École Doctorale
d'Informatique,
Télécommunications
et Électronique de Paris

Thèse

présentée pour obtenir le grade de docteur

de TELECOM ParisTech

Spécialité : Informatique et réseaux

Alessandro DUMINUCO

Redondance et maintenance des données dans les systèmes de sauvegarde de fichiers pair-à-pair

Data Redundancy and Maintenance for Peer-to-Peer File Backup Systems

Soutenue le 20 Octobre 2009 devant le jury composé de

Prof. Isabelle Demeure	**Président**
Prof. Pascal Felber	**Rapporteurs**
Dr. Philippe Nain	
Prof. Thorsten Strufe	**Examinateur**
Prof. Ernst Biersack	**Directeur de thèse**

a Piero,
che mi ha insegnato
piú d'ogni altro.

Abstract

The amount of digital data produced by users, such as photos, videos, and digital documents, has grown tremendously over the last decade. These data are very valuable and need to be backed up safely.

Solutions based on DVDs and external hard drives, though very common, are not practical and do not provide the required level of reliability, while centralized solutions are costly. For this reason the research community has shown an increasing interest in the use of peer-to-peer systems for file backup. The key property that makes peer-to-peer systems appealing is self-scaling, i.e. as more peers become part of the system the service capacity increases along with the service demand.

The design of a peer-to-peer file backup system is a complex task and presents a considerable number of challenges. Peers can be intermittently connected or can fail at a rate that is considerably higher than in the case of centralized storage systems. Our interest focused particularly on how to efficiently provide reliable storage of data applying appropriate redundancy schemes and adopting the right mechanisms to maintain this redundancy. This task is not trivial since data maintenance in such systems may require significant resources in terms of storage space and communication bandwidth.

Our contribution is twofold.

First, we study erasure coding redundancy schemes able to combine the bandwidth efficiency of replication with the storage efficiency of classical erasure codes. In particular, we introduce and analyze two new classes of codes, namely Regenerating Codes and Hierarchical Codes.

Second, we propose a proactive adaptive repair scheme, which combines the adaptiveness of reactive systems with the smooth bandwidth usage of proactive systems, generalizing the two existing approaches.

Résumé

La quantité de données numériques produites par les utilisateurs, comme les photos, les vidéos et les documents numériques, a énormément augmenté durant cette dernière décennie. Ces données possèdent une grande valeur et nécessitent d'être sauvegardées en sécurité.

D'une part, les solutions basées sur les DVDs et les disques durs externes, bien que très communes, ne fournissent pas un niveau suffisant de fiabilité. D'autre part les solutions basées sur de serveurs centralisées sont très coûteuses. Pour ces raisons, la communauté de recherche a manifesté un grand intérêt pour l'utilisation des systèmes pair-à-pair pour la sauvegarde de donnés. Les systèmes pair-à-pair représentent une solution intéressante grâce à leur capacité de passage à l'échelle. En effet, la capacité du service augmente avec la demande.

La conception d'un réseau de sauvegarde de fichiers pair-à-pair est une tâche très complexe et présente un nombre considérable de défis. Les pairs peuvent avoir une durée de connexion limitée et peuvent quitter le système à un taux qui est considérablement plus élevé que dans le cas des systèmes de stockage centralisés. Notre intérêt se concentre sur la manière de fournir efficacement du stockage de données suffisamment fiable en appliquant des schémas de redondance appropriés et en adoptant des bons mécanismes pour maintenir une telle redondance. Cet effort n'est pas négligeable, dans la mesure où la maintenance du stockage de données dans un tel système exige des ressources importantes en termes de capacité de stockage et de largeur de bande passante.

Notre contribution se porte sur deux aspects.

Premièrement, nous proposons et étudions des codes correcteurs pour la redondance capables de combiner l'efficacité en bande passante de la réplication à l'efficacité en stockage des codes correcteurs classiques. En particulier, nous présentons et analysons deux nouvelles classes de codes: Regenerating Codes et Hierarchical Codes.

Deuxièmement, nous proposons un système de réparation, nommé "adaptive proactive repair scheme", qui combine l'adaptabilité des systèmes réactifs avec l'utilisation régulière de la bande passante des systèmes proactifs, en généralisant les deux approches existantes.

Acknowledgements

This thesis is the result of more than three years spent at EURECOM. During this time I had the chance to interact with a lot of people, each of them gave her contribution to my accomplishments, and I feel obliged to thank them all.

The first place must be occupied by my advisor Prof. Ernst Biersack, not because of his position, but because he is without any doubt the person who contributed most to the success of my Ph.D. He gave tireless feedback to my ideas and my proposals, he helped me find the right research directions, he has been supportive when results did not arrive, and he showed appreciation when results arrived. I feel that I grew up a lot as a researcher in these three years and I owe a lot to him. I had a great opportunity to work with you and I want to thank you for everything you taught to me.

I want to thank all the people who are or have been at EURECOM during these years. It has been a pleasure and helpful to work, chat and exchange ideas with them in lunch breaks or in front of our beloved espresso coffee machine. Some of them, however, deserve a special mention. Taoufik En-Najjary was extremely helpful in formalizing my ideas, especially at the beginning, thanks to his precious competence in statistics related topics. He was co-author of my first publication as a Ph.D. student and I really appreciated his collaboration. I am really grateful to Damiano Carra, who was the person who helped me the most after my advisor. I went countless times to his office to explain my ideas, my doubts and my questions and he was always available to listen and to give precious hints. Finally, I want to thank Pietro Michiardi, with whom I had the opportunity to discuss about my topic and to exchange ideas and suggestions. A special thanks also to Matteo Dell'Amico and Erik-Oliver Blass, who have been brave enough to read and comment the preliminary version of this manuscript.

My thesis was partially supported by the "Microsoft Research European PhD Scholarship" and I thank Microsoft Research for the financial support that made my research possible. In particular I want to thank Fabien Petitcolas, who organized the scholarship and offered the opportunity to participate to two summer-schools, which enriched my skills and my vision of the research. His work was just perfect from any point of view.

I also got the opportunity to spend three months at Microsoft Research in Cambridge as an intern, where I had the chance to meet and work with a lot of smart and skilled researchers. I really enjoyed this experience and I want to thank especially Christos Gkantsidis, who supervised me during my internship and has been my *second advisor*, he guided me in my research and in the *hard life* of distributed system designers.

My work would have not been the same if my housemates were not there. So thanks to Corrado, Marco and Xiaolan for having shared all the "joys and sorrows" of being a PhD student in the French Riviera. Living, eating and fighting with you has been really a priceless experience.

As any good Italian, I express all my gratitude to my two families. My *natural* family, which supported me at any step of my Ph.D., and my second family that I had as student in Turin, which has been always present, in the best and in the worst days of the last eight years.

Finally, I need to thank all my friends from Sapri, my hometown in Italy. I cannot say that they contributed directly to this thesis, since they have always been a good reason for giving up and run back home, but I have to admit that without them I would not be the same person, and this thesis would definitely be worse.

Grazie a tutti!
Alessandro

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Motivation

1.1.1 Need for user data backup

The amount of data produced in the world is increasing at an incredible rate. A study in 2003 [71] estimated that about 5 exabytes ($5 \cdot 10^{18}$ bytes) of original data were produced in 2002, showing an increase of more than 30% over the previous year. Part of this tremendous growth has been driven by the digital data produced by users. Fig. 1.1 (Source: [71]) focuses on hard disks installed on user PCs and shows that the amount of *original* digital data yearly stored on PCs by users has increased by two orders of magnitude from 1996 to 2003. User data proliferation is in large part due to the digitalization of information: digital photos, digital videos, and electronic mail became part of everyday life of all computer users.

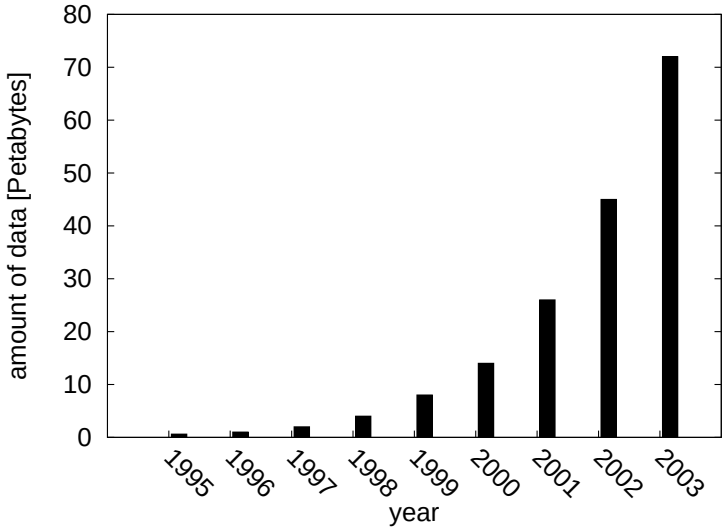


Figure 1.1: Amount of digital data produced yearly by users and stored on hard disks in Petabytes (10^{15} bytes).

The durability of these very valuable and mostly irreproducible data is jeopardized by several threats, such as media failures, software bugs, malwares, user errors, and theft [25].

All this said, it is evident that data backup has become an urgent need for users. Obviously, data backup is not a new topic: Companies have been facing the problem of data loss for decades and are equipped with data backup systems. However, the costs sustained by companies for these systems are not affordable for ordinary users, which need to find cheap solutions to store their data safely.

1.1.2 Existing solutions and their limits

The data backup solutions that ordinary users can afford generally belong to one of the two categories described in the following.

Removable devices and optical supports

It is a very common practice to put a backup copy of important data on pluggable storage devices, such as external hard-drives and flash memories or on optical media, such as DVDs and CDs. The main shortcomings of these solutions reside in:

- **Complexity of operations**: burning DVDs or copying data on external hard-drives can be a tedious task and is far from being automatic.
- **Lack of availability**: while it is a common belief that data stored on CDs and DVDs are safely recorded for ever, they can instead suffer from data loss much more frequently than we would expect, especially if not handled with care [13]. To obtain long-term durability, data stored on external media must be checked and refreshed on regular basis, making data management even more complex. Moreover, these media are normally stored in the same place where the primary copy of data is stored and are very likely lost if external disasters occur (fire, theft, etc.).

Online Storage

Outsourcing data backup to online services is another popular solution. Initially born in the '90s as a service for companies [59], it is now also proposed to ordinary users, thanks to the spread of the Internet access.

With the advent of cloud computing big players, like Microsoft with SkyDrive [11] and Amazon with S3 [3], are offering online storage services. A number of smaller companies, such as iBackup [9], mozy [12], carbonite [4], Allmydata [1] and iDrive [10], offer online backup services.

These services partially overcome the shortcomings of the methods based on external media mentioned in the previous section:

- Data backup can be completely automatic.

- The process of checking and refreshing data is transparent to the user.
- External disasters occurring to the user PCs do not impact the online stored data.

However, the adoption of outsourced data backup is not a *panacea* for a number of different reasons:

- There are concerns on data privacy. Users must trust a single party, which stores their data and can potentially violate data confidentiality.
- Users do not have any control on how data are handled and kept durable, while failures and data losses occur also in these systems.
- These services have a cost that not all users are willing to pay. Online backup prices are currently around 5$ to 10$ a month for an amount of storage that ranges from 10GB to an unlimited volume.

1.1.3 Peer-to-peer as the solution

As argued in the previous sections, there is an increasing need for user data backup solutions, and the existing solutions have serious limitations in terms of reliability and costs. These two factors motivate the investigation of new ways to do user data backup.

An idea that has been around in the research community for several years is the development of storage systems and in particular of data backup systems based on peer-to-peer technology.

The traditional architecture of distributed applications follows the client-server paradigm, which consists of two distinct entities:

- **The server**, which provides the service and all the resources needed for the service.
- **The client**, which uses the service and exploits the resources provided by the server.

The main characteristic of peer-to-peer networks is the fusion of these two roles. In a peer-to-peer network all the peers play the role of both, the server and the client: every peer contributes to the service sharing part of its resources and, at the same time, it is a customer of the service. Moreover, peer-to-peer networks differ from a client-server architecture by their self-organization, which means that usually they do not need a centralized administration. The peer-to-peer approach empowers the system with two intrinsic and very appealing properties:

- **Self-scaling**, which means that the amount of resources increases along with the demand. This is a tremendous advantage over the client-server approach, where the server needs to be dimensioned to sustain all the possible clients and must be upgraded whenever the service demand increases.
- **Fault-tolerance**. In the client-server applications, the server represents a single point of failure and resiliency can be provided only using multiple servers, as in the case of mirroring, or equipping the server with very reliable and expensive hardware. The de-

centralized organization of peer-to-peer networks, instead, exploits the independence of the different peers to provide a good level of reliability.

In the case of storage systems a peer-to-peer approach means that each peer dedicates part of its storage space to the community; in exchange the community (i.e. other peers) will reliably store data for this peer. A very trivial example of peer-to-peer data backup could be represented by two peers that store each the other's data, providing mutually a backup service.

A peer-to-peer data backup solution can be seen as a special case of an online backup service, which overcomes the problems we outlined in the previous section:

- Data are usually distributed over many different peers, which means that users do not need to trust a single entity. A malicious entity must gain the control of many independent peers to get access to the data of a specific user.
- The way data are organized and handled is publicly known by all the participating peers.
- A peer-to-peer service is potentially free. The costs of such a service are shared among all the participants.

1.1.4 Enabling technology trends

A peer-to-peer storage system can be realized only if participating peers are equipped with spare storage resources. Fig. 1.2 (Source: [8]) depicts the evolution of the capacity of hard disks during the last 30 years, showing an exponential growth of storage capacity. This growth is not associated with a corresponding growth in price, which implies an exponential decrease of the cost per byte stored. The availability of cheap storage suggests that users can easily have a lot of storage space, which may be underutilized. This intuition is confirmed by a study conducted on desktop PCs within Microsoft [46], which shows that disks are only half-full in average.

1.2 Description of a peer-to-peer storage system

The design of a peer-to-peer storage system and specifically of a peer-to-peer data backup system is a complex task, which involves lots of different aspects and poses a number of challenging problems. Before we discuss the different issues involved in such design, we propose a description of the system from two different points of view.

First, we define the properties that are expected from the system. We propose a sort of service contract that specifies what are the functionalities that a storage system must provide to the user. At this step, the system is seen as a black box and all the implementation and architectural aspects are ignored.

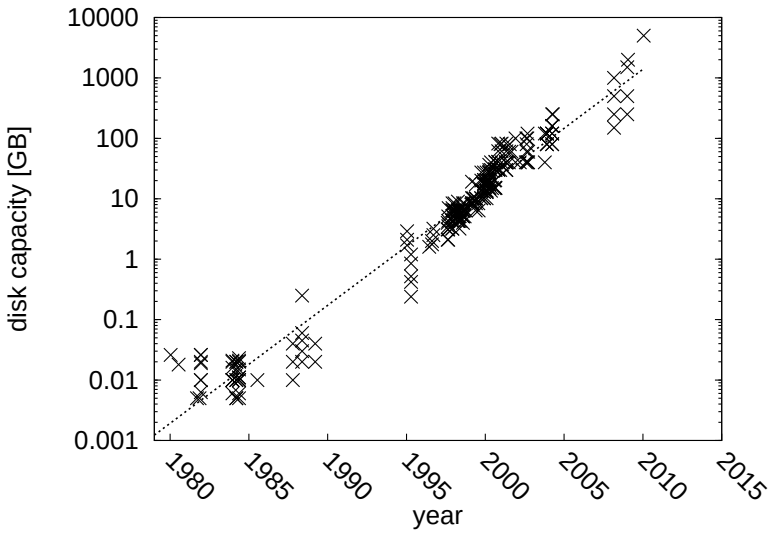


Figure 1.2: Capacity of hard disk over the time.

Second, we describe the constraints we have on the construction of the system. In practice we define the peer-to-peer nature of the system and we detail the expected behavior and characteristics of the participating peers.

1.2.1 Definition of a data storage service

We propose here a formal definition of a data storage service. We enumerate the features such a system must offer to the users and describe a typical interface between the service and the user.

In essence, a data storage system provides a reliable and secure storage of data. If one considers the system as a black box, the user will have two very simple primitives: *Store Data* and *Retrieve Data*, which correspond to the insertion of the data in the system and their retrieval. The system must guarantee a basic set of three properties with respect to these primitives:

- **Data Durability**. This is the ultimate purpose of a data backup system: the data that is inserted in the system is reliably stored and *never* lost. Eventually the user is able to retrieve his data back. Durability implies also that data are not corrupted or modified.

- **Data Availability**. The availability of data implies the ability of retrieving the data upon the user request. Note that this property differs from data durability. A given stored object could be durable but unavailable at certain periods. Availability and durability can also be evaluated in terms of the latency between a retrieval request and the actual data retrieval. Perfect data availability corresponds to a zero-latency, low data availability means high latency, while durability implies only finite latency.

- **Data Confidentiality**. The data inserted can be read only by a group of authorized users, which consists, in the most common case, only of the data owner.

1.2.2 Constraints of the peer-to-peer environment

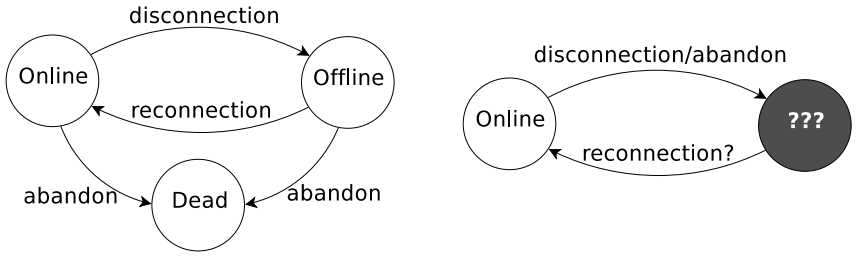
As mentioned in section 1.1.3, the essential property of peer-to-peer systems is that the resources are provided by the participating peers themselves. In the case of peer-to-peer data storage systems, the most important resource is the storage capacity. Every peer provides a certain amount of its local storage space, used by the system to provide a reliable storage service.

The peers participating in the system, however, are components that are not under the control of the system itself. The service properties must be guaranteed in spite of this lack of control, which can be characterized as follows:

- **Intermittent connectivity**. Peers are not connected to the system all the time for a number of different reasons: users may disconnect from the Internet, machines could crash or reboot, and there might be temporary network outages. This intermittent connectivity implies that data stored on peers incur periods of unavailability.
- **Permanent disconnections**. Beyond the temporary disconnections, peers may quit the system forever and the data stored on that peer are lost. The event of data loss may occur because of voluntary leave but also because of permanent hardware failures or deletion of data (accidental or voluntary).
- **Peer misbehavior**. Peers are not trusted parties. They can misbehave or behave in unexpected way. For example, Peers may claim to store data that they are actually not storing or can corrupt data they are storing. Moreover, peers may be non-collaborative, which means that they may try to exploit the services without providing any resource in return, which is usually referred to as **Free Riding**.
- **Limited bandwidth connection**. Peers are connected to the system through a connection with a limited bandwidth. This limitation might be due to the actual constraints on the access link or to a bandwidth utilization cap that the user may deliberately pose on the peer-to-peer application.

To clearly understand the peer behavior, we introduce in Fig. 1.3a a state machine that models the evolution of the life-cycle of every single peer. Every peer alternates periods in which it is online to periods in which it is temporarily offline. This alternation of disconnections and reconnections constitutes what we defined as intermittent connectivity. After a period, called **lifetime**, the peer can abandon the system and become dead. We called this event permanent disconnection, and, as already mentioned, a permanent disconnection translates to data loss and it is conceptually equivalent to when a peer deletes the data it is storing.

It is important to notice that the real peer behavior is not observable, since the system knows only whether or not a peer is connected and it is not able to tell apart disconnections from abandons. In Fig. 1.3b we depict the state machine as it is observable from the system. As we will discuss in the following sections, the lack of knowledge about the actual status of the peers and thus of the data they are storing, makes the data maintenance challenging.



(a) Real peer behavior **(b)** Observed peer behavior

Figure 1.3: State machine modeling the behavior of a peer.

1.2.3 File-systems vs. data backup systems

While the main focus of this thesis is on peer-to-peer data backup systems, these systems share a lot of features and design issues with peer-to-peer file systems. Indeed a considerable part of the literature addresses issues that refer to the generic field of peer-to-peer storage systems. Yet, data backup systems and file systems differ a lot in terms of assumptions and properties required and these differences can have a strong influence on the design choices. In this section we point out such differences and discuss how they are reflected in the design.

Data availability and durability

Previously we introduced the concept of data availability and data durability. It is obvious that availability implies durability, while the opposite is not always true: a durable object can be unavailable at certain periods. A first difference between file systems and file backup systems consists of the different importance given to data availability and data durability. It is obvious that availability is essential in file systems, since file reads and writes are operations that are not delay-tolerant, which implies that a given file must be available practically at any point in time. In file backup systems, there is no such constraint: the data owner accesses the storage repository after a disaster to recover what he has lost and normally can accept some reconstruction delay. The essential property of a file backup system is that data are *alive* somewhere, i.e. they will be eventually available.

The different importance of data availability and data durability has also another interesting implication. A file system user accesses stored files very often, which means that this operation should be relatively cheap. In file backup systems, data retrieval is hopefully very seldom needed, and, since this operation is very rare, it *can have* some additional costs.

Data updates

In file systems, files are not only read, but also modified. A big challenge in the design of peer-to-peer storage systems is to manage data updates and guarantee consistency. In data backup systems, data could be considered immutable (see for instance [81]) : data are backed-up on regular basis in big and static snapshots. Consecutive snapshots of the same

data can be either stored entirely or stored incrementally [33], which makes it possible to avoid any update of stored data.

Design guidelines

The differences between file systems and backup systems will lead the system designer to choices that could be radically different in the two cases. For this reason, the following guidelines can be formulated. A peer-to-peer backup system:

- Can relax the assumption on availability, which must be guaranteed only for what is strictly needed by the maintenance process in order to assure data durability.
- Should be organized such that the maintenance operations are very cheap. However, it is not essential that data retrieval and insertion be extremely fast and cheap.
- Can assume that data are immutable and avoid all the design complications due to data updates, concurrent accesses, and replica consistency.

1.3 Issues in peer-to-peer storage systems

In this section we discuss the issues that must be addressed in the design of peer-to-peer storage systems, paying particular attention to those that relate closely to peer-to-peer data backup systems. For each issue we provide a brief overview of the problems and an outline of the different solutions proposed in literature.

The challenge "number one" in the design of peer-to-peer data backup systems (and of peer-to-peer systems in general) is the construction of a reliable storage service out of many unreliable and uncontrollable components. The key solution to this problem consists of exploiting the diversity in behavior and characteristics of these components. The idea relies on the fact that even if all the peers will incur periods of temporary or permanent disconnections and some of them will delete or corrupt data, these events are not strongly correlated and at any point in time there will be enough connected and well-behaving peers to guarantee the correct functioning of the system.

To exploit this idea there are three main tools: data redundancy, data maintenance, and data placement, which are discussed respectively in the following three sections.

Data redundancy and its maintenance, however, are not the only important issues in the design of a peer-to-peer data backup system. In particular there are lots of architectural aspects that need to be addressed. One of them is for example the organization of peers and the organization of data and metadata. The organization of peers concerns how peers interact with each other and the level of decentralization of the system: is there a central entity that coordinates all the operations, is there a structure in the peer-to-peer overlay network, are the peers completely autonomous. The organization of data, instead, is related to how the data inserted in the system are formatted, distributed and referenced. The organization of peers and data are strongly related and, for this reason, both of them are discussed in section 1.3.4.

