since the storage server is always online and the downlink capacity of a peer is fully utilized. While retrieving data stored on remote peers entails a longer TTR, this quantity remains well within a day in the majority of cases.

Now, we can draw a first important conclusion: by using adequate allocation policies in which a storage server is

[5]Storage costs for the P2P application are zero, hence we do not report them on the figure.

only used temporarily, a peer-assisted backup application can achieve performance comparable to traditional client-server architectures at much lower costs. Our results also show that a P2P application, despite being free of charge, can meet a reasonable performance only for a small fraction of peers.

B. Data Placement and Cost of Fairness

We now focus on the impact of the data placement policy on performance, and compare the DC, PA and P2P backup applications. The DC case is shown for reference, since backup fragments are constrained to be stored on the storage server.

First, we study the *random data placement* strategy. Although remote peers are randomly selected, we show results by the availability class of a peer, *i.e.*, peers are grouped depending on their online behavior as discussed in Sec. IV-A.

Fig. 3(a) and Fig. 3(b) represent the median TTB as a function of peer availability class, with and without peer deaths (because of disk crashes) respectively. Clearly, TTB is correlated to the availability class: higher online times entail lower TTB. Comparing the case with and without peer deaths reveals the sensitivity of the P2P approach to data maintenance traffic. Instead, the PA application tolerates well peer deaths because the storage server helps in speeding up repairs.

Fig. 3(c) shows the amount of data stored on peers: for each availability class, the left boxplot is for the PA case and the right boxplot is for the P2P case. Random data placement introduces unfairness: indeed, peers with a larger online time are more likely to be selected as remote locations to store backup fragments and their excess capacity is exploited by peers with low availability. This result motivates the adoption of the *symmetric selective* data placement policy, whose goal is to impose system fairness as explained in Sec. III-D.

Fig. 4(a) shows the CDF of the TTB for PA and P2P applications. For the PA application, the TTB achieved by highly available peers decreases, whereas peers with low availability experience a slightly increased TTB. This trend is more pronounced for P2P applications. Not only the TTB can be very large, but a substantial fraction of peers cannot complete their backup operation: this happens for more than 15% of the cases with a random data placement and for almost 80% of the cases with the symmetric selective policy.

Imposing fairness in a PA application modifies the costs for end-users, as illustrated in Fig. 4(b). When the symmetric data placement policy is used, peers with low availability cannot exploit the excess capacity in the system and are compelled to store their data on the storage server.

In summary, we justify the system-wide loss in performance with the notion of *cost of fairness*. When peers are constrained to store backup fragments on remote peers with similar online behavior and are compelled to offer an amount of local storage space proportional to the amount of (redundant) data they inject in the system, the excess capacity provided by highly available peers cannot be exploited. In a P2P application, peers can suffer a severe loss in performance or eventually cannot complete their backups. Instead, in a PA application system fairness translates into higher storage costs for peers with low-availability, but system performance is only slightly affected.

C. Impact of Fragment Size

As discussed in Sec. III-B, data availability is achieved through erasure coding, which is a technique adopted both by the P2P and our PA backup application. Backup data is split into k equally sized blocks: s blocks are stored on a server and $p(s)$ blocks are uploaded to *distinct* remote peers. In this section we analyze the impact of block size. Fig. 5 illustrates storage costs, TTB and TTR for $k = \{8, 32, 128, 512\}$, resulting in fragment sizes of $1280, 320, 80, 20$ MB respectively. Here we focus on the PA application only, when opportunistic allocation and random data placement policies are used. We obtain similar results with the symmetric selective policy, which are omitted due to space constraints.

Fig. 5(a) and Fig. 5(b) clearly illustrate the trade-off between storage costs and backup times, as a function of fragment size. On the one hand, small fragments require a large number of *distinct* remote peers with free space to hold encoded blocks. If this condition is not satisfied, opportunistic allocation allows a peer to send backup blocks to the storage server. Hence, higher storage costs for small fragments. Maintenance operations exacerbate the need for distinct remote peers to store encoded fragments: as a consequence, the optimization phase has little effects on storage costs. On the other hand, the transmission of large fragments may occupy many scheduling intervals, hence data transfers are more likely to be interrupted by the online behavior of remote peers. Only in these cases the residual uplink capacity is dedicated to transfer data to the storage server. As a consequence, backup times are longer but storage costs are drastically reduced.