The aspects we discuss in section 1.3.5 are incentives and peer fairness. The feasibility of peer-to-peer systems requires that peers contribute their resources to the system. Without any incentive mechanism, peers would tend to be selfish, i.e. they would consume resources without providing any in return. One of the most widely used incentive mechanism consists of enforcing **peer fairness**. Peer fairness consists basically of excluding free-riders, applying a sort of "tit-for-tat" policy between the resources consumed by peers and those provided.

A lot of the solutions to the issues presented take advantage of mathematical models that capture the behavior of system. In section 1.3.6, we illustrate briefly some of these models, focusing especially on models of peer behavior and repair process, which represent also an essential building block in our contributions.

1.3.1 Data redundancy

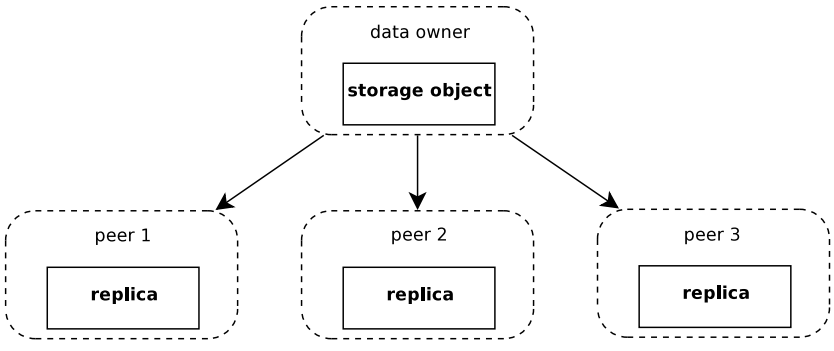
Data redundancy is the essential tool in order to exploit peer diversity. Data redundancy consists of storing multiple instances of the same data on different peers in order to mask data unavailability: even if part of the stored data is unavailable, the remaining part should be enough to reconstruct the original data.

There exist a lot of different techniques to add redundancy to data and they are usually called **redundancy schemes**. Each redundancy scheme determines (i) how to create the redundant data and (ii) how to rebuild redundant data when lost. These two operations generate costs that differ from one scheme to another. Here we introduce the most widely used redundancy schemes: replication and erasure coding.

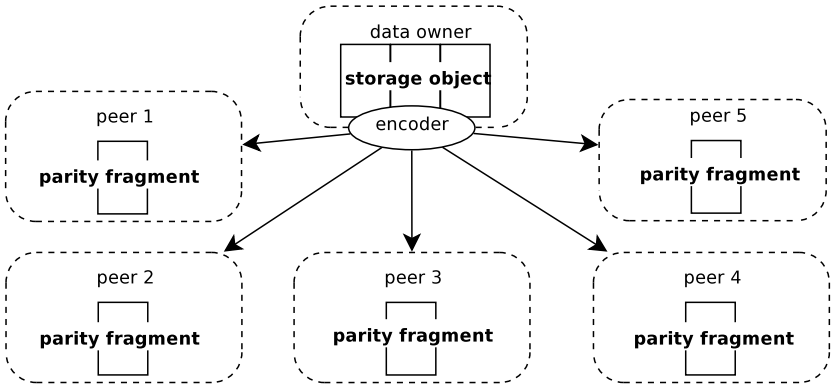
The simplest way to store data redundantly is **replication**: If we store N_{rep} replicas of the same file, even if $N_{rep} - 1$ of these replicas are stored on peers that are unavailable, we are still able to retrieve our file back. The data insertion for replication is depicted in Fig. 1.4a for the case $N_{rep} = 3$. In the case of replication, rebuilding a lost replica is straightforward, since it is enough to create an exact copy of any other alive replica, as depicted in Fig. 1.5a.

A smarter redundancy scheme is **erasure coding**. In general terms, a set of k data objects can be erasure coded in $k + h$ parity objects, where the size of any parity object is the same as the one of an original object. These parity objects are such that any k of them are sufficient to reconstruct the original k objects. The name "erasure coding" expresses the ability to sustain up to h erasures of parity objects without losing data. A way to apply erasure coding to a single storage object is to divide it in k (original) fragments and then code them in $k + h$ parity fragments, such that any k of them are sufficient to reconstruct the original object. The insertion of data for erasure coding is depicted in Fig. 1.4b in the case of $k = 3$ and $h = 2$. Note that erasure codes are able to consume less storage space as compared to replication, providing the same level of reliability. To understand this point, consider the replication scheme (with $N_{rep} = 3$) and the erasure coding scheme (with $k = 3$, and $h = 2$) of Fig. 1.4. In both cases, the system can accept up to two losses without losing the original data, however replication consumes storage space of 3 times the size of the original data, while erasure coding consumes only $5/3$ times the size of the original data.

In erasure coding, repairs are more complex than in replication. When a parity fragment is lost, its reconstruction requires the access to other k alive parity fragments. Note that to



(a) Replication with 3 replicas.



(b) Erasure coding with $k = 3$, $h = 2$

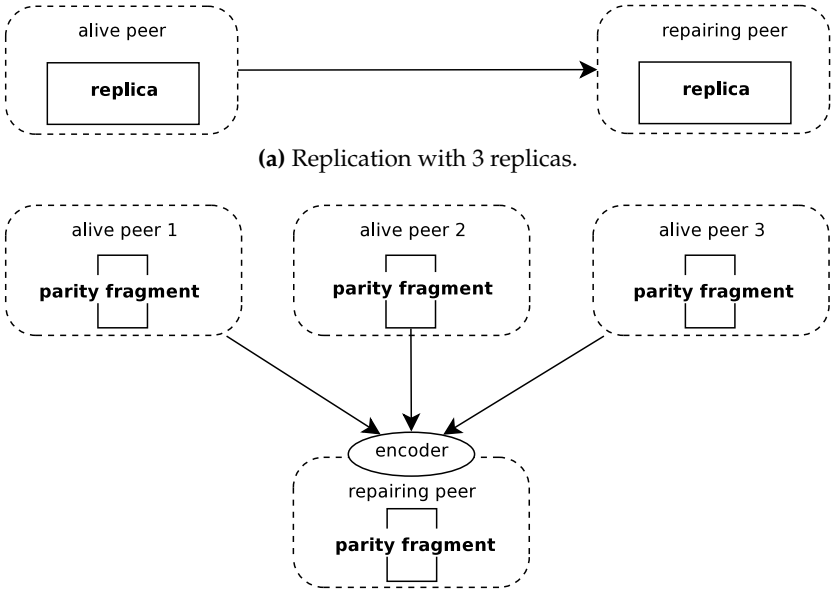
Figure 1.4: Redundancy schemes: data insertion.

exploit data redundancy it is essential that the replicas or the parity fragments are stored on different peers. Our assumption, implicitly introduced in Fig. 1.4, is that *each replica or parity fragment is stored on a distinct peer*. This choice maximizes the probability that data are available, since it exploits maximally the diversity of peers, but implies that all the data reads correspond to *data transfers*. When a repair of an erasure coded parity fragment is performed, the system needs to download k other parity fragments, which translates into the data transfer of an amount of data equal to the size of the *entire storage object*. This repair procedure for erasure coding is depicted in Fig. 1.5b in the case of $k = 3$ and $h = 2$.

Since redundancy schemes and the cost trade-offs they entail are one of the main focus of this thesis, we refer the reader to the section 2.4 for a detailed presentation of the state of the art and a description of the open issues.

1.3.2 Data placement

Thanks to data redundancy, multiple instances of the same data can be placed in different locations, i.e. on different peers. This is the necessary condition to exploit the diversity of these peers. However, it is important that such diversity be actually present in the peers employed. If, for instance, peers are perfectly synchronized in their disconnection times, no matter how much redundancy we add, there will be periods in which data will be unavail-



(a) Replication with 3 replicas.

(b) Erasure coding with $k = 3$, $h = 2$

Figure 1.5: Redundancy schemes: data repair.

able. For this reason, the choice of the peers used to store the data, usually referred to as data placement, plays an important role for the final data availability and durability.

In a scenario where peers behave randomly and independently, a random data placement is a reasonable solution. A lot of systems assume such independence and adopt a random placement [27, 29, 30, 34, 103].

In reality, independence in peer behavior may not always be realistic. Some peers exhibit correlated behavior, which means that it is possible to identify classes of peers that behave in a similar way. An example of such correlation is given by diurnal online patterns for peers that are geographically close [28, 95]: users tend to stay connected during given periods of the day (e.g. working hours or evening), while they are disconnected during the night. Correlation may also happen in the failure patterns: Peers that are belonging to the same sub-network may be affected by the same network outage, peers that run the same operating system or the same software may be subject to the same bugs or vulnerabilities [58, 75]. Finally, it is possible to identify peers that show a very high connectivity in terms of up-time or communication bandwidth, while others that are very poorly connected. These correlations suggest to investigate non-random placement policies, which are aware of patterns in the peer behavior and characteristics.

Douceur et al. [47, 48, 49], in the Farsite project, address the optimization of data availability. They first analyze different heuristics for the initial placement of data and then propose a hill-climbing strategy, which exchanges constantly the position of replicas to increase data availability. Tian et al. [96] address both data availability and efficiency of resource utilization, propose a stochastic model to capture different behaviors of peers and propose a placement strategy able to exploit such stochastic model. Weatherspoon et al. [104] observe the peer behavior and group peers in clusters that show correlated behavior. These clusters are then used to pick a set of uncorrelated peers for data placement. Similarly the "Phoenix recovery

system" [62] identifies clusters of peers using information about their software configuration or network position and then chooses peers that are most likely not correlated.

1.3.3 Data maintenance

The event of data loss requires that the stored data be checked and refreshed, or in general terms maintained. **Data maintenance** consists of two different parts: the detection of unavailability, which we call **data monitoring**, and the actual repair of data, which we call **repair policy** and determines when, given the information collected by the data monitoring, a repair must be performed.

Data monitoring

Data monitoring implies a procedure that constantly monitors the status of the stored data. This activity can be in turn divided in two parts:

- **Peer monitoring**, used to detect if peers are online or not.
- **Data integrity check**, used to detect if data have been deleted or corrupted.

The basic mechanism to detect if a peer is online consists of checking if that peer answers to an external solicitation, like a ping message. An important remark is that the peer monitoring is only able to assess the fact that a peer is non-online, but it cannot say anything about the kind of disconnection (temporary or permanent) that has occurred. In other words, the peer monitoring has access only to the observable state machine of peers, as already depicted in Fig. 1.3b

The challenge in a peer-to-peer system that is completely distributed is how to organize these checks in a decentralized way. In structured overlay networks (see discussion in section 1.3.4), the structure of the network provides an easy way to perform such checks: in DHTs for example, every peer has a set of neighbors, exchanges messages with them and it is able to detect whether or not they are online. In unstructured networks, one may use gossip-based membership protocols [54], which rely on flooding or partial-flooding of connectivity information. In peer-to-peer storage systems this connectivity information must be used to understand which data are affected and take repair actions if needed. In DHT-based storage systems, for every storage object, a responsible peer is identified, usually called root, and this peer is in charge to listen to peer disconnections and understand if they affect the data it is responsible for. In the first systems [38, 50], the root stores the data it is responsible for on neighbors and the detection of a neighbor failure corresponds to the detection of data loss. In BitVault [110] the root relies on a DHT-based membership protocol and listens to all the disconnection and reconnection events filtering the ones it is concerned by. In Glacier [58], replicas from the same set of storage objects are stored approximately on the same set of peers. This choice allows peers belonging to the same *set* to exchange a list of stored objects and detect if some replicas are missing.

Orthogonal to the detection of peer failure is the detection of data integrity. Accidental data corruptions may be detected by means of checksums, while the detection of malicious data

corruption or deletion is trickier. One possibility is to create self-verifying data [89, 106], where every block of data is associated with a signature that is function of the block of data itself. If the function adopted is for example a cryptographic collision-resistant hash function any modification of the data won't be reflected in the signature and the modification is easily detected. This approach, however, requires the verifier to download the complete data block and recompute the signature to perform the comparison, which may be costly in terms of communication and computation. Another approach is to use a challenge-response scheme, where the verifier proposes a challenge to the data holder that can be solved only if the data holder actually stores the data block. These schemes usually adopt a probabilistic approach and use cryptographic functions [24, 67, 76].

Repair policy

The repair policy defines *when* to perform repairs, i.e. when to create new replicas or new parity fragments. This decision is non-trivial, since it has to be taken without having a complete knowledge of the status of the data. While data integrity checks can easily detect data that are corrupted and thus need to be repaired, peer monitoring is only able to say if data is unavailable, but cannot say if these data will become available again or they are permanently lost.

A repair policy must guarantee that at least the data stored on peers that abandon the systems, i.e. the permanent disconnections, be repaired. For this reason one very trivial approach is to repair all the data that become unavailable, no matter if they will become available again or not. This **eager policy** however is very costly in terms of (i) storage, because it consumes more space than what is strictly needed and (ii) in terms of communication bandwidth, since any repair corresponds to the transfer of the whole storage object in both cases of replication and erasure coding.

Smarter repair policies strive to reduce these costs by avoiding useless repairs. One approach is timeout based, which means that the system considers as permanently lost only the data that remain unavailable for a period of time that exceeds the timeout. In this **lazy policy** the choice of the timeout is critical, since a small timeout may cause useless repairs, while a big one may detect permanent losses too late, jeopardizing data durability.

Another lazy policy is based on a threshold, which fixes the minimum amount of redundancy that must always be available. In this case, the repairs are triggered when this threshold is attained. This approach, however, makes the repair process to act in bursts, which means that a lot of repairs are performed in a small window of time, while in other periods, when the system stays above the threshold no repairs are done. As we will explain in detail later, this bursty behavior causes an inefficient use of communication bandwidth, which ideally should be used as smoothly as possible.

Finally, there are the **proactive policies**, which try to choose a fixed rate at which repairs are performed to achieve a smooth utilization of communication bandwidth. However, the choice of the correct rate is very difficult and a wrong choice can cause either useless repairs or data loss.

Together with data redundancy schemes, the repair policies are the main focus of this thesis. For this reason, as in the case of data redundancy schemes, we will give a more detailed discussion of the related work in section 2.5.

1.3.4 Peer and data organization

The peer-to-peer nature of the system is given by the fact that the storage space is provided by the peers participating in the system. Under this broad definition there are many possible system architectures, which differ in the level of decentralization.

A very simple approach is the presence of a central entity, usually called tracker, which coordinates all the operations performed in the storage system. This design comes from the early file-sharing systems like Napster [15], where the tracker has a central index of the files, mapping them to their locations, and a complete knowledge of the peers' status. A design based on a central tracker is much easier than a completely distributed system. However, it poses problems of scalability, robustness and security, being the tracker a single point of failure. Lillibridge et al. [67] employed a central server to track the peers. This server is used only to find backup partners in order to exchange symmetrically storage. In the Google File System (GFS) [55] the architecture is based on a single master and many chunkservers. The master is a central point of control, which coordinates all the operations of data placement, data maintenance, etc., and serves all the requests. The files stored in GFS are divided in big chunks, which are then replicated and stored on the chunkservers. This choice implies that the master holds an index that maps files to chunks and chunks to chunkservers. This index however is not stored on the master, but it is stored in the chunkservers themselves; the master holds a volatile version of the index and can reconstruct it in case of failure.

In Farsite [20] machines are partitioned in subgroups and the system administrator assigns to each subgroup a partition of the file system, called namespace root. These groups of machines work in completely autonomous way and each of them runs a Byzantine-fault-protocol [32] such that any machine in the group has a consistent view of the file system partition. Such an organization, however, requires a relevant administration effort in the setup phase, which can be feasible in a corporate LAN, but it becomes impossible in an Internet-wide peer-to-peer scenario.

The solution to build a completely distributed system comes from Distributed Hash Tables (DHTs), which represented a revolution in the design of peer-to-peer storage systems. The purpose of DHTs is to provide a consistent mapping between keys and values (or storage objects) in a completely decentralized fashion. To give an example of how DHTs work, let us consider the design of Chord [93]. In Chord every peer and every object is identified by an m-bit key, which means that peers and objects could be logically positioned in a circle composed by the 2^m possible key values. The keys of peers partition the space into disjoint intervals and every peer is responsible to store the objects, whose keys fall in the interval that precedes the peer's key. The responsible peer replicates the objects also in some of its successor peers, to guarantee persistence of objects. Every peer maintains a set of links to other specific peers in the circle, these links constitute a structured overlay and provide a way to quickly find the peer responsible for a given key through simple routing protocols. When a peer joins or leaves, the peers run simple procedure to maintain the overlay in a consistent status. There exist many different DHTs [72, 83, 87, 111], which propose different

design choices and give different performance guarantees, but they all share the same basic functionalities.

An interesting issue in the use of DHTs for storage purposes is how to generate the unique key to identify a storage object. A very popular solution for the key generation is content hashing, which actually was already suggested in the design of Chord. Content hashing identifies a block of data with a key that is the result of a hash function applied to the data block itself. This choice provides a set of interesting properties:

- If the key space is big enough, this choice guarantees a very low probability of collision, which means that it is very unlikely that different blocks have the same key, as argued in the design of Venti [81].
- With high probability the keys are uniformly distributed in the key-space, which guarantees a good load-balancing among peers [93].
- It provides an easy way to perform integrity check "for free", i.e. a modification in data blocks is immediately detectable. As a side effect, the stored blocks can be considered as immutable data, since any modification of the data would change the identifier key as well.

If one wants to use DHTs directly as the basis of a peer-to-peer storage or backup system, another issue is how to store complex structures, such as file metadata and directory information in data blocks. A solution, which is employed in many systems, is to pack all the metadata information in the data blocks themselves. One instance of such solution is based on the construction of a hierarchy of hash functions and has been adopted for example in Venti and CFS [38]. To understand how this hierarchy works, let us consider the case of CFS. CFS relies on a block storage layer called DHash, which provides an interface to store and retrieve a generic fixed-size block B using its hash $H(B)$ as key. On top of DHash, CFS builds a hierarchical structure of linked blocks, which allows to group multiple data blocks keys together into inode-blocks F, which represent files, and in turn group multiple inode-block keys together into directory-blocks D, which represent directories. Finally directory keys can be packed together to constitute the root of the storage assets of a single user. Such hierarchical structure is depicted in Fig. 1.6.

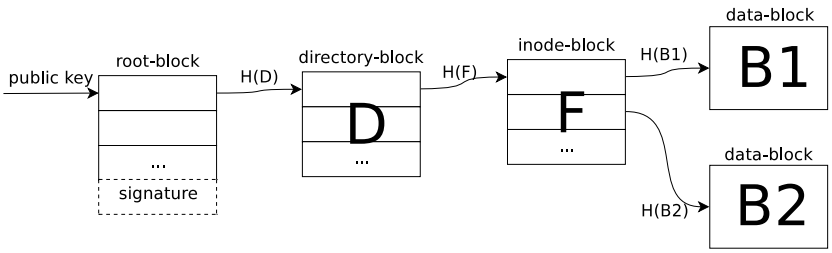


Figure 1.6: Data layout in CFS.

As previously remarked, modifications of a data block in a hierarchical structure like the one described, cause a chain of modifications in a whole subtree up to the root. If blocks are never deleted, different versions of the same data may coexist in the system and be referenced by many different roots. There are two implications of this property: (1) Storing multiple times

the same data does not require more space, which means that full backups are automatically implemented as incremental backups. (2) If different users store identical data they do not occupy twice the storage space. This property is very interesting, since, according to a study within Microsoft corporation [31], users that store the same content are not rare.

While the use of DHTs directly as storage layer, as in the case of DHash, is straightforward, many systems use them only to build an overlay among peers and for indexing purposes. These systems are then free to use their own placement strategies, redundancy schemes and repair policies. As an example of this approach, BitVault [110], as mentioned before, uses a DHT only to locate the responsible peer of a given storage object and to have a membership service: the data maintenance is delegated to the responsible peer and it is not performed by the replication mechanism of the DHT. DHTs can also be used to find nodes with given characteristics, like in Pastiche [36], which uses two DHTs, one to locate peers close in the network and the second to select nodes that have a good level of overlapping data.

Another important aspect to be taken into consideration is the data encryption in order to assure data confidentiality. Symmetric or asymmetric encryption techniques represent a very popular solution and the discussion of such techniques is out of the scope of this dissertation. It is interesting, however, to remark that classical data encryption prevents to store identical files only once: the same data inserted by two different users will be encrypted with different keys and need to be stored separately. A possible solution is to use convergent encryption [45], which uses the content hash as the encryption key, guaranteeing confidentiality and coalescing of identical content.

1.3.5 Incentives and Fairness

In peer-to-peer systems it is essential that peers contribute at least as many resources as they consume. Selfish peers may decide to exploit other peers' resources without providing their resources, which is usually referred to as free-riding. Free-riding can be catastrophic for the stability of the system and can compromise its feasibility.

In peer-to-peer storage systems the simplest case of free-riding takes place when peers insert data (including redundancy) whose size is larger than the storage space they offer to the system. To prevent this phenomenon a mechanism that enforces storage fairness is needed. One of the essential elements of any of such mechanism is a technique that allows to check if a peer is actually storing what he claims to store. This problem is closely related to data integrity, which we have already discussed in section 1.3.3. On top of this technique there are multiple ways to enforce fairness. One of these ways is symmetric storage exchanges: every peer selects a set of other peers and data are exchanged symmetrically. This "tit-for-tat" approach is simple and effective and has been employed in a number of systems [36, 67]. The limitation of symmetric exchanges is given by the constraints posed on the data placement, which as discussed in section 1.3.2 may improve the data availability and durability. Samsara [37] overcomes this issue defining the concept of **storage claim**. A storage claim held by a peer can be exchanged with storage space of another peer and becomes a sort of currency. This approach resembles a more general technique based on payments [57, 61], which, however, needs the existence of a central authority.

Other interesting approaches are given by reputation systems based on regular audits [77]. The efficacy of such systems can be demonstrated by game-theoretic models [78]. Game theoretic models have also been proposed by Toka and Michiardi [97, 98], where peers choose partners on the basis of their profiles, defined by online time, bandwidth, storage space etc. This selection creates a sort of stratification among peers and creates a strong incentive for peers to improve their profile to get a better service experience.

1.3.6 Modeling peer behavior and data repair

Evaluating redundancy schemes, data maintenance techniques, or incentives strategies in real implemented systems is very hard and costly. A cheaper way to understand the dynamics of a peer-to-peer storage system is to model its behavior through mathematical models, which should be able to capture the essential aspects of its functioning. In this section we discuss some of the models proposed in literature.

Utard and Vernois [100] model the behavior of a peer with a state machine which is conceptually similar to the one we presented in section 1.2.2. They also define temporary and permanent disconnections, which are driven by two parameters: the availability, which expresses the percentage of time a peer spends online in its lifetime and the volatility, which expresses the probability that upon a disconnection the peer permanently leaves the system. All the sojourn times in the different states are assumed to be exponentially distributed. On top of this simple model of the peer behavior, they build an aggregate model of the durability of a file stored in the system based on either replication or erasure codes, which store each replica or parity fragment in a distinct peer. The status of replicas or parity fragments is represented by a Markov chain in which transitions correspond to temporary disconnections of peers (i.e. data that become unavailable), permanent disconnections of peers (i.e. data that are lost), reconnections of peers (i.e. data that become available again), and repair operations (i.e. new redundant data created by the maintenance mechanism). Since the repair operations are modeled differently in replication and in erasure coding, this Markov chain is used to evaluate the impact of these two repair schemes on the durability of data.

Ramabhadran and Pasquale [82] propose a similar model based on a Markov chain, which however does not distinguish explicitly between temporary and permanent disconnections. It models repair operations for a system that uses replication. The study discusses the impact of the number of replicas and the aggressiveness of repairs on the data durability. The results suggest that, due to bandwidth and storage constraints, it is more convenient to maintain few replicas aggressively than a lot of replicas in a lazy way. It is interesting to mention that this work validates the exponential distribution of the connection and disconnection times analyzing the behavior of nodes in PlanetLab.

Dandoush et al. [39] propose a model similar to the one Ramabhadran and Pasquale based on a Markov chain. Their model, however, addresses both, replication and erasure codes, and aims to compare different repair strategies. In particular, they evaluate lazy and eager repair policies with both, centralized and distributed repair process. A centralized repair process is performed by a central entity, which downloads all the data needed for repairs and build all the data that has been lost at once. A distributed repair process, instead, can be performed by any peer, which carries out a single repair. Their numerical results for the models can be used to tune the system parameters to fulfill predefined requirements. It is

interesting to note that the authors propose also an improved model in which sojourn times are hyper-exponentially distributed, which offer a better representation of the real system. In a successive work [40], the authors show through simulation that repair times are also hyper-exponentially distributed.

1.4 Focus of the thesis

As we will discuss in chapter 2, the idea of distributed storage based on a peer-to-peer paradigm is not new. It has been around for about ten years and there are a number of relevant publications. According to their nature, the publications can be put in one of two categories:

- **Complete systems**, which describe fully-working systems and represent a general proof of concept. While they propose interesting techniques and solutions, the complexity of the whole design often prevents the authors to give a detailed discussion of the design choices they make.
- **Solutions to well-defined sub-problems**. These publications, instead, focus on specific aspects and provide an analysis and results that are reusable by others who want to build a complete system.

We decided to follow this second approach: the contributions of this thesis aim to provide *clearly understood and reusable building blocks* that solve specific issues efficiently.

Among the aspects involved in the design of peer-to-peer backup systems, we focus our attention on specific aspects of data storage reliability, such as the redundancy schemes and the repair policies. Our contributions pivot on a vital point: communication bandwidth can be a scarce resource. This fact, which will be discussed and justified in chapter 3, has been ignored in many works that investigated data redundancy and maintenance.

A lot of redundancy schemes proposed in literature focus on the efficient use of storage, while paying a high price in terms of communication. Classical erasure codes are one example. They are able to save a lot of storage, but they consume a lot of communication bandwidth for data repair, which can compromise the feasibility of the system. Our intuition is that it is essential to investigate redundancy schemes specifically conceived for distributed storage systems that take into account both, the storage space and communication bandwidth efficiency.

As in the case of redundancy schemes, maintenance policies need to consider the bandwidth consumption as well. In this case, however, the point is not how much communication bandwidth is needed for the maintenance, which is determined by the redundancy scheme itself, but it is *when* this communication bandwidth is needed. An important observation is that the communication bandwidth *cannot be saved for later use*, on the contrary unused communication bandwidth is just lost. As explained in section 1.3.3, threshold based repair schemes, which are the most commonly used, tend to make a bursty use of bandwidth, while proactive schemes are not able to match properly the behavior of peers. This fact makes us focus on the design of data maintenance algorithms that strive to smooth the bandwidth utilization and at the same time to adapt to the real behavior of peers.

1.5 Organization of the thesis

As discussed in the previous section, in the area of peer-to-peer storage systems, the related work can be divided in two categories. Chapter 2 proposes an overview of both categories, describing first some of the most important projects that build complete peer-to-peer storage systems and then discussing works that deal with subjects related to the focus of the thesis, namely data redundancy and maintenance.

In chapter 3 we present an analysis of the costs of a peer-to-peer backup system and pay particular attention to the interplay between the storage capacity of the system and the available resources at the peers. This chapter makes evident the importance of an efficient use of the communication bandwidth, and also provides all the tools to correctly frame and appreciate the contributions of the remaining chapters.

Chapters 4, 5 and 6 describe our main contributions and represent the core of this dissertation. While the first two chapters (4 and 5) deal with redundancy schemes that are both, storage and communication efficient, the last one (chapter 6) proposes a repair policy that aims to adapt to peer failure patterns and to smooth the communication bandwidth consumption.

Finally, in chapter 7 we draw some conclusions and propose directions for further improvements and extensions.

CHAPTER 2
Related Work

2.1 Introduction

In the last decade the topic of peer-to-peer storage has been actively investigated. In particular, we can find in the literature the description of a number of projects that aim to design complete peer-to-peer storage systems. While we presented in the previous chapter all the issues involved in such design, we provide in this chapter a discussion on how some of the most representative systems address these issues. The discussion proposed cannot be exhaustive and the objective is to provide the reader with the concepts needed to understand and evaluate the contributions of this dissertation.

A much more detailed discussion is dedicated to the topics that are closely related to our contributions, namely data redundancy and repair policy. In this part we discuss the state of the art and we describe the limitations of the current approaches. This discussion, together with chapter 3 will frame and motivate our contributions.

This chapter is organized as follows: in section 2.2 we give a historical perspective of the evolution of peer-to-peer storage systems. In section 2.3 we select the most relevant projects from literature and we propose a comparative discussion of them. In section 2.4 we conduct an analysis of the data redundancy strategies, discuss their evolution, and pose the motivations for two of our contributions in this domain. Finally, in section 2.5 we discuss the related work in the domain of repair policies for data maintenance, which is the subject of our third contribution.

2.2 The genesis

The seminal ideas of peer-to-peer storage systems come from two different research activities. On the one hand, companies made an effort to build network file systems that do not rely on central servers; on the other hand, the need to share files anonymously led the development of the first peer-to-peer systems.

2.2.1 Server-less network file systems

In organizations such as companies and universities it is very important to have a *global* file system facility, which provides a global namespace for files and a location-transparent access to them. This facility can be provided using a centralized server or a set of servers, which are accessed by clients through a network file system (e.g. [88]). Centralized implementations however are very expensive, servers often require special hardware to handle a very high rate of request or to provide a high level of reliability. Moreover, they represent a single point of failure.

The drawbacks posed by centralized implementations led to research in the field of distributed file systems. Some examples are represented by XFS [23] and Frangipani [94]. The assumptions made by both systems is that the set of machines collaborating are completely trusted and are under the control of the system administrator. These assumptions simplified a lot the design of these systems, which are almost a straightforward extension of the centralized implementations based on a cluster of servers. A step forward is represented by the Farsite project, started with a preliminary study [31], which investigated the feasibility of the deployment of a server-less file system on a set of existing PCs. The big novelty of Farsite was that the design did not assume any trust in the participating PCs, tolerating a significant number of failures and misbehavior of malicious nodes. For these reasons, Farsite represented the first example of a peer-to-peer file system and a milestone on this research area.

2.2.2 Peer-to-peer file-sharing and DHTs

In the late 90s the first storage systems distributed over the entire Internet were proposed [44]. The purpose of such systems was mainly the anonymous exchange of files. Even if the main objective in the design of these systems was to hide the identity of users, since most of the stored files were illegal and/or copyright protected, they underlined how the storage resources present at the users PCs could be used to build a global storage service and represented the first example of peer-to-peer storage systems.

While the first systems, like Napster [15], were based on a central server, called tracker, which knows the location of the different files, more advanced systems started to implement an unstructured routing mechanism, mainly based on flooding or on simple routing protocols, like Gnutella [7] or FreeNet [35].

The need for an efficient way to locate the content without a central tracker led the researchers to propose structured networks that provide the functionalities of a hash table but are implemented in a distributed way. These distributed data structures are called Distributed Hash Tables (DHTs) and have been proposed in many different flavors [72, 83, 87, 93, 111]

Except from Kademilia [72], no DHTs were deployed for peer-to-peer file sharing, but they enabled researchers to pursue the investigation in peer-to-peer file systems and storage in general. Indeed, the first works in this direction [38, 50], as we will explain in the following section, were conceived as a natural extension of DHTs.

2.3 An overview of selected projects

One can find in literature a significant number of projects that aim to design complete systems for distributed storage, distributed file systems, or backup. In this section we present a selection of the most significant of them.

At the end of the section we propose in Table 2.1 a comparison of the most relevant features of the systems described.

Farsite The Farsite project [6] has been conducted by Microsoft with the objective of building a serverless file-system exploiting the desktop PCs present within the company. Farsite aims to provide a network file system service (like NFS [88]), which means that data are not immutable, but they can be modified and deleted. The architecture of Farsite [20] proposes to partition the file system in namespaces and assigns different namespaces to different group of machines. Each group of machine uses replication and a Byzantine-fault-tolerant protocol to guarantee data integrity and data consistency. Data durability is provided using a timer-based repair policy in each group. To increase availability, data placement is continuously updated using a hill-climbing optimization algorithm. Data are encrypted by the clients before insertion in the system using convergent encryption.

PAST PAST [50] is an Internet-wide peer-to-peer storage system for immutable data. The purpose of the system is to build a per file reliable storage service, which provides data durability, data availability and data security. Even if it does not deal with automatic backup, or snapshot of big portion of users' data, PAST resembles a backup system, since it does not allow data modification or data deletion. Every peer can insert files into the system, which are identified by a unique FileID assigned by the system, and placed/located by the means of the Pastry DHT [87]. Like other DHT-based systems, the FileId determines the peer responsible for a given file and the redundancy is performed creating replicas on fixed number of neighbors of this peer in the overlay network. To guarantee load balancing, however, PAST allows replica diversion, which means that replicas can be placed also on peers that are not neighbors of the responsible peer. Peer fairness is guaranteed by the means of smartcards and a central authority, which enforces a quota-system.

CFS CFS [38] is a read/write Internet-wide peer-to-peer file system. The spirit of CFS is very similar to the one of PAST, the difference resides in the fact that CFS provides a sort of file system structure: files are not stored separately but they can be organized in a directory hierarchy. As discussed in the previous chapter, CFS relies on a block storage layer called DHash, on the top of which the file system is built. DHash uses the Chord DHT [93], which is in charge of replication and data maintenance. Fairness is provided imposing a per IP address quota system.

PStore PStore [26] is the first Internet-wide peer-to-peer storage system that addresses explicitly data backup. Data are organized in signed and encrypted chunks and then stored using the Chord DHT. The chunks support versioning, which allows to perform incremental

backup. Redundancy is performed by means of chunk replication. The replication however is not performed automatically by the DHT as in the case of CFS. Chunk replicas are identified differently and each replica is stored autonomously in the DHT, while the maintenance of such replicas is delegated to the data owner.

OceanStore OceanStore [16] is a large project based at University of California at Berkeley. The purpose of the project is to propose an architecture [64] for an Internet-wide archival storage utility based on the federation of untrusted servers. Within the framework given by this architecture, the project produced a large number of contributions in all the areas related to peer-to-peer storage. Some of the specific contributions will be discussed later in this chapter, while the interested reader can refer to the PhD dissertation by Weatherspoon [101] for a comprehensive view of the design aspects investigated in the project. To give some details of the OceanStore architecture we refer to an implemented prototype called Silverback [105]. In Silverback, every object is identified by a unique identifier, called GUID. The GUID is used as key to route in the peer overlay network, called Tapestry [111], which is indexing mechanism similar to a DHT. As in DHT-based systems, the GUID identifies a peer responsible for the object, which will be aware of the position of the erasure coded blocks created from the object. Updates are performed by versioning, which means that objects are never modified in place, but new versions (possibly incremental) are created. Data maintenance is performed thanks to two alternative mechanisms: (1) every peer sends periodic heartbeat messages to the root of every erasure coded block it stores, when the root misses some of the heartbeats it detects a disconnection; (2) there could be a third centralized party that checks the availability of objects. Placement and redundancy parameters are decided through introspection: the system observes itself and tries to infer the statistical properties of participating peers and then takes decisions based on these properties.

Pastiche Pastiche [36] is an Internet-wide peer-to-peer backup system. In Pastiche every peer finds a set of backup buddies. These buddies cooperate to backup their own data. The buddy selection is performed with two criteria: locality, which means that closer peers are preferred, and data overlap, which means that peers with more data in common are preferred. These two properties are obtained using two separate Pastry DHTs built using locality and overlap as distance metrics. Data are organized in chunks, which are encrypted using convergent encryption and then replicated on other buddies. Every peer is responsible to check the status of its chunks by polling on regular basis the buddies to detect their failure and ask them periodically some of the chunks they are storing to detect data corruption or deletion. Data modifications are allowed using a log-based incremental update scheme.

BitVault BitVault [110] is a follow-up of another project called RepStore [68] and aims to provide a storage service of immutable data within a local network, where peers are trusted. Peers are organized in a DHT called XRing [109], which provides a membership service. A membership service is able to notify every connected peer about other peers' connections and disconnections. Every object stored in the system has an identifier, which associate the object to a responsible peer. The responsible peer stores object replicas on other peers and stores locally a list of these peers. When a responsible peer receives the notification of the disconnection of a peer storing an object it is responsible for, it immediately triggers a repair.

When peers storing replicas receive the notification of the disconnection of the responsible peer, they elect the new responsible peer and communicate to him their identity. The responsible peer is in charge of choosing the peers to store the replicas, the choice is performed striving to provide load balance.

TotalRecall Total Recall [29] is an Internet-wide peer-to-peer storage system. The architecture of the system is very similar to the other systems based on a DHT. Every file is stored autonomously and identified by a key, which in turn identifies a responsible peer in the DHT. The novelty in Total Recall is that the repair policy and the redundancy techniques are differentiated accordingly to the kind of data to be stored and the characteristics of the peers. In general, file metadata, which point to data blocks, are replicated through the DHTs on the neighbors of the responsible peers, while data-blocks are erasure-coded and spread on other peers. Replicated nodes are repaired with an eager policy, while erasure-coded blocks are repaired through a lazy threshold based policy. Data updates are allowed and consistency is guaranteed by the responsible peer.

Glacier Glacier [58] is a peer-to-peer storage system intended to be used in combination with a *primary storage* within a large company network. While the primary storage uses a distributed replication mechanism with read/write facilities, Glacier acts as a backup of such a primary storage providing data durability against large-scale correlated failures. Data inserted in the primary storage are periodically packed in big storage objects and stored in Glacier as erasure coded blocks. Objects are signed with a hash-based manifest that is stored with each block to guarantee integrity. Each block is identified by a key, which is function of the object identifiers and a salt, this key is used by a placement function to place each block in the DHT. The placement function stores similar keys on the same set of peers, which means that sets of peers store the same sets of objects. Such a constrained data placement allows for an easy data monitoring: the peers belonging to the same set exchange a list of what they store and any discrepancy is fixed by repairs.

System	Type	Scope	DHT	Redundancy scheme	Repair policy	Data access	Data placement	Peer monitoring	Peer fairness
Farsite	file system	company LAN	-	replication	eager	read/write	availability	byzantine protocol	-
PAST	storage system	Internet	Pastry	replication	eager	read-only	DHT	DHT	smart-cards
CFS	file system	Internet	Chord	replication	eager	incremental updates	DHT	DHT	IP address quota
PStore	backup system	Internet	Chord	replication	-	incremental updates	random	-	-
OceanStore	storage system	Internet	Tapestry	hybrid	eager/ lazy	read/write	proximity	-	-
Pastiche	backup system	Internet	Pastry	replication	-	incremental updates	proximity/ overlapping	-	symmetric exchanges
BitVault	backup system	company LAN	XRing	replication	eager	read-only	load balance	membership service	-
TotalRecall	storage system	Internet	generic	hybrid	lazy	-	read/ write	DHT	-
Glacier	storage system	company LAN	generic	erasure coding	eager	read-only	load-balance	Bloom filter exchanges	-

Table 2.1: A comparison of existing peer-to-peer storage systems.

2.3.1 Commercial Solutions

It is interesting to mention that in the last years several start-ups have been created to propose peer-to-peer based file backup. However not much is known about the design and implementation of these systems.

Wuala [18] is the result of the research at ETH Zurich and offers an online file backup. Users start with 1 GB of storage and then can get as much as they want, either by trading idle disk space and bandwidth, or buying additional one.

Ubistorage [17] is a French company, which provides the users with a dedicated storage device connected to the Internet. This device is used to build the peer-to-peer infrastructure for data backup.

Allmydata [1] is an online backup company, which developed an open source software, called Tahoe-LAFS [2], intended for peer-to-peer data backup.

2.4 Data Redundancy Strategies

Data redundancy is a very important issue in reliable storage and in distributed storage in particular. It is not thus surprising that numerous papers focused on this subject and investigated optimal redundancy strategies. We first give a brief overview of the different redundancy schemes, and then discuss how these schemes are adopted in distributed storage systems.

2.4.1 Data redundancy schemes

Data redundancy is used in both, storage and communication systems. In the first case to provide data availability and durability, while in the second case for reliable delivery of data. This dual usage of redundancy is the reason why progress in this field comes from both areas, data storage and data communication.

Data redundancy techniques belong essentially to two categories: replication and erasure coding. We already introduced in section 1.3.3 the main characteristics of these two techniques.

Erasure codes based on XOR operations are used for RAID systems [79], while erasure codes based on linear operations on Galois Fields are the basis of Reed-Solomon codes [80, 84].

The main drawback of Reed-Solomon codes is the computational complexity of coding and decoding. To overcome this complexity, Luby et al. proposed Tornado Codes and LT Codes [69, 70], which are near-optimal codes with linear coding and decoding times, paying a small price in terms of reliability: the reconstruction is guaranteed when slightly more than k parities are used. These codes are based on Low Density Parity Check codes (LDPC) first proposed in 60s by Gallager [53]. An additional interesting property of these codes is that the parameter h is not fixed in the design, which means that the redundancy rate is not fixed

and can be accommodated as needed. The codes with this property are referred to as rateless codes or Fountain Codes. A survey of the possible applications of fountain codes is given by Mitzenmacher [73].

Another way to build rateless erasure codes comes from Network Coding in the domain of data delivery in a network. Network Coding is a generalization of the classical data forwarding mechanism. In traditional systems, intermediate nodes are only allowed to forward input data on the output links. Network Coding allows them to encode input data and send the encoded results on the output link. Ahlswede et al. [21] show that Network Coding can achieve the theoretically optimal data throughput. Other studies [63, 66] showed that linear encoding operations are enough to reach this optimum. Finally Ho et al. [60] proved that random linear encoding operations performed independently at the intermediate nodes can achieve the optimum throughput with a very high probability, which depends only on the field size adopted and can be made arbitrarily close to one increasing such field size. This last study opened up the possibility to build rateless erasure codes based on random linear codes and to use such codes for storage purposes [52] and in particular for distributed storage applications [19].

2.4.2 Redundancy in distributed storage systems

Replication is the simplest redundancy scheme and it is not surprising that most of the systems in literature are based on replication. This is particularly true with the first systems conceived as an extension of DHTs, where it is natural to adopt replication: the object are usually stored in the peer responsible of the object key, while its replicas are stored in the successor peers in the DHTs. For example PAST [50] adopts full-file replication, while CFS [38] divides files in blocks and then performs replication at the block level. Other examples of replication based systems can be found referring to Table 2.1.

The first system that proposed the use of erasure codes is FreeHaven [44], which argued that erasure codes are able to provide a higher data confidentiality: storing a coded blocks of an object is more *secure* than storing a full replica of the object, even if encryption is applied.

A study conducted by Baghwan et al. [27] compares replication and erasure codes and advocates the use of erasure codes for their higher storage efficiency: erasure codes are able to provide the same level of reliability as replication, consuming a much smaller storage space.

OceanStore [64] is based on a hybrid scheme, where meta-data are replicated, while normal data are erasure-coded. The reason for this is detailed in a study from the same project [103]. This study proposes an analytical comparison between replication and erasure codes and while it confirms the higher storage efficiency of coding with respect to replication, it argues that coded data can be accessed with a higher latency than replicated data. These reasons justify the adoption of erasure codes for stored data to save storage space and the use of replication for meta-data to guarantee a faster access. For the same reasons TotalRecall [29] proposes an hybrid scheme where data is both replicated and erasure-coded to provide low-latency, thanks to replicas, and durability at low storage cost thanks to coding.

In 2003, Blake and Rodrigues investigated the impact of the maintenance process on the economy of the system. Their study [30] showed how the communication bandwidth needed for

data maintenance may be quite large and might be unsustainable in common Internet scenarios when erasure codes are used. The reason for this is that when an erasure coded block must be repaired, the whole file is needed. This means that any repair operation requires the transfer of an amount of data equal to the whole file size, which becomes unfeasible if the failure rate is too large or the network bandwidth is too small. Rodrigues and Liskov [86] carried out a comparison between replication and erasure codes taking into consideration the maintenance communication bandwidth needed and concluded that advantages of erasure codes in terms of storage might not be worth their disadvantages in terms of communication bandwidth. To overcome this problem a number of works [30, 86, 107] proposed the use of a hybrid replication/coding scheme, where the erasure codes are used to provide a high level of reliability, while the replicas are used to compute repairs avoiding the transfer of the whole file.

Dimakis [42], however, argues that maintaining both replicas and coded-blocks increases the complexity of the system: replicas must be maintained as well and the criteria to maintain replicas are different from the ones to maintain coded-blocks. Moreover, the introduction of replicas decreases the storage efficiency of the redundancy scheme. Dimakis claims that the optimal solution is to find erasure codes that keep their storage efficiency providing communication efficiency as well and proposes a class of codes called Regenerating Codes.

2.5 Repair Policy for Data Maintenance

Storing data redundantly may not be enough to make data available and more importantly to guarantee data durability. During the lifetime of the system it is vital that data are monitored and maintained. The maintenance consists in replacing the data lost with new redundant data. The mechanisms to perform this replacement depend on the particular redundancy scheme adopted, as mentioned in the previous section, while the process that decides when this replacement must be done is a completely different issue, which can be referred to as the repair policy. In this section we give a short overview of the existing techniques, underlining what are the major trends and the open issues.

The first systems were very naïve from this point of view. In particular the main target of these systems was to provide availability at all costs. For this reason some of them [38, 50, 85] adopt a very simple technique: whenever a peer is detected to be offline, all the content it was storing needs to be recreated. The extreme simplicity of this technique is the reason for its popularity, however it is extremely inefficient as well. The inefficiency resides in the fact that the system considers all the disconnections as permanent disconnections: when a peer reconnects after a short period its content is not reintegrated. This choice may be acceptable if the rate of transient disconnections is low compared to the rate of permanent ones. Bhagwan et al. [28] made an analysis of the connectivity traces of Overnet, a file sharing network based on Kademilia, and pointed out how transient disconnections are very common. The conclusion of their study is that availability can be provided efficiently by taking into account both, short-term and long-term evolution of the peer behavior, i.e. transient and permanent disconnections. The same authors in Total Recall [29] classified repair policies in eager and lazy ones. An eager policy reacts immediately to disconnections, while a lazy policy delays the repairs in order to reintegrate peers that disconnected temporarily, avoiding

useless maintenance. They propose a lazy repair technique, based on thresholds. The idea is as follows:

- Identify the amount of redundancy that is strictly needed to guarantee the availability required. This amount is determined through introspection, i.e. observing the statistical behavior of peers, and it defines a repair threshold that must be guaranteed at all costs.
- When the data are stored for the first time, insert in the system an amount of redundancy larger than the repair threshold.
- Avoid any repairs as long as the amount of redundancy available in the system is larger than the repair threshold.
- When the available redundancy *hits* the repair threshold, perform all the repairs to bring the level of redundancy back to its initial level.