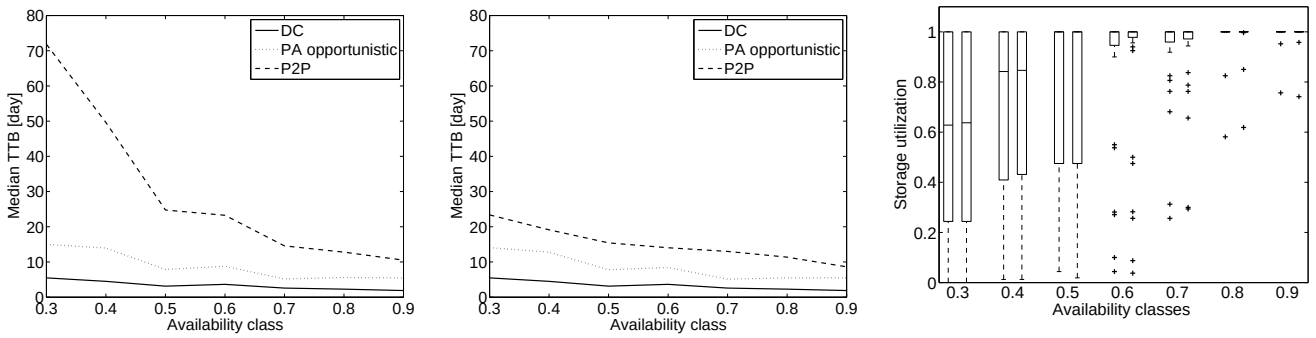
Data restore times are proportional to the fragment size, as shown in Fig. 5(c): the larger the fragment size, the longer the TTR. Larger fragments require less remote peers to store encoded blocks: as a consequence, the downlink capacity of a peer involved in a restore operation may not be saturated because remote peers may be offline.

In summary, the fragment size is a delicate parameter: its choice depends on the bandwidth distributions of peers, on scheduling decisions and on the redundancy mechanism used to achieve data availability. Our experiments indicate that extreme values achieve conflicting properties in terms of storage costs and TTB. With respect to the scenarios examined in this work, $k = 32$ blocks represent the a good compromise between costs and backup times.

D. System Scalability and Heterogeneity

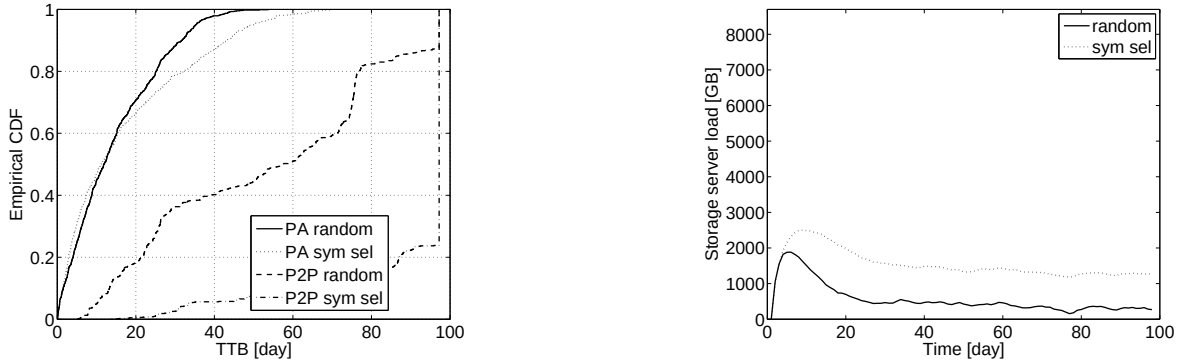
We now address two fundamental questions for an Internet application, *i.e.*, the impact on performance of the system scale and peer heterogeneity. Here we focus on the PA application only and report results for opportunistic allocation and random data placement policies.

Fig. 6 illustrates the average amount of data each peer stores in the storage server, the average TTB and the average TTR, as a function of the number of peers in the system. Fig. 6(a) indicates that as the number of peers grows, the amount of data each peer stores in the server decreases. Indeed, for larger scales, the probability of finding enough *distinct* remote peer to store backup fragments increases. Fig. 6(b) and Fig. 6(c)