One limitation of this approach is that the system is able to reintegrate reconnecting fragments only as long as the threshold has not been hit and the repairs have not been performed. Chun et al. [34] proposed a very similar technique, called Carbonite that solves this issue. We propose a discussion about the differences between these two techniques in chapter 6.

In backup systems, as discussed in chapter 1, durability is far more important than availability. Lefebvre and Feeley [65] showed how durability could be provided at much lower price than availability and advocated the use of a repair policy able to separate durability from availability. The technique proposed is based on timers: a fragment is considered unavailable, but alive, when it is not available for a period smaller than a given timer value. This timer value is again determined by introspection.

Datta and Aberer [41] studied the time evolution of the system based on Markov models. They compared eager and lazy policies and proposed a new lazy policy based on randomization: Every disconnected node is repaired with a given probability that depends on the state of the system. They show how this randomized technique provides the same steady state guarantees of other lazy techniques, improving the resilience from a time-evolution perspective.

Sit et al. [90] put the emphasis on the smoothness of the bandwidth utilization during the repair process. They show that repair policies that make a bursty use of the network bandwidth are not efficient. The solution proposed is called Tempo and spreads the repair process over time to smooth as much as possible the network bandwidth utilization. Tempo assigns to every peer a bandwidth budget and then asks the system to perform repairs at a fixed rate that consumes the entire budget.

CHAPTER 3
Challenges and Costs

3.1 Introduction

The main pitfall in the evaluation of a peer-to-peer system is the misconception that no particular care in managing resources is needed as long as the resources provided by peers are larger than the resources that those peers would require if the service was implemented as a centralized service. This naïve evaluation neglects that the coordination of many different distributed entities requires a continuous exchange of information, which represents a cost in terms of communication. Moreover, the unreliability of peers requires a more robust and often more costly design, which would not be needed in a centralized service.

In the special case of data backup, one may believe that the system needs that peers provide only storage space, which must be larger than the global storage capacity we ask to the system. The reality is that assuring data durability in such a *hostile environment* entails a number of costs that include not only storage, but also communication and computation. These resources must be administered with care, since their availability is possibly limited and an inefficient use may reduce the total storage capacity the system can provide.

The objective of this chapter is to build an analytic model that captures the main costs of a peer-to-peer file backup system and their impact on the storage capacity of the system. In particular, the final outcome will be an upper-bound on the amount of backup capacity available for each participating peer as function of the resources that peers contribute on average to the system. Such an analysis is deeply important for two main reasons:

- Understanding the costs gives precious insights on what a system with given resource constraints can achieve in terms of backup storage capacity. The system designer, aware of these constraints, can infer what is reasonable to expect in terms of service and performance from a given set of peers.
- The analysis of the costs unveils the mechanisms that produce them, providing a clear path to follow if one wants to improve the system performance. This path justifies the directions of the work presented in this dissertation.

The objective of our analysis is not to cover all the details of such a complex system. The purpose is, instead, to give an idea of the inter-play between available resources, system performance, and system efficiency.

Section 3.2 describes the main characteristics of the modeled system. In particular it describes the life-cycle of a file stored in the system, and states the assumptions and the details of the model used to describe the behavior of peers. In section 3.3 we recall the characteristics of the main redundancy schemes and we analyze their efficiency, which will be then used in the following analysis. The core of the chapter is represented by section 3.4, where the upper-bound on the system storage capacity is computed. Finally in section 3.5 we discuss the results obtained, which allow to motivate and introduce the contributions presented in the next chapters.

3.2 Overview of the modeled system

Formalizing the definition of a peer-to-peer file backup system involves two different aspects: one refers to the definition of a file backup system, while the other is related to the peer-to-peer nature of the system.

A file backup system is a system able to store safely the data to be backed up. This means that, when the owner of the data needs them, they can be retrieved from the system. As detailed in section 1.2.3, this generic feature articulates in two specific properties: data availability and data durability. We already discussed how in a file backup system durability is essential, while a very high level of availability is not a strict requirement. In formal terms, a file backup system must guarantees something around 5 nines for durability, while an availability of 99% can be considered enough.

The peer-to-peer architecture of the system implies that the resources (storage, bandwidth, and computation) are provided by service customers themselves, which collaborate, hopefully without central coordination, to guarantee data durability.

In the next subsections we describe how this kind of system works in practice, we propose a model of the fundamental problems it has to cope with, and we elaborate the basic solutions adopted to solve these problems. All these elements will then be used to understand the cost-performance trade-offs.

3.2.1 File life-cycle

In a peer-to-peer file backup system, the participating peers collaborate to store *durably* a given amount of data. To understand how this is done, let us analyze the life-cycle of a single file inserted in the system. This life-cycle consists of three steps:

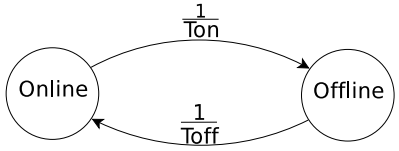
- **Insertion:** The file owner processes the file to add redundancy to it. The redundant data are uploaded to the system, i.e. they are uploaded and stored on peers participating to the backup system as already illustrated in Fig. 1.4.

- **Maintenance:** Once inserted the key-role of the system is to guarantee the durability of the files. This operation is non-trivial, since data durability is threatened by a lot of factors. In very rough terms the maintenance process can be described as the monitoring of data and the refurbishment of them when losses are detected. It is important to note that the maintenance operation is done by the system autonomously, without the presence of the file owner, which means that the original file is not available during this phase.
- **Reconstruction:** Eventually, the file owner needs to retrieve the file inserted in the system. This operation consists of downloading enough redundant data from the participating peers and reconstructing the original file from them.

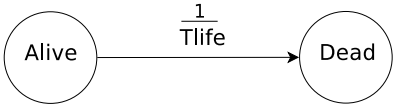
3.2.2 Modeling the behavior of peers

As mentioned in the description of a file life-cycle, there are a lot of factors that threaten the survival of the files stored in the system. The main problem is the intrinsic unreliability of peers, in particular the fact that peers are not continuously connected and more seriously the fact that peers can abandon the system for ever, experience software or hardware failures or delete data.

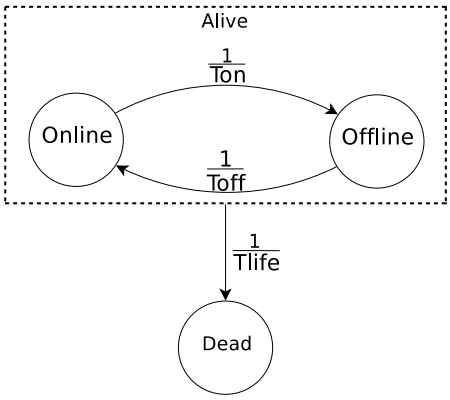
We already presented an approximate model of the peer behavior based on a simple state machine. Now we refine this model as depicted Fig. 3.1 and give a more formal description.



(a) Model of intermittent connectivity.



(b) Model of permanent disconnections.



(c) Combined model.

Figure 3.1: State machine modeling the behavior of a peer.

The model consists of two nested state machines. The first state machine is depicted in Fig. 3.1a and describes the intermittent connectivity of a peer, which alternates periods in which it is connected to periods in which it is temporarily disconnected. In particular the model defines two states:

- **Online**. A peer online is a peer that actively participates in the system. In this situation the peer is connected, i.e. can be contacted by any other online peer, and it is safely storing all the data that the system asked to store.
- **Offline**. An offline peer is temporarily disconnected, i.e. it is not reachable by other peers, but it is safely storing the data that the system asked to store. This condition covers the cases in which the peer is off, is experiencing a network outage, or its application has crashed.

The transition rates define how often a peer changes state and represent the inverse of the times spent by a peer in the different states. In particular we define:

- **Average session time**. The average time spent by a peer in the online state. It is denoted with T_{on}, while $1/T_{on}$ denotes the **disconnection rate**.
- **Average disconnection time**. The average time spent by a peer in the offline state. It is denoted with T_{off}, while $1/T_{off}$ denotes the **reconnection rate**.

A peer that is either in the online or in the offline state is defined as an **alive** peer. The union of the two states of the state machine of Fig. 3.1a corresponds, indeed, to the *alive* state of the model depicted in Fig. 3.1b, which represents the process of permanent disconnections. Formally, this model defines two states:

- **Alive**. An alive peer is either online or offline and it is safely storing all the data that the system asked to store.
- **Dead**. A dead peer is a peer that has logically abandoned the system and the data stored on that peer are permanently lost. This condition covers the cases in which the peer has deleted the data or it has quit the system for ever.

In this case there is a unique transition with a rate, called **abandon rate**, whose inverse defines the **lifetime**:

- **Lifetime**. The average time spent by a peer in online and offline state, before going to the dead state.

In Fig. 3.1c the two models presented are combined in a nested model.

It is useful to define an auxiliary parameter α, called the **up ratio**, which represents *the percentage of time a peer is online during its lifetime*:

$$\alpha = \frac{T_{on}}{T_{on} + T_{off}}$$

To be precise, in the nested model of Fig. 3.1c, the average time spent by a peer in the online state, i.e. the average session time, is not exactly T_{on}, since the average session time is function of both T_{on} and T_{life}. However if one assumes that $T_{life} >> T_{on}$, the influence of T_{life} can

be considered negligible (The same observation holds for T_{off} and the average disconnection time). In our case this assumption is valid since we consider T_{on} and T_{off} smaller than 1 day and T_{life} larger than 1 month.

3.2.3 The role of data redundancy

When redundancy is added to data, the obtained redundant data are such that even when part of them is not available the original content can be reconstructed. Clearly data redundancy inserted at the beginning can easily cope with the temporary disconnections, since if enough redundancy is added, even when part of the peers are in the offline state, the rest are enough to provide the stored files. However, we know that peers may also permanently disconnect and produce data loss, which means that, if nothing is done, the amount of redundancy present in the system decreases. For this reason we need data maintenance, which compensate the data losses by constructing new data, i.e. performing repairs.

To sum up, data redundancy inserted at the beginning copes with temporary disconnections and strives to provide data availability, while the maintenance of such redundancy copes with permanent disconnections and strives to provide data durability.

3.3 Redundancy Schemes

A redundancy scheme defines how the redundancy is added to the stored data. There exist different techniques to perform this operation and each technique presents a different trade-off in terms of benefits and costs. While in section 2.4.1 we described the main redundancy schemes used for reliable storage, in this section we propose an analysis of their reliability and efficiency.

3.3.1 Metrics

The analysis of a redundancy scheme consists in evaluating the reliability of the data storage and the costs such redundancy scheme produces.

The ability of a redundancy scheme to be resilient to such data loss is usually measured as the probability of a correct reconstruction of a stored object. Note that this measure is not absolute, but it is conditioned by the peer behavior: one of the most important factors is the probability of having concurrent failures. For this reason, the **reliability** of a redundancy scheme is measured as the number of concurrent losses that it can sustain without compromising the ability of reconstructing original data. More formally, one can express this property as the probability of data loss (*failure*) given that l concurrent losses occurred: $P(failure|l)$.

The description of the costs is more complex. To perform a complete analysis, we need to consider separately the two main activities involved in a storage system:

Storage The core activity of a storage system consists of the initial insertion of the data along with redundancy. The cost of the redundancy scheme, in this phase, is merely the absolute amount of storage space consumed. To abstract from the amount of data, it is measured as the ratio between the size of the stored data with redundancy $|data_{red}|$ and the size of the original data $|data|$. This cost can be referred to as **redundancy factor** and denoted as

$$\beta = \frac{|data_{red}|}{|data|}$$

Maintenance During the lifetime of a peer-to-peer storage system, permanent disconnections occur. Whenever this happens part of the redundant data is lost and the chances of losing the original data increase. If nothing is done to compensate these losses, sooner or later the durability will be not guaranteed anymore. The maintenance consists in refurbishing the redundant data when they are lost. This operation is performed reading the *available* data and producing new ones. The reading operation has a cost, which in a distributed storage system translates into network traffic, whose volume depends on the redundancy scheme adopted but also on lots of other factors, such as the peer behavior, the repair policy, the coordination algorithms etc. To evaluate only the contribution of the redundancy scheme, we measure the amount of data read with respect to the amount of new redundant data created. In other words, once the system has decided that a new redundant bit needs to be created, we measure how many (available) bits the scheme has to read. This cost can be referred to as **repair degree** and denoted as d.

Let us now apply the metrics proposed for the most representative examples of redundancy schemes: replication and erasure codes.

3.3.2 Replication

Replication is the most straightforward way to add redundancy. Its basic version consists in creating multiple copies of the object to store. The analysis of such a scheme is very simple. Let us assume that N_{rep} replicas of the original object are stored on different peers. The number of losses that the system can support is $N_{rep} - 1$. In a formal way the probability of losing the object conditioned by the probability of having a given number of concurrent losses is:

$$P(failure|l) = \begin{cases} 0 & l < N_{rep} \\ 1 & l = N_{rep} \end{cases}$$

The redundancy factor is

$$\beta = N_{rep}$$

while the repair degree is

$$d = 1$$

since the reconstruction of a new element corresponds to a simple copy of one replica.

3.3.3 Erasure Codes

A generic erasure (k,h)-code can be described as follows. Consider a file to store and split it in k (original) **fragments**, then process these fragments to produce $k + h$ **parity fragments** such that *any k of them are sufficient to reconstruct the original fragments*. The number of losses that this scheme can sustain is h. In a formal way the probability of losing the object conditioned by the probability of having a given number of concurrent losses is:

$$P(\textit{failure}|l) = \begin{cases} 0 & l \leq h \\ 1 & h < l \leq k + h \end{cases}$$

The redundancy factor is

$$\beta = \frac{k + h}{k}$$

This corresponds to significantly increased storage efficiency. We recall here the example proposed in chapter 1: consider a replication scheme with $N_{rep} = 3$ and an erasure coding scheme where $k = 3$ and $h = 2$. In both cases the system can accept up to two losses without losing the original data, however replication consumes storage space for 3 times the size of the original data, while erasure coding consumes only $5/3$ times the size of the original data. The price to pay for this increased storage efficiency resides in the repair cost. As we will show later, the existing coding schemes (with the characteristics described) require a repair degree of

$$d = k$$

Note that replication can be considered as a special case of coding where $k = 1$ and $h = N_{rep} - 1$.

3.4 An upper-bound on the storage capacity

To understand the implication of the costs on the practical functioning of the system, it is essential to compute the impact of such costs on the ultimate objective of a file backup system, i.e. the durable storage of the *maximum* amount of data.

In this section we propose a quantitative analysis of this impact. In particular, given a set of resources provided by the peers and given the behavior of such peers we compute an analytic upper-bound on the total storage capacity of the system.

Temporary and permanent disconnections impact differently the reliability of the system. Transient disconnections affect availability of data, i.e. the ability of reconstructing the storage object in a given moment, while permanent disconnections affect durability, i.e. the real survival of data. This is quite intuitive if one thinks that, without permanent disconnections, data might be unavailable but cannot be lost, which means that they can be eventually reconstructed. This might lead us to assume that in a backup system, where durability is the only thing that matters, we can avoid taking care of temporary disconnections. However, every repair, which happens in response to a permanent disconnection, *needs at least k parity*

fragments, which implies that providing durability requires availability, or more generally that poor availability may have a negative impact on durability.

In practice a system that wants to provide data durability must first provide data availability through proper data redundancy, which generates a storage cost, and then must guarantee durability by the means of a proper maintenance process, which generates a communication cost. Since these "activities" are inter-dependent, the costs they generate and thus the effects on the system storage capacity are inter-dependent as well. After having formalized in the next section the requirements and the available resources of the system, we first decompose these inter-dependent costs in their two components and then present an aggregate vision of them.

3.4.1 The system from a quantitative perspective

The peer-to-peer file backup system we analyze consists of a number N_p of peers that collaborate to store an aggregate amount of data whose original size (before applying any redundancy scheme) is $|data|$. The data to be stored consist of N_f files. Each file, whose size is denoted by $|file|$, is managed separately and when the repair policy detects for a given file the need of a repair, a new parity fragment is reconstructed using the existing ones.

In this analysis we do not consider the details related to the organization of peers, such as the overlay management, the monitoring infrastructure and so on. These issues are orthogonal to the problems studied here. While they can have an impact on the efficiency of the system, they mostly do not interact with the parameters we are considering.

We focus on the following resources that the peers provide and study how they impact the overall storage capacity of the peer-to-peer storage system.

Bandwidth Every peer is connected to the network through a data link with an average upload bandwidth of b. For the present analysis we consider that download capacity is unlimited, which simplifies a lot the task and can be justified by the fact that home users are mostly connected through asymmetric links, where download bandwidth is much higher than upload bandwidth. In other words since the upload link is most likely the communication bottleneck, assuming an infinite download bandwidth should not affect significantly the results. Moreover, the effect of a finite download bandwidth could only be a further reduction of the storage capacity offered by the system, which does not affect the validity of our analysis, which aims to find an *upper-bound* of such capacity.

Storage Peers participating to the system provide a certain storage space to the peer-to-peer storage system. The average storage space each peer offers is denoted as $|storage|$.

3.4.2 Amount of Redundancy

Following the same approach as in [30], we first consider a system where peers have an infinite lifetime, which means that stored data can never be lost. This system correspond to

the model of Fig. 3.1a. In such a system, durability is not an issue and availability can be provided using a proper redundancy scheme. The objective is to store for each file a number of parity fragments $k + h$ such that at any given moment k of them are on peers that are online.

In this computation we assume that the session time and the disconnection time are distributed according to an exponential distribution with a mean value equal respectively to T_{on} and T_{off}. The *exponential distribution* assumption is not novel and it is supported by the analysis performed in a number of other studies [39, 82, 100, 102].

Under this assumption the model corresponds to a continuous-time Markov chain and we can derive analytically (the derivation is proposed in section 3.6) the probability π_r of r peers out of $k + h$ to be online (and consequently $k + h - r$ peers to be offline), which is function of k,β, and α.

$$\pi_r = \begin{cases} \left[1 + \sum_{j=1}^{n} \prod_{i=0}^{j-1} \frac{\beta k - i}{i+1} \frac{\alpha}{1-\alpha}\right]^{-1} & r = 0 \\ \pi_{r-1} \frac{\beta k - r + 1}{r} \frac{\alpha}{1-\alpha} & 0 < r \leq n \end{cases}$$

A file is considered available, i.e. it can be reconstructed, when at least k peers are online. The probability of such condition is denoted with a and can be expressed as:

$$a = \sum_{r=k}^{n} \pi_r \tag{3.1}$$

which means that a is in turn function of k, β, and α.

If we set a target availability of $\hat{a}$, given the values of k and α, we can find using eq. 3.1 the minimum value of β that guarantees:

$$a_{k,\beta,\alpha} \geq \hat{a}$$

If we set, as discussed in section 3.2, a target availability of 99%, which corresponds to $a = 0.99$, we obtain the results shown in Fig. 3.2 as function of the parameter k and for different values of *up ratio* α.

An obvious observation is that more stable peers (larger α) require less redundancy. More importantly, the redundancy factor is highest for $k = 1$, which corresponds to replication, and the more we increase k the less redundancy is required.

3.4.3 Maintenance Process

The analysis carried out in the previous section assumes that no maintenance is needed. A more realistic analysis must take into account that peers have a finite lifetime and that, to provide durability, the pieces lost due to permanent disconnections must be periodically refurbished through repairs. A repair requires to transfer to a repairing peer the parity fragments needed to construct a new one. Given that the available bandwidth is limited, implies that the number of repairs in a given period of time is limited, which in turn will put a bound to the total amount of data we can store safely.

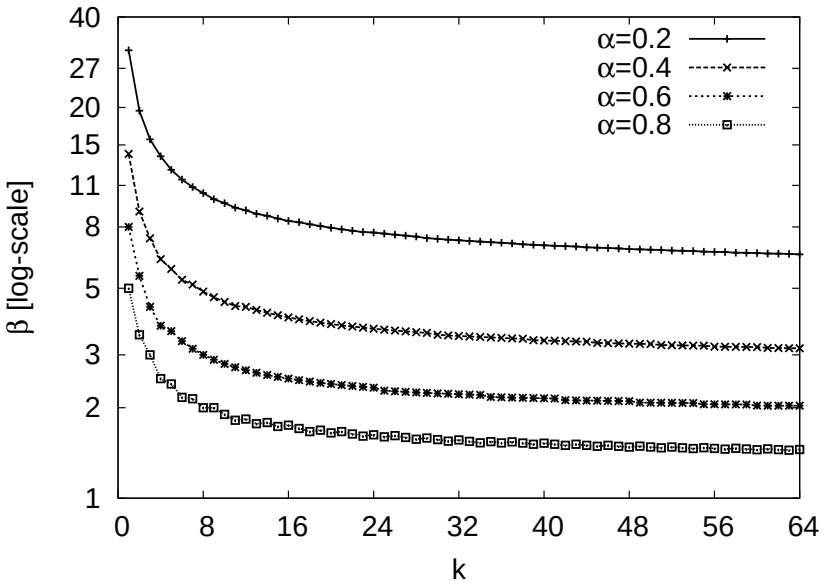


Figure 3.2: Redundancy factor β required for an availability of 99% as function of the *up ratio* α and the number of original fragments k the file is cut into.

Let us consider an ideal case in which an oracle tells apart permanent disconnections from temporary ones, which means that we perform only the repairs strictly needed to assure the durability of the data. Since the average lifetime of each of the N_p peers in the system is T_{life} and each peer stores on average $(k+h)N_f/N_p$ pieces, we can say that the number of pieces lost permanently constitutes a flow of events with a rate

$$\lambda = \frac{(k+h)N_f}{T_{life}}$$

From the model proposed in [51], the piece losses can be interpreted as jobs that enter a queuing system, which represents the repair process as depicted in Fig. 3.3. The "jobs" leaving the queue represent the new pieces created. To guarantee the stability of the system we must assure that the repair rate μ is larger than the loss rate λ.

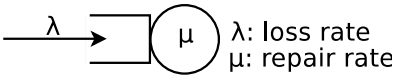


Figure 3.3: Queuing system representing the repair process.

To compute the repair rate, let us analyze a single repair operation. In this analysis we assume the use of traditional erasure codes, where a repair requires transferring k parity fragments from k connected peers to a new peer, called **newcomer**. In terms of upload data transfer, this implies the transfer of an amount of data corresponding to the size of a file. Since we are dealing with an average case analysis and assuming that the organization of repairs distributes equally the different repairs among all the connected peers, we can say that the effort required by a single repair is shared by all the connected peers[1], which are on

[1]We can make this assumption, since multiple repairs are performed at the same time and on average they load equally all the peers.

average αN_p. This means that the rate at which the system is able to complete a repair is given by

$$\mu = \frac{\alpha N_p b}{|file|}$$

where b represents the upload bandwidth of a peer. Imposing the stability condition $\lambda \leq \mu$ we have:

$$\frac{(k+h)N_f}{T_{life}} \leq \frac{\alpha N_p b}{|file|} \Rightarrow N_f |file| \leq \frac{\alpha N_p b T_{life}}{k+h}$$

The quantity $N_f|file|$ corresponds to the total amount of data stored in the system, on which we have obtained an upper-bound, which considering that $k+h = \beta k$, can be expressed as:

$$|data| \leq \frac{\alpha N_p b T_{life}}{\beta k} \tag{3.2}$$

3.4.4 Peer Data Share

In the previous section we derived an upper-bound on the total amount of data that can be stored safely in the system. More interesting is to quantify the amount of data that every single participating peer can store, which we refer to as the data share $|share|$.

The value of $|share|$ can be obtained by dividing the right hand side of eq. 3.2 by the number N_p of participating peers. We also know from section 3.4.2 that β depends on k and α. Given this we obtain:

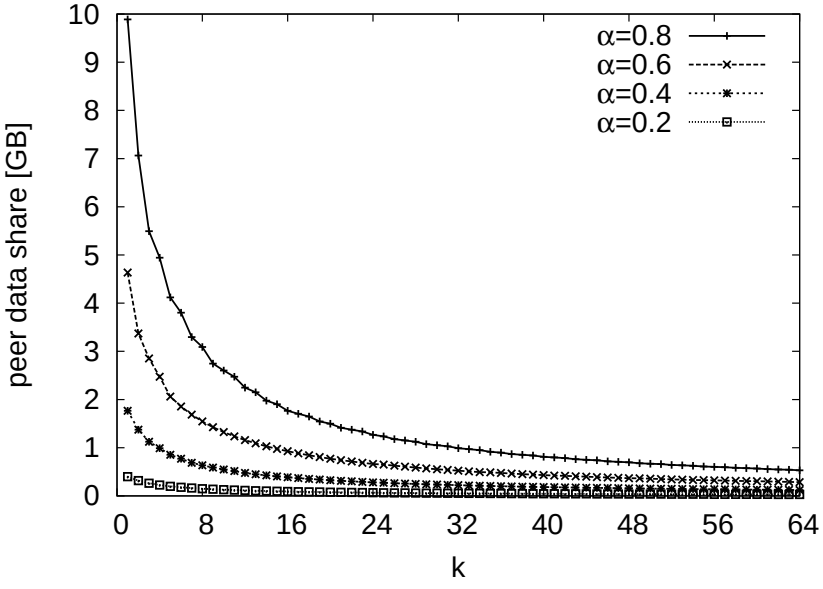
$$|share|_{k,\alpha,T_{life},b} = \frac{\alpha b T_{life}}{\beta k} \tag{3.3}$$

This result is very powerful as it expresses the relationship between the characteristics of the peers, the parameters of the erasure code employed, and the amount of data that every single peer can insert in the system. In other words, given all the other factors, no matter how peers are organized, how the system detects piece losses or performs repairs, $|share|$ as given in eq. 3.3 gives an upper-bound on the amount of data that can be stored safely.

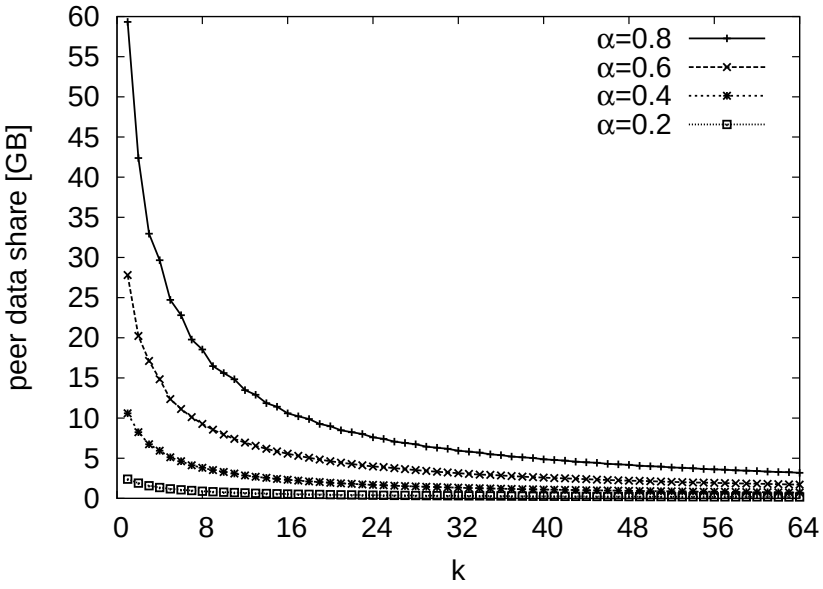
If we want to store more data, we need to adopt more efficient techniques to manage data redundancy.

To get an intuition for what values of $|share|$ are feasible, we plug in some values in eq. 3.3. If we set the upload bandwidth to 100Kbps and the target availability to 99% we obtain the results depicted in Fig. 3.4. One can easily see that poorly connected peers will prevent the system from storing more than few GBytes per peer. Moreover, these numbers reflect an ideal case in which an oracle tells the system when to do repairs and the system is able to do repair in the most efficient way possible. It is reasonable to expect that the amount of data that can be actually stored in a real system would be even lower.

Another conclusion one may draw from the results in Fig. 3.4 is that the more we increase k the less data can be stored safely. As a consequence, replication is the redundancy scheme



(a) 2 months lifetime.



(b) 1 year lifetime.

Figure 3.4: Peer Data Share for an upload bandwidth b=100Kbps, target availability 99% as function of the *up ratio* α and the number of original fragments k the file is cut into.

one should use since it requires the least communication bandwidth for piece repair and consequently allows to store more data. However, we need to remember what we saw in section 3.4.2, namely that replication requires much a higher redundancy factor than erasure codes. Since every peer may provide a limited amount of storage space, replication is possibly not the best choice. More formally, the total amount of data stored at a peer, including the redundant data, must be smaller than the total space capacity provided by a peer.

If we denote as $|storage|$ the average storage space provided by a single peer we get the additional constraint:

$$\beta|data| \leq N_p|storage| \tag{3.4}$$

If we fix the storage space provided by each peer at $|storage| = 10GB$ eq. 3.3 and eq. 3.4 will give the amount of data that can be stored safely by each peer as depicted in Fig. 3.5. This figure shows that the peer data share $|share|$ reaches a maximum at $k = \hat{k}$:

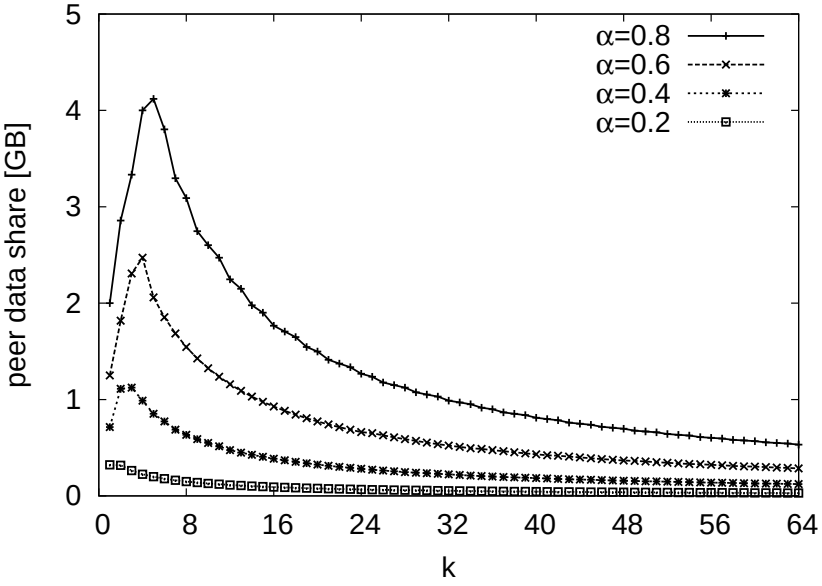
- For $k < \hat{k}$, the system is *storage limited* since the quantity of data that every peer stores amounts to $10GB$ and the amount of data $|share|$ that a peer can safely store is given by $10GB$ divided by the redundancy factor required for that particular value of k.
- For $k > \hat{k}$, the system is *communication bandwidth limited* and the curves are identical to the ones in Fig. 3.4.

These results illustrate nicely how the choice of k in the use of erasure codes allows to trade-off between storage and bandwidth consumption.

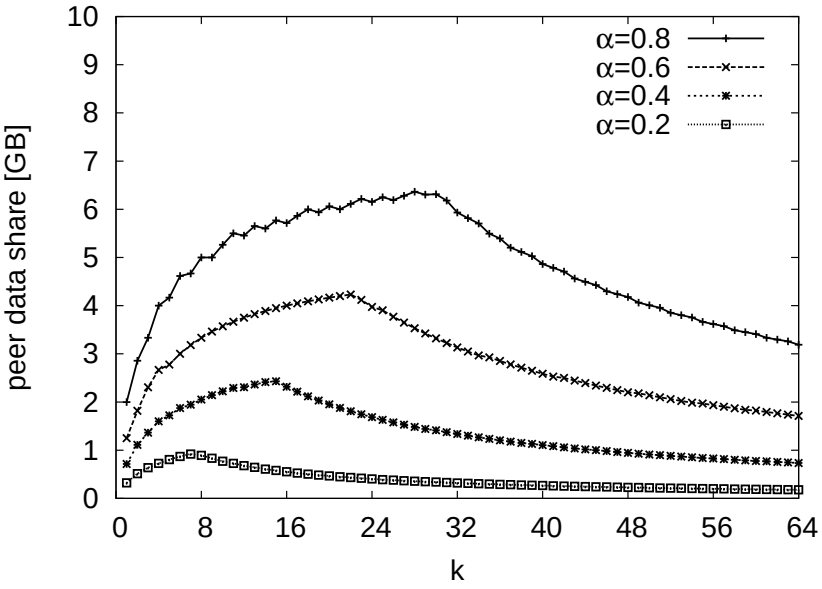
For reference and clarity we summarize in Table 3.1 a list of the symbols adopted and their meaning.

symbol	meaning
T_{on}	Average session time of peers.
T_{off}	Average disconnection time of peers.
T_{life}	Average lifetime of peers.
α	Peer up ratio.
β	Redundancy factor.
d	Repair degree.
N_{rep}	Number of replicas in a replication scheme.
k	Number of original fragments in an erasure code.
h	Number of additional fragments in an erasure code.
N_p	Number of peers participating in the system.
N_f	Number of files stored in the system.
$\|file\|$	Size of a file stored in the system.
$\|data\|$	Total amount of data stored in the system without redundancy.
$\|data_{red}\|$	Total amount of data stored in the system with redundancy.
$\|storage\|$	Average storage space offered by every peer.
b	Average upload bandwidth offered by every peer.
$\|share\|$	Average amount of data every peer can insert in the system

Table 3.1: Table of symbols used in the computation of the upper-bound the system storage capacity.



(a) 2 months lifetime.



(b) 1 year lifetime.

Figure 3.5: Peer Data Share for an upload bandwidth b=100Kbps, target availability 99%, and a storage contribution of 10GB per peer, as function of the *up ratio* α and the number of original fragments k the file is cut into.

3.5 Lesson learned

The analysis we have proposed deals with an *ideal* system and neglects a lot of details present in a *real* system. However, the results are able to illustrate how important it is to administer efficiently the resources available to maximize the storage capacity provided by the system. The need for efficiency motivates our contribution and translates into the two issues we address: efficient redundancy schemes and efficient repair schemes.

3.5.1 Efficient Redundancy Schemes

From the discussion at the end of section 3.4.4, it is clear that existing redundancy schemes, namely replication and erasure coding, defines a clear trade-off, which is either *communication bandwidth efficient* or *storage efficient*.

The first question we aim to answer is: *Is there a redundancy scheme able to combine the storage efficiency of erasure codes and the communication efficiency of replication?*.

Our contribution to answer this question is twofold:

Implementation and analysis of Regenerating Codes Regenerating Codes, originally proposed by Dimakis [42, 43], seem to solve this issue, since they decrease significantly the amount of communication required for maintenance, while providing almost the same storage efficiency. However the original paper only provides a theoretic framework that allows the construction of these codes, while it does not discuss a real implementation.

Our contribution in chapter 4 consists of a description of a practical implementation Fenerating Codes, based on random linear codes. On the basis of this implementation, we evaluate analytically and experimentally the costs of such codes not only in terms of storage and communication bandwidth, but also in terms of computation. We show how Regenerating Codes are actually able to provide higher bandwidth efficiency as compared to traditional erasure codes, with almost the same storage efficiency, but they may generate considerable computational complexity, which in some cases renders their use impractical.

Hierarchical Codes We propose in chapter 5 a new class of codes, called Hierarchical Codes. They are conceived with the same objective as Regenerating Codes, but they do not suffer of a higher computation complexity. Hierarchical Codes are based on linear codes, but they allow the construction of parity fragments that depend only on a subset of the original fragments. This choice, on the one hand, reduces the required repair degree, while on the other hand, increases the number of repairs needed to provide the same level of reliability of a traditional erasure code. We show through experiments that Hierarchical Codes establish an interesting trade-off in terms of repair degree and number of repairs, which can be exploited to reduce the global communication bandwidth needed by maintenance.

3.5.2 Efficient Repair Policies

In chapter 6 we address the efficiency of the system from a different perspective focusing on repair policies. The repair policy defines when the repairs must be performed and, as discussed in section 1.3.3, this task is non-trivial mainly because the system is not able to tell apart temporary disconnections from permanent ones.

In the model adopted in this chapter, there are two *ideal* assumptions about the repair policy: (i) an oracle tells apart temporary disconnections from permanent ones (ii) repairs are done such that the communication bandwidth utilization is constant. The first assumption guarantees that the number of repairs performed is the minimum possible, while the second one

guarantees that the bandwidth is used as smoothly as possible. In a real system, where such an oracle does not exist, the repair policy likely performs more repair than what is strictly needed. Moreover, existing repair policies, mainly threshold-based, tend to perform repairs in bursts and thus to make a bursty utilization of the repair bandwidth, which is extremely inefficient.

Our contribution consists of a repair policy, based on a statistical model that is able to infer the permanent disconnections and to perform the needed repairs at a rate that is as constant as possible, which maximizes the smoothness of the bandwidth utilization.

3.6 Derivation of the distribution of the number of available fragments with infinite lifetime

Consider a set of $n = k + h$ peers that behave independently accordingly to the model of Fig. 3.1a. If we assume that session times and disconnections times are exponentially distributed with mean values respectively of T_{on} and T_{off}, and we store one fragment on each peer, we can model the number of available fragments with the continuous-time Markov model depicted in Fig. 3.6. The generic state r in this model corresponds to the condition in which r fragments are available and $n - r$ fragments are unavailable.

Assuming that the current state is r, the system can move to the state $r - 1$, if one of the r online peers disconnects, which happens with a rate r/T_{on}. The system can also move to the state $r + 1$, if one of the $n - r$ offline peers reconnects, which happens with a rate $(n - r)/T_{off}$. Repeating this analysis for each state the transition rates of Fig. 3.6 are obtained.

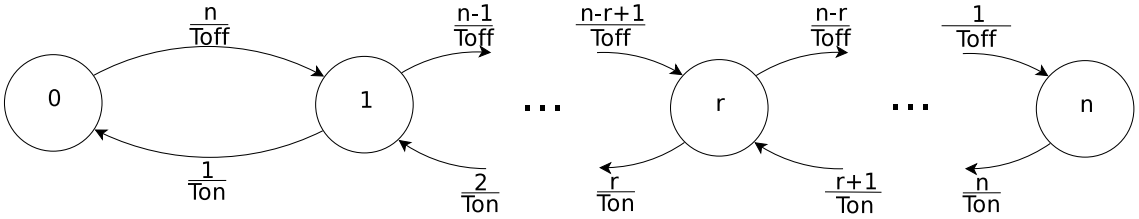


Figure 3.6: Continuous-time Markov model expressing the number of available fragments.

The model is a particular case of a birth-death process with closed population, as depicted in Fig. 3.7, where the rates are mapped as follows:

$$\begin{aligned} \lambda_r &= \tfrac{n-r}{T_{off}} \\ \mu_r &= \tfrac{r}{T_{on}} \end{aligned} \tag{3.5}$$

For such a system the expression of the state probability is well-known [99]:

$$\pi_r = \begin{cases} \left[1 + \sum_{j=1}^{n} \prod_{i=0}^{j-1} \frac{\lambda_i}{\mu_{i+1}}\right]^{-1} & r = 0 \\ \pi_{r-1} \frac{\lambda_{r-1}}{\mu_r} & 0 < r \leq n \end{cases} \tag{3.6}$$

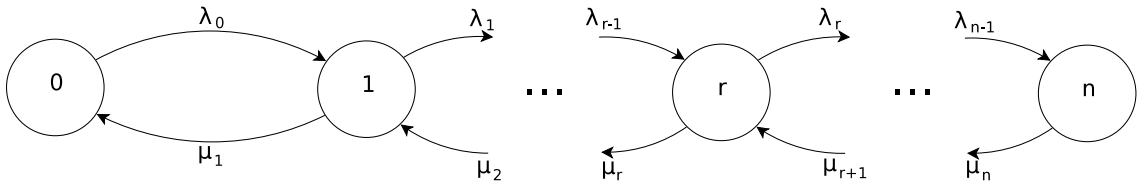


Figure 3.7: Generic birth-death process with closed population.

Using the mapping of eq. 3.5, we have:

$$\frac{\lambda_{r-1}}{\mu_r} = \frac{n-r+1}{r}\frac{T_{on}}{T_{off}}$$

Using the relation $\alpha = T_{on}/(T_{on} + T_{off})$, we have:

$$\frac{T_{on}}{T_{off}} = \frac{\alpha}{1-\alpha}$$

Plugging what we obtained in eq. 3.6 and using the relation $n = k + h = \beta k$, we can finally derive the state probabilities of the model of Fig. 3.6:

$$\pi_r = \begin{cases} \left[1 + \sum_{j=1}^{n} \prod_{i=0}^{j-1} \frac{\beta k - i}{i+1} \frac{\alpha}{1-\alpha}\right]^{-1} & r = 0 \\ \pi_{r-1} \frac{\beta k - r + 1}{r} \frac{\alpha}{1-\alpha} & 0 < r \leq n \end{cases}$$

Chapter 4

Regenerating Codes

4.1 Introduction

The cost analysis in chapter 3 shows how important it is to find redundancy schemes that provide both, storage and communication efficiency. **Regenerating Codes** are a class of erasure codes that provide almost the same storage efficiency as classical erasure codes, with a significant reduction of the communication bandwidth needed upon repairs.

The original description of Regenerating Codes does not propose a practical way to implement them and more importantly does not deal with the costs that such codes imply and consequently with their feasibility in the real world.

Our contribution aims to fill this gap. First, we give a detailed description of the theoretic properties of Regenerating Codes, then we describe our implementation based on random linear codes, and finally we perform an analytic and experimental evaluation of the different cost/performance trade-offs.

In the next three sections we formalize the notation used for generic erasure codes, describe how the redundancy is employed in the different phases of the system life-cycle, and we propose a detailed analysis of the costs for each of these phases. After this general introduction, we recall in section 4.5 the fundamental properties of Regenerating Codes and we describe in section 4.6 our implementation based on random linear codes. The last part of the chapter is dedicated to the evaluation of the costs generated by Regenerating Codes: in section 4.7 we propose an analytical evaluation of the complexity of the operations required by Regenerating Codes and, finally, in section 4.8 we describe the experimental results obtained testing our implementation of random linear regenerating codes.

4.2 Notation for Erasure Codes

Before going into the details of Regenerating Codes, it is essential to establish a unique notation for generic erasure codes. This notation must be generic enough to express all kinds

of erasure codes this dissertation deals with, namely traditional erasure codes, regenerating codes and hierarchical codes, which will be discussed in chapter 5.

Let us consider a file, whose size is denoted by $|file|$. Applying a (k,h) erasure code to the file means producing, out of this file, $k + h$ parity blocks such that any k of them are sufficient to reconstruct the original file. An important remark is that in this generic definition there is no constraint in the way these blocks are built and neither on their size. In particular the size of a parity block is $|block|$ and, since k of them must carry enough information to reconstruct the original file, the only constraint is:

$$|block| \geq \frac{|file|}{k}$$

The most common way of building erasure codes is linear codes. The mechanisms that linear codes rely on will be described in section 4.6.1, at this point, however, it is useful to introduce the notation used. In linear codes the original file is cut into n_{file} fragments, called **original fragments**. The i^{th} original fragment is denoted as o_i and its size is:

$$|fragment| = \frac{|file|}{n_{file}}$$

The original fragments are linearly combined to obtain new fragments, called **parity fragments**. The i^{th} parity fragment is denoted as p_i and its size is equal to the fragment size and denoted $|fragment|$ as well.

The parity fragments are then used to build the **parity blocks** of the erasure codes. In particular a parity block contains one or more parity fragments. The number of parity fragments contained in a block is denoted as n_{block}.

The parameter n_{file}, the way the parity fragments are built and the way these parity fragments are put in the parity blocks is determined by the particular erasure code adopted and will be amply discussed in this chapter and in the next one.

4.3 Refinement of the file life-cycle

In this section we recall what we presented in section 3.2.1 and we refine the operation involved in the life-cycle of a file when an erasure code is used. This will be useful in the description of regenerating codes and more importantly in the evaluation of the costs they produce.

1. **Insertion:** The insertion consists of processing the file, creating $(k + h)$ parity blocks and distributing them over distinct peers. The processing can be as trivial as building replicas of the file[1], or can be a complex coding operation. No matter which redundancy scheme is used, as explained in the previous section, the property of these parity blocks is that any k of them are sufficient to reconstruct the original file.

[1] This is the replication case where $k = 1$.

2. **Maintenance:** Maintenance consists of rebuilding the redundancy lost due to peer failures. Maintenance is performed by the means of repairs. A repair requires the cooperation of d peers that send data to a new peer[2], called newcomer, which in turn processes the received data to obtain a new parity block. We refer to d as the **repair degree**. If the repair is correctly executed, the new parity block has the same properties as all the others, i.e. with any $(k-1)$ other parity blocks it forms a set of redundant blocks sufficient to reconstruct the original file.

3. **Reconstruction:** If the owner of the file wants to retrieve it from the system, a reconstruction needs to be performed. The reconstruction consists of downloading data from k peers and processing them to obtain the original file.

4.4 Quantification of the costs

Thanks to the description of the operations involved in a file backup system, we can now propose a formal description of the costs produced by each operation separately. In particular there are three kinds of costs:

1. **Storage:** Redundancy implies that the stored file consumes more storage space than the original file. The storage requirement is easily computed by:

$$|storage| = (k+h) \cdot |block| > |file|$$

2. **Communication:** All three phases in the life cycle require data to be transferred among peers. At *insertion*, all the parity blocks must be transferred, which amounts to a volume of $|storage|$. At *maintenance*, for every repair, d peers upload each an amount of data equal to $|repair_{up}|$ to the newcomer for a total of $|repair_{down}|$, with the obvious relation:

$$|repair_{down}| = d \cdot |repair_{up}|$$

   At *reconstruction*, the file owner needs to download at least an amount of data equal to $|file|$ (See section 4.6.2 for details).