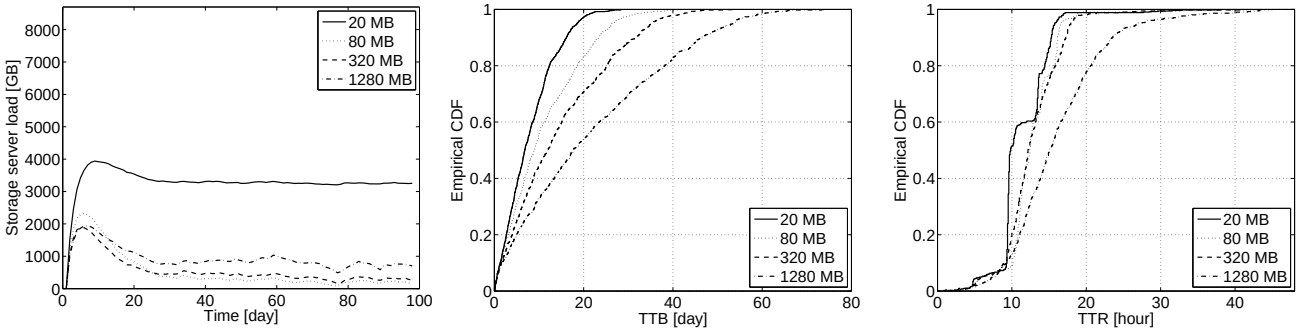
(a) TTB with peer deaths. (b) TTB with no peer deaths. (c) Storage load on peers.

Fig. 3. System performance with random data placement, in terms of median values of TTB and storage load on peers, grouped by availability classes. Note the impact of maintenance traffic due to peer deaths and the uneven distribution of storage load on peers.



(a) TTB. (b) Storage costs.

Fig. 4. Comparison between random and symmetric selective data placement. Introducing fairness has a significant impact on the performance of a P2P design, while it introduces additional costs for a PA application.



(a) Storage costs. (b) TTB. (c) TTR.

Fig. 5. Impact of fragment size on the PA application. The right choice of a fragment size depends on a trade-off between performance and storage costs.

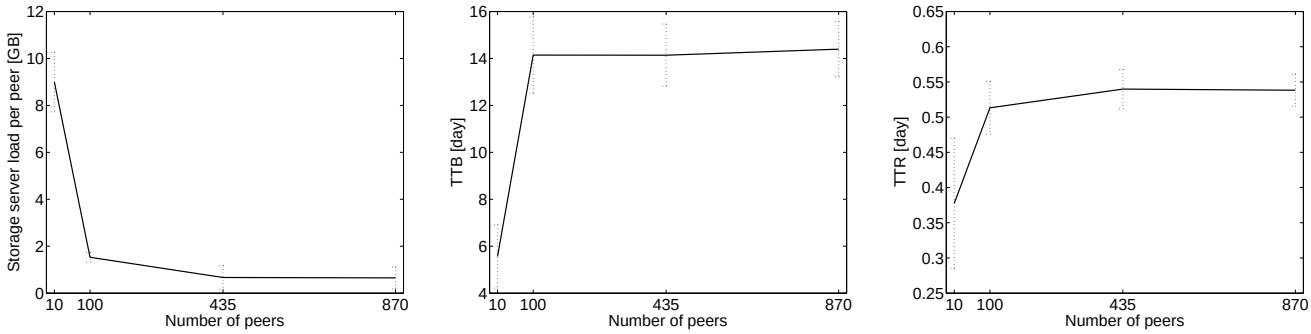
illustrate that both TTB and TTR reach a plateau as the system scale grows.

We conclude that a peer-assisted architecture is scalable: storage costs dramatically decrease while backup and restore times do not increase with an increasingly large number of peers. It is important to notice that a peer-assisted design overcomes the typical bootstrap problem of peer-to-peer systems, which require a critical mass of peers to be fully functional when the system scale is small.

We now evaluate the behavior of the PA application when peers are heterogeneous. In a first scenario, we assign a random amount of backup data to each peer, while we keep the storage capacity dedicated to the system constant and equal to 50 GB. Conversely, in a second scenario we keep a constant backup data size equal to 10 GB and randomly assign the amount of storage space each peer dedicates to others. Backup data sizes and dedicated storage space are randomly drawn from a truncated power law with exponential cutoff:

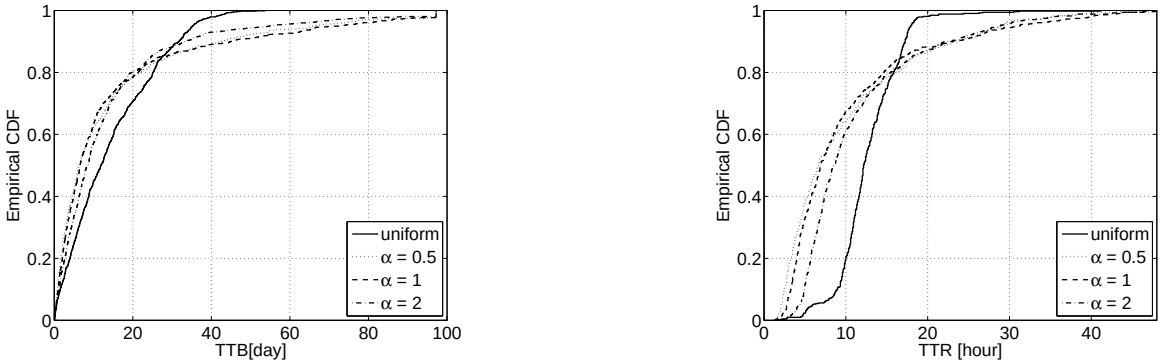
$$p(x) \propto x^{-\alpha} e^{-0.1x}$$

The values of this distribution are normalized to obtain an average backup size of 10GB and an average offered storage



(a) Storage costs. (b) TTB. (c) TTR.