3. **Computation:** When coding is used, all the three phases described require processing of data[3]. At *insertion*, all the parity blocks need to be coded with a cost of *CPU(encoding)*. At *repair*, part of the processing is done on the d participating peers, denoted as $CPU(repair)_{up}$ and part is done on the newcomer, denoted as $CPU(repair)_{down}$. At *reconstruction*, the original file must be reconstructed from k parity blocks with a cost *CPU(reconstruction)*.

The particular redundancy scheme defines how the redundant data are generated and handled and what is the cost in terms of computation, communication, and storage. As an example let us consider traditional erasure codes (like Reed-Solomon codes [80]). For these

[2] A new peer is a peer that at the moment is not storing any parity blocks of the file.
[3] In case of replication there is no processing.

codes, the following two constraints hold with respect to the repair degree d and the parity block size:

$$\begin{aligned} d &= k \\ |block| &= |file|/k \end{aligned} \tag{4.1}$$

which means that every repair is performed collecting data from $d = k$ existing peers and that every peer stores an amount of data equal to $1/k$ of the file size. It can be shown that given these constraints, *the amount of data that needs to be transferred from every participating peer to the newcomer is equal to the size of a parity block, which means that in total an amount equivalent to the size of the whole file will be transmitted*. In terms of maintenance, the communication costs are:

$$\begin{aligned} |repair_{up}| &= |block| \\ |repair_{down}| &= |file| \end{aligned}$$

Note that this means that for every new bit that we create during a repair, k existing bits need to be transferred.

The computation costs are implementation dependent (see section 4.7 for details).

4.5 Description of Regenerating Codes

In this section we give a description of the main properties of Regenerating Codes as they have been described originally by Dimakis et al. [42, 43].

In essence, Regenerating Codes address the following question: what is the impact on the communication cost if we relax the constraints defined for traditional erasure codes given in eq. 4.1?

To answer this question, Dimakis et al. introduce a theoretic framework, called **information flow graph**, which maps the distributed storage problem onto a network communication problem. This framework is used to derive a lower-bound on the amount of information that need to be transferred upon a repair, as function of the repair degree d and the parity block size $|block|$.

We propose the results obtained in the following. Along with the generic formulation of the lower-bounds, to show their practical implications, we also propose the specific values obtained with a code with $k = 32$ and $h = 32$. This setting allows the system to sustain up to 32 losses. We consider this reasonable under the massive churn we may observe in an Internet scenario. However, results with other parameters show the same trends.

Given k and h, Regenerating Codes can take $k \cdot h$ different values for the pair of parameters $(d, |block|)$. In fact Regenerating Codes can be considered as a generalization of traditional erasure codes, which *trade-off increased storage cost for reduced communication cost*.

More formally, a generic Regenerating Code denoted by $RC(k, h, d, i)$, sets the following constraints on the repair degree d and the parity block size:

$$\begin{array}{rcll} d & \in & [k, k+h-1] & \\ |block| & = & p(d,i) \cdot |file| & i \in [0, k-1] \end{array} \tag{4.2}$$

Given a repair degree d, the parameter i, called the **block expansion index**, determines the parity block size through the function $p(d, i)$, which is defined[4] as:

$$p(d,i) = 2\frac{d-k+i+1}{2k(d-k+1)+i(2k-i-1)}$$

The values of $p(d, i)$ for $k = 32$ and $h = 32$ are plotted in Fig. 4.1.

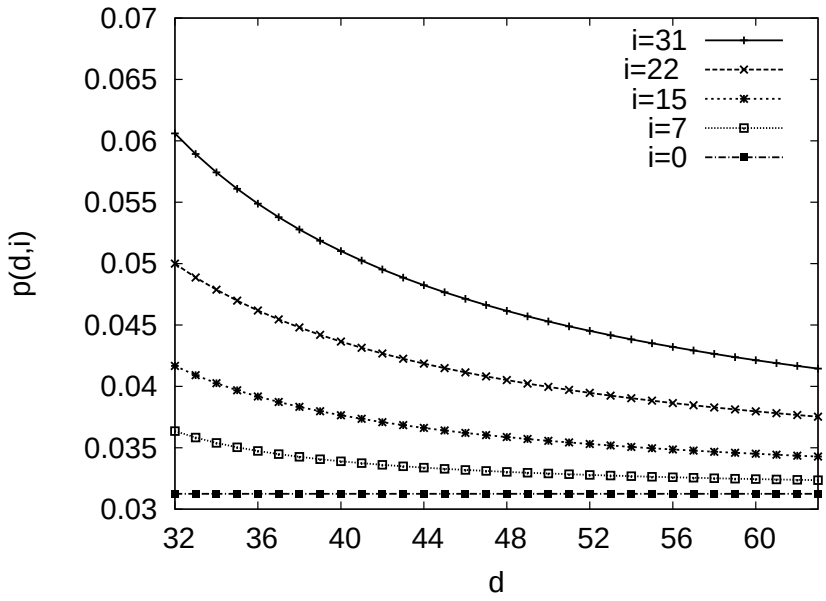


Figure 4.1: Function p(d,i), which determines the block size, for a Regenerating Code with k=32 and h=32.

It can be proved that $RC(k, h, d, i)$ requires that each of the d peers participating in a repair needs to transfer to the newcomer an amount of data **at least** equal to

$$|repair_{up}| = r(d,i) \cdot |file| \tag{4.3}$$

where $r(d, i)$ is defined as:

$$r(d,i) = \frac{2}{2k(d-k+1)+i(2k-i-1)}$$

consequently

$$|repair_{down}| = d \cdot r(d,i) \cdot |file| \tag{4.4}$$

The values of $r(d, i)$ for $k = 32$ and $h = 32$ are plotted in Fig. 4.2.

[4] We reformulate the expressions given in [42] in a different way to facilitate the successive computations.

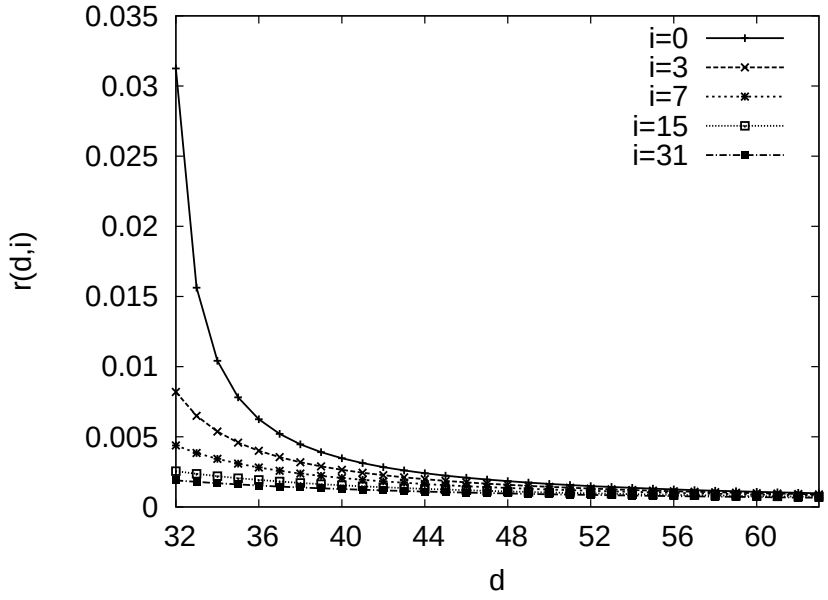


Figure 4.2: Function r(d,i), which determines the amount of data that needs to be transferred upon a repair, for a Regenerating Code with k=32 and h=32.

Fig. 4.3 depicts how the parity block size $|block|$ and the volume of repair traffic $|repair_{down}|$ evolve as a function of d and i for a code with $k = 32$ and $h = 32$. In particular, all the values are *relative* to the parity block size and the volume of repair traffic required by a traditional erasure code, which in the framework of Regenerating Codes corresponds to $RC(32, 32, 32, 0)$, i.e. with $d = 32$ and $i = 0$. As described in the previous section, these reference values are:

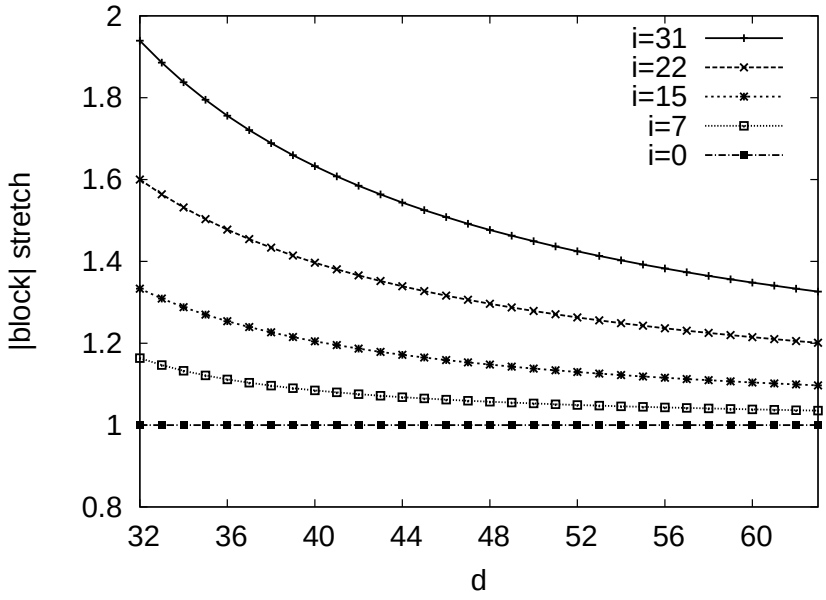
$$\begin{aligned} |block| &= |file|/32 \\ |repair_{down}| &= |file| \end{aligned}$$

Fig. 4.3 shows that moving to larger repair degree d and to larger parity block size (increasing the block expansion factor i) it is possible to obtain *an impressive reduction of the repair traffic*.

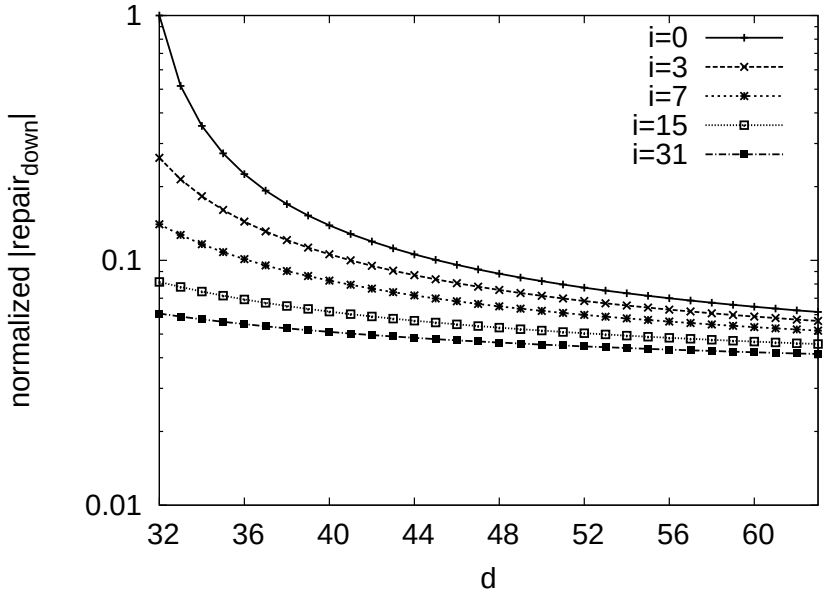
Dimakis et al. [42] identify two notable cases for the block expansion index, namely $i = 0$ and $i = k - 1$. For $i = 0$, the size of the parity blocks stays constant at the minimum possible size and the codes are called **Minimum Storage Regenerating codes** (MSR). For $i = k-1$, repair traffic is minimized and the codes are called **Minimum Bandwidth Regenerating codes** (MBR).

4.6 Random Linear Implementation

While Dimakis et al. [42, 108] present the theoretic framework that supports the construction of Regenerating, Wu et al. [108] show that these codes can be constructed through linear operations over Galois Fields. In particular, focusing on the case of $d = n - 1$, they prove the existence of both randomized and deterministic linear regenerating codes.



(a) $|block|$ stretch



(b) Normalized $|repair_{down}|$

Figure 4.3: Size of the parity blocks and repair communication cost (in log-scale) normalized by the reference values of a traditional erasure code, for $RC(32, 32, d, i)$.

Relying on the theoretic results of Wu et al. [108], we design a real implementation of Regenerating Codes based on random linear codes and we discuss its practical implications.

4.6.1 Traditional erasure codes based on random linear codes

Let us first explain how linear codes and random linear codes work for the case of traditional erasure codes.

The essence of linear codes is that all the operations are linear operations performed on fixed size fragments. These linear operations are performed over a Galois Field. The essential property of Galois Fields is that they define addition, subtraction, multiplication and division that are all **internal operations**, which means that the results of these operations applied to elements of the field are still in the field.

If the size $|GF|$ of the field, which is the number of elements belonging to the field, is of the form $|GF| = 2^q$, the elements of the field can be expressed as q-bit words. As we will explain in the following, a fragment can be interpreted as a vector of q-bit words and all the linear operations to be performed on fragments, are performed as linear operations on vectors in $GF(2^q)$

In section 4.2, we mentioned that for generic linear erasure codes, the original file is divided into n_{file} original fragments, used to produce parity fragments, which are in turn used to build up parity blocks. In the specific case of traditional erasure codes in discussion in this section, $n_{file} = k$ and a parity block contains exactly 1 parity fragment: $n_{block} = 1$. For this reason, *in traditional erasure codes, the words "redundant fragments" and "redundant blocks" are interchangeable*.

The easiest example of a linear erasure code is given by the parity check, which can be considered as a an erasure code with $h = 1$: consider k bits and build an additional bit applying the *XOR* operator to all the other bits. The *XOR* operation can be extended to all the bits in a fragment. If we denote as o_i the sequence of bits in the original fragments, and with p_i the sequence of bits in the created parity fragment, we can describe the parity check as follows:

$$\overrightarrow{p}_i = \begin{cases} \overrightarrow{o}_i & i <= k \\ \overrightarrow{o}_1 \otimes \overrightarrow{o}_2 \otimes \cdots \otimes \overrightarrow{o}_k & i = k+1 \end{cases}$$

It is clear that any k parity fragments are sufficient to build the missing one, applying again the *XOR* operator.

The *XOR* operation can be interpreted as a linear combination of all the original fragments in the domain of Galois Field with size 2, denoted as GF(2). In this field there exists only one possible linear combination among all fragments, which is the reason why only one additional fragment can be built. In erasure codes, a larger field size is used to be able to build $h > 1$ additional fragments.

Consider a Galois Field GF(2^q), where the elements of such a field can be expressed by q-bit words. This means that every original fragment and every parity fragment can be interpreted as a sequence of words in GF(2^q). For simplicity of notation, let us assume that every fragment is composed by a single q-bit word. In the general case of fragments composed by a sequence of words, all the operations are applied to all the words contained in the fragment. Let us denote as o_i and p_i the words belonging respectively to the i^{th} original fragment and the i^{th} parity fragment. A linear code can be built using the following linear operations in GF(2^q):

$$p_i = \begin{cases} o_i & i \leq k \\ \sum_{j=1}^{k} c_{i,j} o_j & k < i \leq k+h,\ c_{i,j} \in GF(2^q) \end{cases} \tag{4.5}$$

We can introduce the following vectors and matrices, all composed by elements in GF(2^q):

$O_{k,1}$	Vector of original fragments.
$P_{k+h,1}$	Vector of parity fragments.
$I_{k,k}$	Identity matrix.
$C_{h,k}$	Coefficient matrix.

Using these matrices we can give an alternative expression of the code:

$$P = \begin{bmatrix} I \\ C \end{bmatrix} O = C'O$$

If the matrix C is such that any sub-matrix S built using k rows from C' is invertible, then the original fragments can be always reconstructed by F=$S^{-1}P_S$, where P_S is the subvector of k elements of P, corresponding to the coefficients chosen in S. If this property is satisfied, the code obtained is a (k,h) linear erasure code.

Many choices are possible for the coefficient matrix, and consequently there exist multiple implementations of this class of codes. One of the most prominent are **Reed-Solomon** codes [80], which define the matrix C as a $h \times k$ Vandermonde matrix, i.e. $c_{i,j}$=j^{i-1}. For Reed-Solomon codes as well as for all the codes that fix a specific coefficient matrix, the repair of a lost block requires first the reconstruction of all the original fragments, and then their recombination accordingly to the coefficient row that corresponds to the lost parity fragment. This explains why the repair degree is d=k.

Another approach is to build the matrix C' choosing randomly the coefficients in the Galois Field[5]. This class of codes is called **Random Linear Codes**.

The theory of Random Linear Network Codes [19, 21, 66] says that the probability to successfully invert the matrix upon reconstruction depends only on the size of the Galois Field and that this probability can be made arbitrarily close to 1 by increasing the size of the Galois Field. For all practical purposes a field size equal to 2^{16} is considered sufficient.

The repair of a lost parity fragment can be done like in Reed-Solomon codes, i.e. first reconstructing the original fragments and then combining them again. In this case, the repair degree would be again d=k. However, with random linear codes we can do better: In fact the reconstruction of original fragments is *not necessary* and the result is indeed equivalent as to when the k parity fragments are combined directly using random coefficients.

Now that we know how random linear codes work from a mathematical point of view, we can describe their application following the life cycle of a file.

1. **Insertion:** In this phase, we have to create $k+h$ parity blocks, which correspond to $k+h$ parity fragments of size $|file|/k$. To do that, it is enough to cut the file in $n_{file} = k$ equal sized (original) fragments, and compute any of the $k + h$ parity fragments as a random linear combination of them. The random coefficients used for such combinations are stored along with the parity fragments.

2. **Maintenance:** As already explained, a repair in traditional erasure codes requires the transfer of the whole parity block (i.e. 1 parity fragment) from d participating peers to

[5]Note that replacing the identity matrix with random coefficients transforms the code from a systematic one to an unsystematic one.

the newcomer. The newcomer then builds the new parity block performing a random linear combination of the d received parity fragments. Again, the resulting coefficients are stored along with the new block.

3. **Reconstruction:** The owner of the file downloads k parity blocks (i.e. k parity fragments) from k other peers and uses these parity blocks to reconstruct the file. The procedure consists of inverting, if possible, the matrix composed by the coefficients of all the received parity fragments, and multiplying the inverted matrix by the parity fragments. The results of the multiplication are the original fragments, i.e. the original file.

4.6.2 Random Linear Regenerating Codes

For traditional erasure codes, random linear implementation is straightforward, because $n_{file} = k$ and $n_{block} = 1$. This means that the size of a parity block is equal to the size of a parity fragment and can be used as the basic unit of information in all the linear combinations and in the decoding.

For Regenerating Codes things are different because:

- They allow the amount of data stored $|block|$ to be different from the amount of data transmitted by a participant upon a repair $|repair_{up}|$.
- They do not require that the amount of data downloaded by a newcomer $|repair_{down}|$ be not a multiple of $|block|$.

In other words, the basic unit of information, which is the size of the fragments we cut the original file into, will not be the size of the block stored on a peer anymore. We can write the constraints as follows:

$$\begin{aligned} |file| &= n_{file} \cdot |fragment| \\ |block| &= n_{block} \cdot |fragment| \\ |repair_{up}| &= n_{repair} \cdot |fragment| \end{aligned} \tag{4.6}$$

where n_{file}, n_{block} and n_{repair} are integers. Using eq. 4.2 and eq. 4.3, we can compute:

$$\frac{n_{block}}{n_{repair}} = \frac{|block|}{|repair_{up}|} = \frac{p(d,i)}{r(d,i)} = d - k + i + 1$$

and:

$$\frac{n_{file}}{n_{repair}} = \frac{|file|}{|repair_{up}|} = \frac{1}{r(d,i)} = \frac{2k(d-k+1) + i(2k-i-1)}{2}$$

Both ratios are integers. This means that we can set $n_{repair} = 1$, which corresponds to setting $|fragment| = |repair_{up}|$, and consequently:

$$n_{file} = \frac{2k(d-k+1) + i(2k-i-1)}{2} \tag{4.7}$$

$$n_{block} = d - k + i + 1 \tag{4.8}$$

Given these parameters, we can describe the operations needed for Random Linear Regenerating Codes:

1. **Insertion:** We cut the file in n_{file} equal size original fragments, and compute any of the $k + h$ parity blocks as n_{block} random linear combinations of them. The random coefficients used for such combinations are stored along with the parity block. They form a (n_{block}, n_{file}) matrix[6].

2. **Maintenance:** A repair involves d existing peers, which send data to the newcomer. The data sent by any of the d peers correspond to the results of one random linear combination of the n_{block} parity fragments contained in the stored parity block, as depicted in figure Fig. 4.4a. The newcomer receives thus d parity fragments and the corresponding coefficients and obtains its new parity block as n_{block} random linear combinations of them, as depicted in Fig. 4.4b. Note that in the particular case of $d = n_{block}$ the newcomer does not need to perform linear combinations of the received parity fragments, since they constitute already the new parity block.

3. **Reconstruction:** The owner of the file downloads k parity blocks from k peers, which correspond to $n_{block} \cdot k$ parity fragments, along with the coefficients which form a $(n_{block} \cdot k, n_{file})$ matrix. It tries to find n_{file} independent rows in the coefficient matrix, then it inverts the resulting square submatrix and finally multiplies this matrix by the concerned parity fragments. An important observation is that if the file owner downloads k redundant blocks, it potentially downloads an amount of data quite bigger than the file size. In [42] it is stated that this can represent a significant drawback for Regenerating Codes. In our implementation, we eliminated this shortcoming: the owner downloads only the coefficients, then extracts a full-rank square submatrix, inverts it, and finally he downloads *only the* n_{file} *parity fragments corresponding to the invertible submatrix* that was extracted. In this way the owner downloads always an amount of data equal to the file size, without paying any extra-cost.

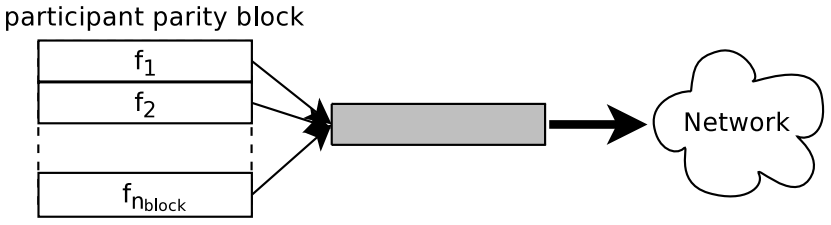
For reference we summarize in Table 4.1 all the symbols adopted and their meaning.

4.7 Analytical Evaluation

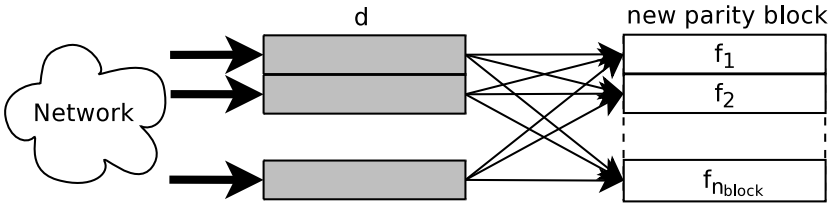
In this section we perform an analytical evaluation of the Random Linear Regenerating Codes. To do this, we give a formal description of the linear operations performed.