Fig. 6. Analysis of the scalability of the PA application. Increasing the system size does not harm performance nor storage costs.



(a) TTB. (b) TTR.

Fig. 7. Performance of the PA architecture when peers have heterogeneous demands in terms of volume of data to backup.

capacity of 50 GB respectively. In our experiments we vary the parameter α of the distribution such as: $\alpha = \{0.5, 1, 2\}$

A skewed distribution of the backup data size implies skewed TTB and TTR values, as shown in Fig. 7(a) and Fig. 7(b). Indeed, the time to upload the necessary number of backup fragments to remote peers or the storage server is proportional to the data size. Instead, for constant backup data sizes and a skewed distribution of the amount of storage space dedicated to the system, there are no visible effects on TTB and TTR values. This is due to the presence of a central server, which absorbs the excess storage demand that cannot be satisfied by the peers, and the fact that peer have enough downlink capacity to support backup and restore operations.

E. A Note on Data Availability

The goal of an online backup application whereby data fragments are stored on unreliable peers with intermittent online behavior is to ensure data durability.

On the one hand, injecting redundant blocks in the system allows a restore operation to be successful despite a fraction of remote peers being offline. On the other hand, we have seen that backup times, storage and maintenance costs are directly related to the redundancy factor derived in Eq. 1.

The question we try and address in this Section is whether, for backup applications, a stringent data availability target can be relaxed, so as to decrease the redundancy factor applied to backup data. A small redundancy factor has several implications. The amount of data to upload to the system (storage server or remote peers) would approach the original size of a backup object, resulting in shorter backup times and lower storage costs. Moreover, few distinct remote peers would be sufficient to store backup data, benefiting a P2P approach.

Now, assume the lifetime of a peer disk (before crash) to be an exponentially distributed stochastic variable with average $\bar{t}$ (i.e., a peer crashes by time t with probability $1 - e^{-t/\bar{t}}$). Assume that a node uploads n redundant blocks to remote peers, of which k are sufficient to restore the original data, yielding a redundancy factor $r = n/k$. If maintenance is not done, data is lost when more than $n - k$ peers die. Therefore, the data loss probability can be expressed as follows:

$$\sum_{i=n-k+1}^{n} \binom{n}{i} \left(1 - e^{-t/\bar{t}}\right)^{i} \left(e^{-t/\bar{t}}\right)^{n-i}.$$

Fig. 8 illustrates the data loss probability as a function of time and redundancy rate, when the average death rate in the system is $t = 90$ days: in average, every peer crashes once during a simulation run. In the figure, each line corresponds to a different value of data loss probability. We observe that even low redundancy rates are sufficient to ensure data availability for a long period of time, which is considerably larger than the time required to complete a restore operation.

This result leads us to the following considerations. A high redundancy factor allows *lazy data maintenance mechanisms*: it is not necessary to respond promptly to a peer death since

data durability is not at stake. Even if their disks crash, peers have plenty of time to restore their data.

Alternatively, the performance of a backup application can be improved by adopting a different encoding strategy. Instead of fixing a data availability threshold using Eq. 1, it could be sufficient to introduce the necessary amount of redundant data in the system to ensure that any restore operation is completed successfully. Our current research agenda includes a different approach to data encoding based on rate-less codes combined with a mechanism to estimate the TTR based on the number of redundant blocks injected in the system.

VI. Conclusion

In this work, we made the case for a peer-assisted design to online backup applications, that complements the current landscape of P2P backup/storage applications and online storage services offered by could providers.

We showed that there is an ample space to explore between client-server and P2P architectures. On the one hand, client-server applications exhibit high performance in terms of the time required to complete backup and restore operations, which comes at a large monetary costs for long-term storage. These costs can be sunk to zero with a P2P approach, with an inevitable and severe loss in performance.

Our experiments showed that, with adequate bandwidth allocation policies in which storage space at a cloud provider is only used temporarily, a peer-assisted design can achieve performance comparable to client-server architectures at a fraction of the costs.

As we target an application for Internet users in the wild, we studied the effects of data placement strategies with embedded incentive mechanisms to foster cooperation and concluded that imposing fairness comes at a cost. For a P2P design, the price to pay is that a large fraction of users may be excluded from taking part to the system since backup and restore operations do not complete. Instead, in a peer-assisted application, the impact of system-wide fairness on performance is negligible and all peers are able to save and retrieve their data. However, peers that contribute little with local resources are compelled to pay higher storage costs.

We also covered two important aspects of our peer-assisted design: the impact of system scale and peer heterogeneity. We concluded that a peer-assisted architecture can handle a large pool of users, and that a larger system size implies lower storage costs. Furthermore, peer heterogeneity has marginal effects on system performance.

Finally, we showed that state-of-the art coding techniques used to guarantee data availability at any point in a peer's life-time impose high data redundancy factors, which can be lowered without affecting the system performance. This paves the way for alternative coding techniques based on the expected time required to restore data rather than the typical availability threshold, which we will explore in our future work. Additionally we will extend the set of results presented in this work by studying the effects of correlation in the online behavior of peers.

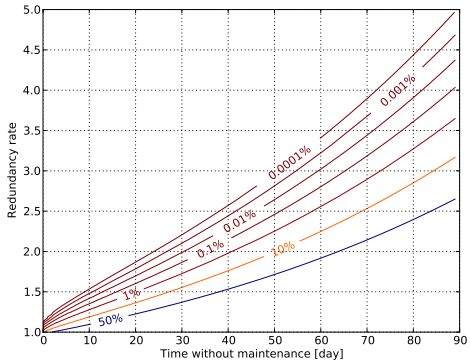


Fig. 8. Data loss probability as a function of redundancy rate. The x-axis represents the time elapsed from a successful backup operation, with artificially disabled data maintenance.

References

[1] http://aws.amazon.com/s3/.

[2] http://dropbox.com/.

[3] http://tcrn.ch/dABxRn.

[4] http://techcrunch.com/2010/01/20/dropbox-4-million-user/.

[5] http://www.apple.com/timecapsule/.

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

[7] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, University of California, Berkeley, 2009.

[8] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pstore: A secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT, 2001.

[9] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *Proc. of USENIX IPTPS*, 2003.

[10] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *Proc. of HOTOS*, Berkeley, CA, USA, 2003. USENIX Association.

[11] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making backup cheap and easy. *ACM SIGOPS Operating Systems Review*, 36:285–298, 2002.

[12] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *Proc. of ACM SOSP*, pages 120–132, New York, NY, USA, 2003. ACM Press.

[13] A. Duminuco. *Data redundancy and maintenance for peer-to-peer file backup systems*. PhD thesis, TELECOM ParisTech, October 2009.

[14] D. Grolimund. Wuala - a distributed file system. Google TechTalks video, http://www.youtube.com/watch?v=3xKZ4KGkQY8, 2007.

[15] R. Hasan, Z. Anwar, W. Yurcik, L. Brumbaugh, and R. Campbell. A Survey of Peer-to-Peer Storage Techniques for Distributed File Systems. In *Proc. of ITCC*, page 213. IEEE Computer Society, 2005.

[16] M. Landers, H. Zhang, and K. L. Tan. PeerStore: better performance by relaxing in peer-to-peer backup. In *Proc. of IEEE P2P*, pages 72–79. IEEE Computer Society Washington, DC, USA, 2004.

[17] N. Laoutaris, D. Carra, and P. Michiardi. Uplink allocation beyond choke/unchoke: or why divide does not always conquer best. In *Proc. of ACM Conext*, 2008.

[18] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *Proc. of USENIX ATEC*, page 3, Berkeley, CA, USA, 2003. USENIX Association.

[19] L. Pamies-Juarez, P. Garcia-Lopez, and M. Sanchez-Artigas. Rewarding stability in peer-to-peer backup systems. In *Proc. of IEEE ICON*, 2008.

[20] Y. Sun, F. Liu, B. Li, B. Li, and X. Zhang. FS2You: Peer-Assisted Semi-Persistent Online Storage at a Large Scale. In *Proc. of IEEE INFOCOM*, 2009.

[21] L. Toka and P. Michiardi. Brief announcement: a dynamic exchange game. In *Proc. of ACM PODC*, 08 2008.

[22] L. Toka and P. Michiardi. Selfish neighbor selection in peer-to-peer backup and storage applications. In *Proc. of Euro-Par*, August 2009.

[23] Z. Wilcox-O'Hearn. Convergent encryption reconsidered. https://zooko.com/convergent_encryption_reconsidered.html, March 2008.