As explained in section 4.6, all the data handled can be interpreted as a sequence of values, called elements, in a given Galois Field. Usually the size of such field is chosen to be equal to 2^q, since this speeds up the computation. In this case every value is a sequence of q bits, a common choice is $q = 16$, which corresponds to an element size of 2 bytes. Every fragment

[6] In the present notation a (n, m) matrix is a matrix with n rows and m columns.



(a) Participant side



(b) Newcomer side

Figure 4.4: Repair scheme on the participant side and on the newcomer side. Every arrow indicates a participation to a random linear combination.

symbol	description
$\|file\|$	Size of a file inserted in the system.
$\|block\|$	Size of the parity block by a peer.
k	Number of parity blocks needed to reconstruct the file.
h	Number of additional redundant blocks stored in the system.
d	Repair degree. Number of peers participating to a repair.
i	Block expansion index, determining the size of the parity block $\|block\|$.
$\|storage\|$	Total amount of data stored for a file (with redundancy).
$\|repair_{up}\|$	Amount of data uploaded by each peer participating in a repair.
$\|repair_{down}\|$	Amount of data downloaded by a newcomer upon a repair.
$CPU(encoding)$	Computational cost to encode data upon insertion.
$CPU(repair)_{up}$	Computational cost sustained by each peer participating to a repair.
$CPU(repair)_{down}$	Computational cost sustained by the newcomer upon a repair.
$CPU(reconstruction)$	Computational cost sustained upon reconstruction by the file owner.
n_{file}	Number of fragments every file is cut into.
n_{block}	Number of parity fragments a parity block is composed by.
$\|fragment\|$	Size of a fragment.

Table 4.1: Table of symbols used in analysis and implementation of Regenerating Codes.

is thus represented by a vector of $l_{frag} = (|fragment|/q)$ elements. The whole file is thus represented by the matrix of the original fragments with size (n_{file}, l_{frag}) and denoted as $O_{n_{file},l_{frag}}$. A set of n parity fragments is represented as a (n, l_{frag}) matrix $P_{n,l_{frag}}$, this matrix can be always represented as a set of linear combinations of the original fragments:

$$P_{n,l_{frag}} = C_{n,n_{file}} O_{n_{file} \times l_{frag}}$$

where $C_{n,n_{file}}$ are elements in the field and represent the coefficients associated with the set of parity fragments.

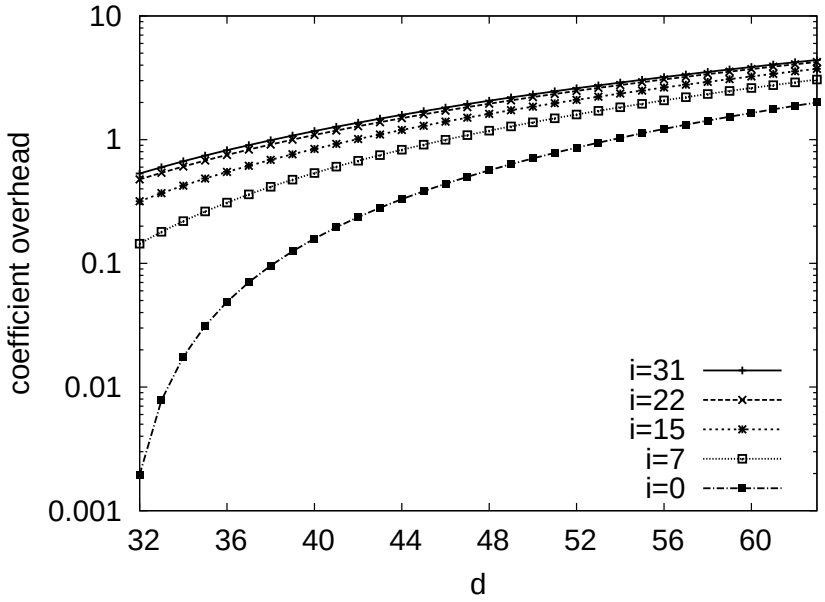


Figure 4.5: Coefficient overhead of $RC(32, 32, d, i)$ for a 1 MByte file.

4.7.1 Impact of coefficients

The first question we address is the impact of the coefficients on the storage and communication costs. Since every fragment is associated to a set of n_{file} coefficients, the relative impact of the coefficients is given by the ratio:

$$r_{coeff} = \frac{n_{file} q}{|fragment|} = \frac{n_{file}^2}{|file|} \cdot q \tag{4.9}$$

this ratio can be interpreted as the overhead due to coefficients: for every bit of data we need r_{coeff} bits of coefficients. Note that this ratio is inversely proportional with the size of the file we store, this means, as one could expect, that the bigger the file the smaller is the coefficient overhead. More importantly, the overhead increases with the square of n_{file}, which increases significantly as we increase the parameters d and i in Regenerating Codes (see eq. 4.7).

To understand the impact of this additional cost, let us consider the class of regenerating codes $RC(32, 32, d, i)$ and let us assume that the field size is $q = 16$, which corresponds to an element size of 2 bytes. In Fig. 4.5 we plot the values of the coefficient overhead when the original file size is $|file|$=1 MByte for all the possible values of d and i.

For such a small file size, the coefficient overhead is non negligible: in the "most expensive" configuration *for 1 bit of data, more than 4 bits of coefficients are needed*, which is clearly unacceptable. By increasing the file size, this overhead decreases (see eq. 4.9). The implication of these results is that when using Regenerating Codes, system designers need to choose a minimum size for storage objects that is significantly bigger than for traditional erasure codes.

4.7.2 Computational Complexity

One of the main concerns when coding is used in real systems is the computational cost they introduce. In this section we propose a formal analysis of the costs of Random Linear Regenerating Codes.

All the operations are performed in a Galois Field. Therefore, we need to make sure to control the cost of the operations by choosing the right field size. If we set the field size equal to 2^q, with $q = 16$ all the operations are performed on unsigned short integers (2 bytes). In this case

- Additions and subtractions correspond to an XOR operation between two elements.
- Multiplication and division are performed in the log-space. For example: $a \cdot b$ becomes $\exp(\log a + \log b)$. $\log$ and $\exp$ for all the possible values in the field are computed *offline* and stored, which requires 256 KB of memory for $q = 16$. The operations $\log$ and $\exp$ can then be implemented as value lookups in a table, which allows to implement division and multiplication in 3 lookups and 1 addition.

All the operations we perform for Regenerating Codes can be reduced to: (1) Linear Combinations and (2) Matrix inversions. Let us analyze them in details:

1. A linear combination of n vectors of length l consists of $n \cdot l$ additions and $n \cdot l$ multiplications for a total of $5nl$ operations.
2. The inversion of a square (n, n) matrix consists of n^3 additions and n^3 multiplications that can be implemented $5n^3$ operations. Actually, for Regenerating Codes the situation is slightly different: we have a (m, n) matrix, $m \geq n$ from which we need to *extract* n rows that are linearly independent, which will result in a (n, n) submatrix that can then be inverted. Extraction and inversion are done in parallel and the cost will vary accordingly to the particular matrix between the bounds $5n^3$ and $5mn^2$.

Now we have all the basic tools to compute the complexity of Regenerating Codes along the lifetime of a file:

1. **Insertion:** In this phase we perform $(k + h) \cdot n_{block}$ linear combinations of n_{file} original fragments for a total number of operations equal to:

$$CPU(encoding) = 5(k + h) \cdot n_{file} \cdot n_{block} \cdot l_{frag}$$

   Using the definitions of the different parts we obtain:

$$CPU(encoding) = \frac{5}{2}(k + h) \cdot n_{block} \cdot |file| \tag{4.10}$$

2. **Maintenance:** As already explained, in a repair, part of the work is done on the participating peers and another part is done on the newcomer. On every participating peer we perform one linear combination of n_{block} parity fragments, which corresponds to:

$$CPU(repair)_{up} = 5 \cdot n_{block} \cdot l_{frag}$$

doing some manipulations we obtain that the number of operations is proportional to the size of the parity blocks expressed in bytes:

$$CPU(repair)_{up} = \frac{5}{2} \cdot |block| \tag{4.11}$$

On the newcomer we perform n_{block} linear combinations of d parity fragments, which corresponds to:

$$CPU(repair)_{down} = 5 \cdot d \cdot n_{block} \cdot l_{frag} = d \cdot CPU(repair)_{up} \tag{4.12}$$

Note that every parity fragment is also associated with a set of coefficients. This means that every time that a new parity fragment is generated as a linear combination of other existing parity fragments, this linear combination must be performed also on the corresponding coefficients, in order to obtain the coefficients associated with the new parity fragment. In terms of computation cost, this can be taken into account assuming that the fragment size is virtually increased by the size of coefficients, which is given by the overhead in section 4.7.1.

3. **Reconstruction:** We can split the reconstruction in two phases: (1) we need to extract n_{file} linear independent rows from a $k \cdot n_{block} \times n_{file}$ matrix, and then invert the obtained submatrix (2) We multiply this submatrix by the correspondent parity fragments. According to these two phases, the cost of reconstruction can be split in two components as well:

$$CPU(reconstruction) = CPU(inversion) + CPU(decoding)$$

As explained before the cost of the inversion is bounded by two limits:

$$5 \cdot n_{file}^3 < CPU(inversion) < 5 \cdot k \cdot n_{block} \cdot n_{file}^2 \tag{4.13}$$

The decoding, then, corresponds to n_{file} linear combinations of n_{file} parity fragments, which leads to:

$$CPU(decoding) = 5 \cdot n_{file}^2 \cdot l_{frag} = \frac{5}{2} \cdot n_{file} \cdot |file|$$

Note that all the costs, except from the inversion cost, are linearly dependent to the file size $|file|$ (This holds also for repair, since $|block|$ is in turn proportional to $|file|$).

4.8 Experimental Evaluation

In this section we evaluate the resource requirements of Regenerating Codes. For this purpose, we wrote an optimized C implementation of Random Linear Regenerating Codes that we executed on an Intel Core 2 Duo CPU at 2.66GHz.

We execute all the operations performed in the life cycle of a stored file, as described in section 4.7, and measure the time needed to perform these operations. All the experiments have been done for a file of 1 MByte in size and the Regenerating Code parameters are fixed to $k = 32$, $h = 32$, and can take all possible values for i and d.

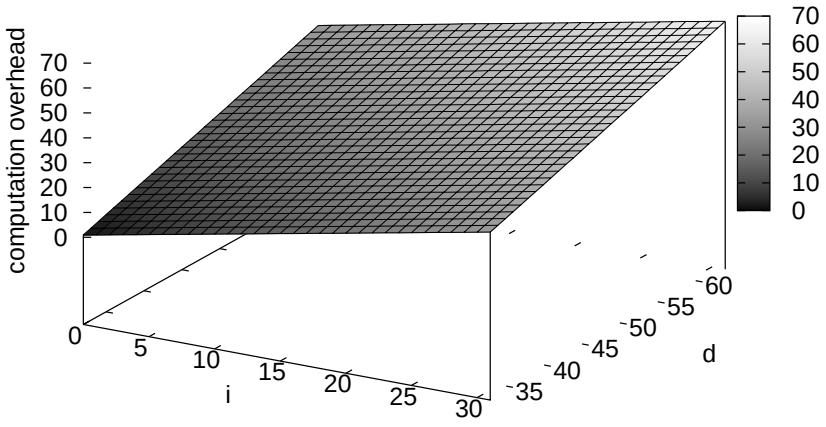
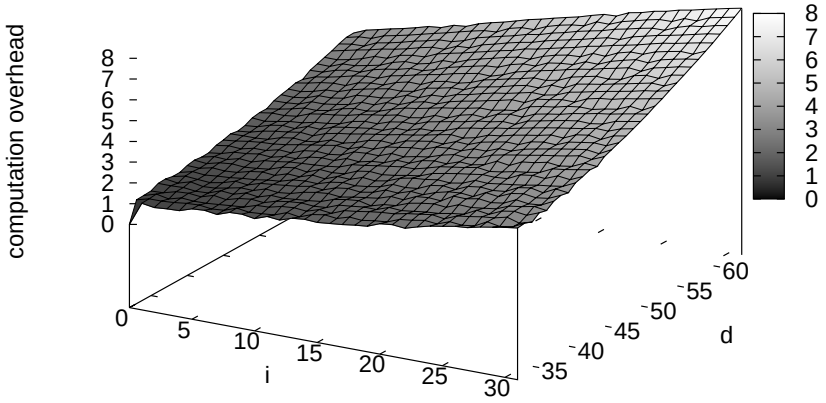
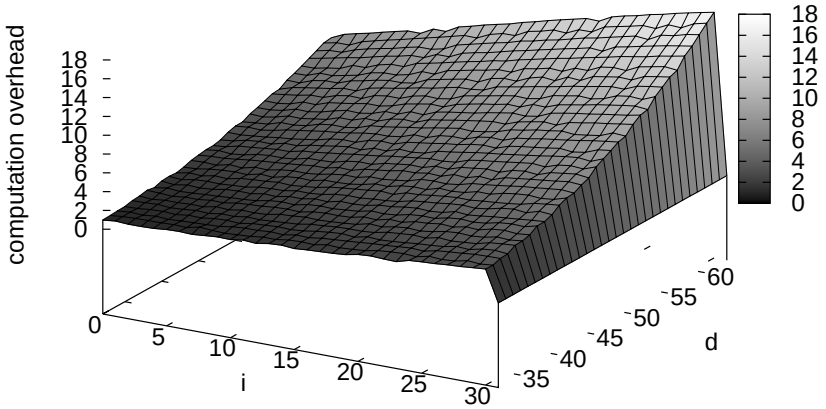


Figure 4.6: Encoding computation overhead for $RC(32, 32, d, i)$.



(a) Participant side.

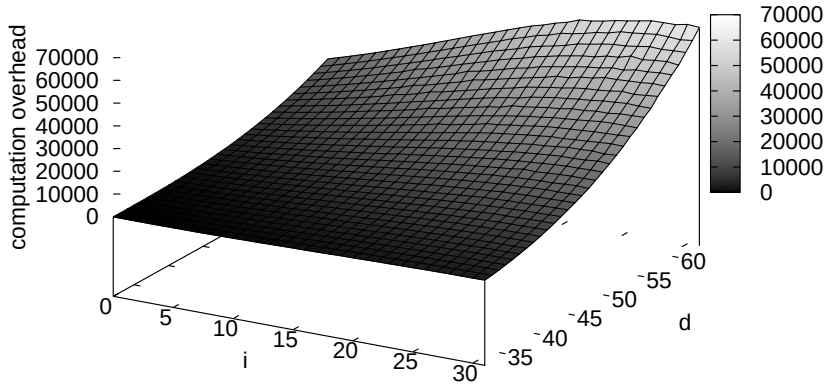


(b) Newcomer side.

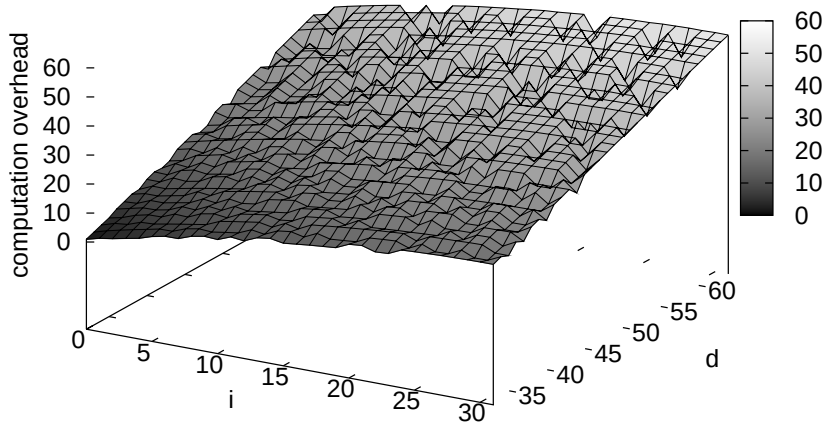
Figure 4.7: Repair computation overhead for $RC(32, 32, d, i)$.

4.8.1 Computational Cost

To have a basis for comparing different configurations of Regenerating Codes, we first show the results obtained for a traditional erasure code, (i.e. a Regenerating Code with



(a) Matrix Inversion.



(b) Decoding.

Figure 4.8: Reconstruction computation overhead for $RC(32, 32, d, i)$.

$RC(32, 32, 32, 0)$) when a file of 1 MByte is stored. Let $t_{d,i}$ denote the time needed by a particular operation for a Regenerating Code $RC(32, 32, d, i)$. Table 4.2 shows the time $t_{32,0}$ needed for each operation.

	$t_{32,0}[sec]$
Encoding	0.52
Participant Repair	0
Newcomer Repair	0.01
Matrix Inversion	0.002
Decoding	0.25

Table 4.2: Time needed for operations by a (32,32) traditional erasure code for a file of 1 Mbyte.

Note that the participant repair has a computation time of zero because in traditional erasure codes repairs do not require any computation at the participant side, which simply sends to the newcomer its entire parity block.

Let us now introduce the results obtained for the general case of Regenerating Codes $RC(32, 32, d, i)$. To understand the computational overhead of these codes, we consider the ratio between the time $t_{d,i}$ and the time $t_{32,0}$ measured for traditional erasure codes. We call

this ratio **computation overhead** *coh*:

$$coh_{d,i} = \frac{t_{d,i}}{t_{32,0}}$$

The computation overhead says how much a given Regenerating Code is slower than a traditional erasure code. Following the life cycle of a file:

1. **Insertion:** we show in Fig. 4.6 the computation overhead of the initial encoding of the file. The plot shows that the overhead grows linearly with i and d. This is consistent with eq. 4.10, which says that the cost is proportional to n_{block}, which is in turn linear with d and i as one can see from eq. 4.8.

2. **Maintenance:** Fig. 4.7a shows the computation overhead on the participant side[7], in this case the computation overhead grows slightly more than linearly with d and i, since as we know from eq. 4.11 it is proportional to the parity block size, which in turn has the behavior shown in Fig. 4.3a. Fig. 4.7b shows the computation overhead on the newcomer side. From eq. 4.12, this cost is proportional to d times the cost on the participant side, which is confirmed by the roughly quadratic relation with d shown by the plot. Note that for $i = k - 1$ the overhead falls to zero, since for this configuration the newcomer does not need to combine the received parity fragments, but simply stores them (cf. section 4.6.2).

3. **Reconstruction:** The reconstruction requires the inversion of the matrix coefficients and then the decoding of the fragments. Fig. 4.8a shows the computation overhead for the inversion, which as we know from eq. 4.13 grows roughly as n_{file}^3. Inversion can be computationally very expensive, in particular for large values of d and i. Fig. 4.8b shows the computation overhead of the decoding, whose shape closely resembles the one for encoding (see Fig. 4.6), which is expected since both perform analogous operations.

4.8.2 Bottleneck Network Bandwidth

As outlined in section 3.3, a redundancy scheme introduces three different costs, namely computation, storage and communication. So far we have only considered computation. However, what it is really interesting is to evaluate which resource (computation or communication) is the overall performance bottleneck of the system.

In a distributed storage system the data handled must be *transferred over the network*. Let us assume that the transfer operation is pipelined with the coding, which means in the case of insertion that each parity fragment is transmitted as soon as it is produced by the initial encoding step. If the transfer takes longer than the computation, then the *bottleneck is communication*, and the use of a computationally more efficient code will not make the insertion operation faster. This means that whether or not computation has an impact on the overall performance of the system depends on the available network bandwidth of the participating

[7] Note that this cost is equal to zero in traditional erasure codes, for this reason the normalization is done by the smallest value larger than zero which occurs for $d = 33$ and $i = 0$ and is equal in terms of computation time to 0.0003 sec.

peers. For this purpose we want to know the minimum network bandwidth of a peer, for which the computation represents the bottleneck for the overall performance. We call this bandwidth **bottleneck network bandwidth**, which is denoted as *bnb*.

The bottleneck network bandwidth can be computed as the bandwidth for which the transfer time is equal to the computation time. If $t_{d,i}$ denotes the time needed to perform an operation for $RC(32, 32, d, i)$ and $|data|_{d,i}$ denotes the amount of data handled by that operation that need to be transmitted over the network. We have:

$$bnb_{d,i} = \frac{|data|_{d,i}}{t_{d,i}}$$

From the above definition it is clear that the bottleneck network bandwidth also gives the *amount of data that can be processed by the coding/decoding operation*.

The values of $|data|_{d,i}$ for the different operations are computed as follows:

- **Encoding:** This operation produces the $(k+h)$ initial parity blocks. The amount of data produced that is sent over the network is given by the size of these blocks:

$$|data| = (k + h) \cdot |block|$$

- **Participant Repair:** This operation produces a single parity fragment plus the corresponding coefficients. The amount of data that is sent over the network is:

$$|data| = (1 + r_{coeff}) \cdot |fragment|$$

- **Newcomer Repair:** This operation produces a new parity block and his coefficients from d received parity fragments and their coefficients. The amount of data that is *received* from the network is given by the size of d fragments plus the corresponding coefficients:

$$|data| = (1 + r_{coeff}) \cdot d \cdot |fragment|$$

- **Inversion:** This operation extracts n_{file} independent rows form the received $(k \cdot n_{block}, n_{file})$ matrix (which describes the k parity blocks used for reconstruction), and inverts the submatrix obtained. This means that the amount of data that is received for this operation is given by the size of the coefficients of the k parity blocks:

$$|data| = k \cdot r_{coeff} \cdot |block|$$

- **Decoding:** This operation produces the original file by multiplying the matrix obtained from the inversion by the correspondent parity fragments. The amount of data that is received for this operation is given by the size of n_{file} fragments, i.e. the file size:

$$|data| = |file|$$

Table Table 4.3 picks some significant values of the parameters d and i and shows the volume of repair traffic $|repair_{down}|$ and the total amount of data stored in the system $|storage|$,

	d	i	$\|repair_{down}\|$	$\|storage\|$
Traditional EC	32	0	1MB	2 MB
Extreme RC	63	30	42.47 KB	2.61 MB
Reasonable RC 1	32	30	62.18 KB	3.76 MB
Reasonable RC 2	40	1	128.40 KB	2.006 MB

Table 4.3: Communication and storage costs for some $RC(32, 32, d, i)$ for a 1 MByte file.

		Bottleneck Network Bandwidth				
d	i	**Encoding**	**Repair**		**Reconstruction**	
			Participant	**Newcomer**	**Matrix Inversion**	**Decoding**
32	0	31.2 Mbps	∞	777.3 Mbps	7.8 Mbps	24.6 Mbps
63	30	655 Kbps	11.0 Mbps	10.2 Mbps	383 Kbps	482 Kbps
32	30	1.9 Mbps	21.6 Mbps	21.6 Mbps	1.6 Mbps	1.3 Mbps
40	1	3.1 Mbps	70.5 Mbps	76.8 Mbps	1.5 Mbps	2.5 Mbps

Table 4.4: Resource requirements of $RC(32, 32, d, i)$ for a 1 MByte file.

while table Table 4.4 shows the corresponding bottleneck network bandwidths for all the operations in the life cycle of a file.

The first row with $d = 32, i = 0$ presents the results for a traditional erasure code, which minimizes the *storage* requirement at the expense of a very large volume of repair traffic ($|repair_{down}| = |file|$). In the second row we consider an extreme regenerating code with $d = 63$ and $i = 30$, which *minimizes the repair traffic*. However, as we showed in figures 4.6, 4.7, and 4.8, this particular code has the highest computational costs and results in bottleneck network bandwidth values that can be as low as a few hundred Kbps.

However, if we remember the results presented in Fig. 4.3b, which show the savings in repair traffic for Regenerating Codes, we know that most of the savings are already achieved by quite small values of d, i.e. where $d = k$ or where d is slightly larger than k. For this reason, the next two rows of table 4.4 consider two *reasonable* Regenerating Codes with values of $d = 32$ and $d = 40$ that illustrate how we can trade off storage requirement and repair traffic:

- If we have plenty of storage space, we can use a big value for the block expansion index i: For $d = 32, i = 30$ the storage space required as compared to the one required by traditional erasure codes almost doubles. However, the reduction in repair traffic as compared to traditional erasure codes is still almost as good as for the Regenerating Code with $d = 63, i = 30$, which minimizes the repair traffic.

- On the other hand if storage space matters, we can choose a code with a small i, and a d slightly larger than k, which still preserves most of reduction in repair traffic. Results for $d = 40, i = 1$ are shown in the fourth row of table 4.4. If we compare the results to the best one achievable for each resource (see first two rows), we see that we achieve a close to minimal storage requirement (2.006 MB vs. 2.0 MB), and a repair traffic (128.40 KB) that is almost one order of magnitude less than for traditional erasure codes.

From the results presented so far, we can conclude that Regenerating Codes, as compared to traditional erasure codes, can provide substantial reductions in repair traffic, at a small extra

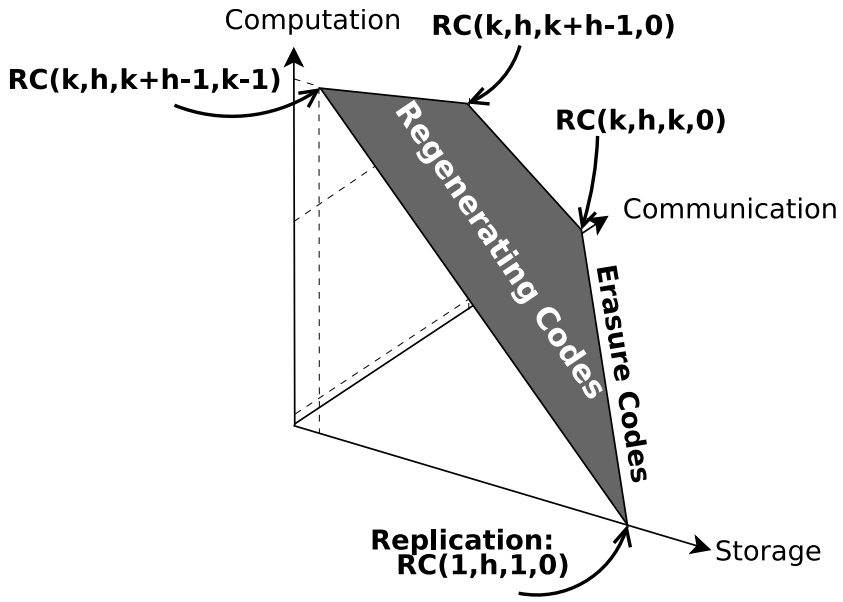


Figure 4.9: Illustration of the trade-offs provided by Regenerating Codes.

cost in terms of storage space required. However, this gain comes at the price of a much higher computational cost as can be seen when looking at the encoding and reconstruction costs, which are nearly one order of magnitude higher than for traditional erasure codes. With the current implementation, we can encode/decode in the order of 1 GByte of data per hour.

This performance may be too low for a large data center. Therefore, Regenerating Codes are best suited for those systems that do not insert or retrieve very large amounts of data and that need to do a significant amount of repairs: An example is given by peer-to-peer data backup systems where the data maintenance due to the high node churn, is far more frequent than data insertion or retrieval.

Moreover, if higher encoding/decoding rates are required, one can consider delegating encoding and decoding to particular peers equipped with (i) higher computation power or (ii) exploit the capabilities offered by GPUs (Graphic Processing Units), as has been advocated recently [22].

4.9 The impact of *d* on the repair policies

Before concluding the chapter, we discuss in this section one practical aspect of the use of Regenerating Codes in a real system. In particular we discuss briefly how the parameter d may impact the repair policies.

As explained before, maintenance consists of building periodically the redundancy lost due to peer failures. One crucial point in the design of such a system is the repair policy, i.e. the mechanism used to decide when to do repairs.

From a theoretic point of view, to provide data durability, a repair is needed only when a peer storing data fails permanently, i.e. he abandons the system or deletes what he stores. Doing this is not trivial for two basic reasons: (1) it is impossible to distinguish between permanent and transient failures, (2) even if we have an oracle able to tell apart permanent and transient failures, when a repair is triggered we might not be able to perform it because not enough parity blocks are available. In [51] it is argued that this means that it is practically unfeasible to guarantee durability without availability.

A classical way to design a repair policy is threshold based, i.e. a repair is triggered when the number of available parity blocks goes below a certain threshold. To provide data availability (and durability), this threshold *TH* is such that in any moment the number of available parity blocks is above k: $TH > k$, the actual value depends on how *far* from data loss the system should run.

In traditional codes, this implies that in any moment there are enough parity blocks to perform a repair. However this property is not always assured for Regenerating Codes when $d > k$. Indeed there might be cases in which the number of available parity blocks is between k and d preventing to perform the repair.

To face this problem, we have two alternative solutions (1) Increasing the threshold to guarantee that d parity blocks are always available (2) Changing d on the fly accordingly to the actual number of available parity blocks.

The first solution is always possible, but as we will show for Hierarchical Codes in chapter 5, it produces a higher number of repairs that could annul the positive effect of Regenerating Codes with respect to bandwidth consumption.

The second solution requires a more detailed discussion. As described in section 4.5, given the parameter i, the choice of d determines the minimum amount of data that needs to be sent upon a repair. Let us assume that in the design phase we fixed a particular value for d and we set n_{file} and n_{block} through equations 4.7 and 4.8. If, upon a repair, we need to use a value $d' < d$, in theory we need only to increase the amount of data transmitted accordingly to this new value. In practice, when Random Linear Regenerating Codes are used, this is not straightforward. The reason for this is that linear codes require that all the operations are performed over fixed size fragments, whose size depends on n_{file}. In other words since in linear codes *data are not liquid*, we cannot increase arbitrarily the amount of data transmitted, this quantity instead is *quantized* by the size of the fragments.

The correct procedure to perform a repair with d' peers for a Regenerating Codes $RC(i, d)$ is thus the following:

- Every participating peer computes through eq. 4.3 the quantity $|repair_{up}|_{i,d'}$, which is the amount of data that the peer would transfer to the newcomer, if the regenerating code was originally designed with a repair degree d':

$$|repair_{up}|_{i,d'} = r(i, d') \cdot |file|$$

- Every participating peer computes n_{repair} linear combinations of the parity fragments it is storing, where n_{repair} is such that

$$n_{repair} \cdot |fragment|_{i,d} \geq |repair_{up}|_{i,d'}$$

Note that if $d' = d$ then $n_{repair} = 1$.

- The newcomer receives $n_{repair} \cdot d$ parity fragments and performs n_{block} linear combinations of them to obtain the new parity block.

Even if we did not address this apsect in the experiments we proposed in the previous section, an important remark is that this procedure does not only increase the communication cost, but also the computation cost, since more linear combinations need to be done on the participant side and more fragments need to be combined on the newcomer side.

4.10 Conclusion

Regenerating Codes can be seen as a generalization of previously known redundancy schemes based on replication and erasure codes. They allow to trade off not only communication and storage requirements, but also computational costs. We schematically depict this trade-off in figure 4.9.

We proposed a practical implementation of Regenerating Codes, based on Random Linear Codes. We evaluated its performance trade-offs. We showed that the important savings provided in terms of repair traffic do not come for free, as Regenerating Codes have much lower coding and decoding rates.

However, we feel that Regenerating Codes have a lot of potential in environments where repairs of lost blocks are frequent and the available bandwidth to carry the repair traffic is limited, as is for instance the case in Internet-wide peer-to-peer backup systems.

CHAPTER 5
Hierarchical Codes

5.1 Introduction

In chapter 4 we discussed and evaluated Regenerating Codes. We showed how they present a new trade-off that involves not only storage and communication but also computation. In this chapter we propose a new class of codes, called Hierarchical Codes, which pursue the same objective of Regenerating Codes, i.e. to provide the same storage efficiency as traditional erasure codes, reducing at the same time the communication required by maintenance. However, Hierarchical Codes differ from Regenerating codes in the way they achieve this objective and do not suffer from the higher computation requirements of Regenerating Codes.

To introduce the basic idea behind Hierarchical Codes, we propose in section 5.2 the efficiency analysis of Fragment Replication. In section 5.3, we describe a formal tool called Information Flow Graph, which represents the evolution of data in a storage system based on linear codes and can be used to determine the minimum repair degree that must be used. Thanks to the Information Flow Graph, we can introduce in section 5.4 our Hierarchical codes and we can perform an efficiency analysis of them. Finally, in section 5.6 we test Hierarchical Codes through simulations using both, synthetic and real traces of the peer behavior. This chapter introduces also a number of theorems, whose proofs are all collected at the end of the chapter in section 5.8.

5.2 Erasure Codes vs. Fragment Replication

In this section we describe a replication scheme, called **fragment replication**, whose essence is to replicate the fragments of a file, instead of the whole file. Through a comparison between the performance of fragment replication and traditional erasure codes, we can introduce some of the ideas at the basis of our Hierarchical Codes.

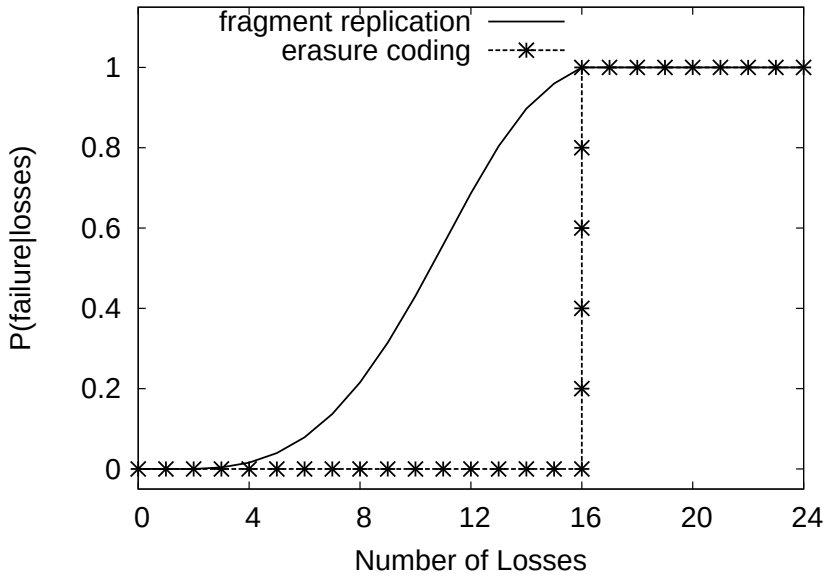


Figure 5.1: Fragment replication scheme compared to erasure codes with $k = 8$ and $N_{rep} = 3$. $P(failure|l)$ as function of the number of concurrent losses l.

Consider a file and cut it into k original fragments, then create N_{rep} replicas for every fragment and place every single replica on a different peer. Now the number of peers involved is $k \times N_{rep}$, the redundancy factor is β=N_{rep}. Whenever a fragment is lost, a repair operation will make a copy of another replica of the same fragment, which means that, as in the case of replication, the repair degree is d=1. The analysis of the reliability of this scheme is more complex. As introduced in section 3.3.1, it corresponds to the computation of the probability $P(failure|l)$. The problem is that in the case of fragment replication, there is not a single number that says how many peers we can lose without compromising the file as the survival of the file *depends on which particular peers fail*. In a very fortunate case we can lose all the redundancy, i.e. $k \times (N_{rep} - 1)$ fragments, and still be able to retrieve the original file, in the opposite case if all the replicas of a single fragment disappear, the file is lost when as few as N_{rep} fragments are lost. The probabilistic expression of the reliability is very helpful in this case. Exploring exhaustively all the possible combinations of losses for the case k=8 and N_{rep}=3, we obtain the solid curve in Fig. 5.1.

To compare the efficiency of fragment replication against the efficiency of erasure codes, let us consider an erasure code with k=8 and h=16, which leads to a redundancy factor β=3. This configuration is comparable with the example proposed for fragment replication, since k and β are the same. The starred curve in Fig. 5.1 shows that the reliability provided by coding is much higher: It can sustain always until 16 losses, while replication is able to do that only in a very small percentage of the cases. This example illustrates that fragment replication presents a reduced reliability but a very small repair degree, while erasure codes provide a high reliability, paying the price of a large repair degree.

If one wants to adopt fragment replication, instead of erasure codes, one has to accept the reduced reliability and has to be ready to perform repairs more often. This simple example raises a question, which the idea of hierarchical codes relies on: are there cases in which,

in spite of the larger number of repairs required by fragment replication, the saving in the repair degree results in a reduced communication cost?

More generally, Hierarchical Codes strive to investigate the trade-off between communication efficiency and reliability and try to provide operating regions in which the communication cost is reduced with respect to traditional erasure codes.

5.3 Repair degree in linear codes

To understand the way hierarchical codes are built it is essential to understand the theory behind *linear codes*. A general description of linear codes and random linear codes has already been given in section 4.6.1. What we discuss here is the mechanisms that determine the repair degree needed by these codes.

Consider a (k,h) traditional erasure code, based on a random linear implementation. When a parity fragment is lost and needs to be repaired, one may be tempted, to use less than k other parity fragments, reducing in this way the repair degree. However, only a repair degree of d=k is able to preserve the properties of the code. In particular for a repair degree of $d < k$, there will be sets of k parity fragments that are not sufficient to reconstruct the k original fragments.

This result can be derived from the literature about network coding [21, 66] and its application to distributed storage systems [42]. We will reformulate here some of the results in a slightly different but equivalent way, which will help us to derive Hierarchical Codes. In particular, in the next subsection we describe a tool to analyze the evolution of a linear erasure code, called Information Flow Graph.

5.3.1 The Information Flow Graph

An **Information Flow Graph** represents the evolution of the stored data across time. In particular, each node represents a fragment of data at a specific point in time t. The time evolves in discrete steps and every step corresponds to one or more losses and repairs. At the time $t = 0$ the graph is populated only by k source nodes representing the k original fragments denoted as $O = o_1, o_2, \ldots, o_k$. At time $t = 1$, the graph consists of $k + h$ nodes that represent the $k + h$ parity fragments initially inserted in the storage system and denoted as $P_1 = p_{1,1}, p_{2,1}, \ldots, p_{k+h,1}$. The graph at time $t = 1$ is referred to as the **code graph**. At $t > 1$, the graph is augmented at each step with $k+h$ nodes that represent the $k+h$ parity fragments present in the system at time t, which are denoted as $P_t = p_{1,t}, \ldots, p_{k+h,t}$. Connections between nodes are only possible among nodes of consecutive time steps and are always oriented from t to $t - 1$. The possible connections and their semantics are:

1. A generic node $p_{1,1}$ at time step 1 is connected to one or more original fragments, denoted as $R(p_{1,1})$. These connections are determined by the equations of the code used and for this reason the graph obtained is called **code graph**. In particular $R(p_{1,1})$ is the set of fragments linearly combined to produce $p_{1,1}$.

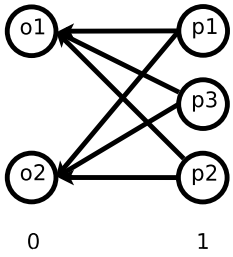


Figure 5.2: Example of a *Code Graph* for a classical (2,1)-linear erasure code. All the parity fragments p_1, p_2 and p_3 are obtained as a linear combination of the two original fragments o_1 and o_2.

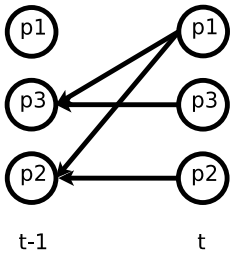


Figure 5.3: Example of one step of an *Information Flow Graph*. Parity fragments p_2 and p_3 have survived at time $t-1$. Parity fragment p_1 has been lost at time $t-1$ and has been repaired at time t combining the parity fragments p_2 and p_3.

2. A generic node $p_{i,t-1}$ in P_{t-1} can be connected to the node $p_{i,t}$. In this case node $p_{i,t}$ *must not be connected to any other nodes in* P_{t-1}. This means that the parity fragment p_i has survived at time $t-1$.

3. Alternatively, a generic node $p_{i,t-1}$ is not connected to any node in the following step. In this case node $p_{i,t}$ is connected to d nodes P^d_{t-1} in P_{t-1}, where $p_{i,t-1} \notin P^d_{t-1}$. This means that the parity fragment p_i has been lost at time $t-1$ and it has been repaired linearly combining the d parity fragments in P^d_{t-1}.

In Fig. 5.2 we show an example of code graph for a classical (2,1)-linear erasure code, while in Fig. 5.3 we show one generic step of the Information Flow Graph.

The *Information Flow Graph* we presented is a variant of the one proposed by Dimakis et al. [42]. It allows us to formulate the following **disjoint path theorem**, which follows Proposition 1 of [42]:
Theorem 1. *A selection of k nodes $P^k_t \subseteq P_t$, is sufficient to reconstruct the original fragments (with a probability that depends only on the size of the Galois Field in which the random coefficients are drawn), only if it is possible to find k* disjoint paths *from the k nodes in P^k_t to the k source nodes in O.*

The *disjoint paths* condition is obviously related to the choice of the repair degree d. The following proposition holds:
Proposition 1. *At any time t, any possible selection of k nodes P^k_t is sufficient to reconstruct the original fragments only if the* disjoint paths *condition is provided at time step $t = 1$ (by the* code graph*) and the repair degree is $d \geq k$.*

The proof is given in section 5.8.2 at the end of the chapter . The Proposition 1 requires that the *disjoint paths* condition be provided by the *code graph*. In that case this condition can be interpreted as the existence of a *perfect matching* between any selection P_1^k and the k source nodes in O. A *Random linear code* clearly provides this condition, since by design any node in P_1 is connected to all the source nodes in O.

5.4 Hierarchical Codes

The previous section showed that for a *traditional* linear erasure code the repair degree d cannot be smaller than k. Indeed, if $d < k$, there will be selections of k parity fragments that are *not* sufficient to reconstruct the original fragments. From this point of view, the fragment replication scheme presented in section 5.2 can be considered as a limit case of a (k,(N_{rep}-1)k)-code in which the repair degree is chosen to be $d = 1$. In this case, only a small subset of all possible choices of k parity fragments (replicated fragments) is able to reconstruct the original file, which may result in a lower reconstruction probability for a given number of losses as we saw in Fig. 5.1.

However, $d = 1$, which corresponds to fragment replication, and $d = k$, which corresponds to a traditional erasure code, are two limit cases. We believe that there is an interesting design space between these two limits that can be explored to find a better trade-off between storage efficiency and repair degree.

In the generic case of random linear codes, The naïve approach of using $d < k$ poses two main difficulties:

1. There is not an easy way to analyze the final reliability of the code, as we did in fragment replication.

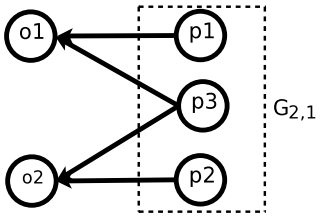
2. There is not a trivial policy for choosing the d parity fragments (to be combined) that are able to prevent a degradation of the reliability of the code through the maintenance process. Note that in fragment replication there is such a way: replace a lost replica with a copy of an identical one.

To overcome these difficulties we need to give a structure to the way we combine fragments. For this reason, we propose a new class of codes which we call **Hierarchical Codes**. A general instance of such a code can be generated through its *code graph* built according to the following procedure:

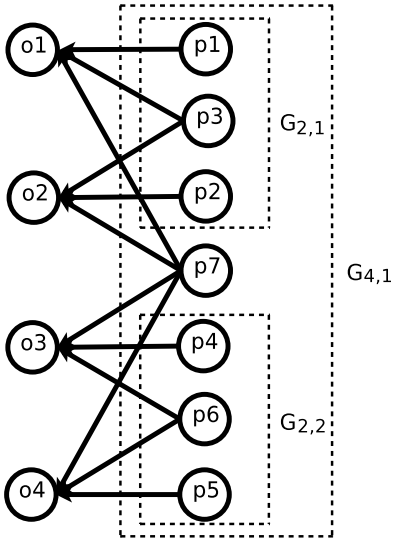
1. Choose two parameters k_0 and h_0 and build a (k_0, h_0)-code using the eq. 4.5 with the coefficients $c_{i,j}$ chosen randomly in $GF(2^q)$. If we set k_0=2 and h_0=1 we obtain the *code graph* in Fig. 5.4a.

   The generated parity fragments constitute a group denoted as $G_{d_0,1}$, where d_0=k_0 is the degree used to generate the fragments and it is called *combination degree*. In Fig. 5.4a, d_0=2.

2. Choose two parameters g_1 and h_1. Replicate the group structure $G_{d_0,1}$ for g_1 times to obtain g_1 groups denoted as $G_{d_0,1} \dots G_{d_0,g_1}$. Then add other h_1 parity fragments,



(a) Hierarchical (2,1)-code



(b) Hierarchical (4,3)-code

Figure 5.4: Samples of Code Graphs for Hierarchical Codes.

obtained combining (with random coefficients) all the existing $g_1 k_0$ original fragments O. This corresponds to a combination degree d_1=$g_1 k_0$=$g_1 d_0$. If we set g_1=2 and h_1=1 we obtain the *code graph* in Fig. 5.4b.

All the parity fragments constitute a group denoted as $G_{d_1,1}$, which corresponds to a hierarchical (d_1, H_1)-code, where H_1=$g_1 h_0$+h_1. The example in Fig. 5.4b is a hierarchical $(4, 3)$-code.

3. The previous step can be repeated several times, adding levels to the code. In the generic step s, we need to choose two parameters g_s and h_s. Replicate the structure of the group $G_{d_{s-1},1}$ for g_s times. Then add other h_s parity fragments, obtained combining all the existing original fragments, which corresponds to a degree d_s=$g_s d_{s-1}$. All the parity fragments constitute a group denoted as $G_{d_s,1}$, which corresponds to a hierarchical (d_s, H_s)-code, where $H_s = g_s H_{s-1}$+h_s.

5.4.1 Efficiency analysis of Hierarchical Codes

The redundancy factor β of a generic hierarchical (k, h)-code does not change with respect of a traditional erasure code: $\beta = (k + h)/(k)$. The other metrics are more complex.

Reliability The analysis of the reliability consists, as usual, in computing the probabilities $P(failure|l)$. These probabilities can be computed if we know what sets of k parity fragments are able to reconstruct the original fragments. Using Theorem 1 applied to the *code graph* we can state the following:

Proposition 2. *Consider P^k, a set of k parity fragments in the code graph of a hierarchical (k,h)-code.*

***If** the nodes in P^k are chosen fulfilling the following condition:*

$$|G_{d,i} \cap P^k| \leq d \ \ \forall G_{d,i} \ \text{belonging to the code} \tag{5.1}$$

which means that in P^k there can be a maximum of d parity fragments chosen from any group $G_{d,i}$,

***Then** the nodes in P^k are sufficient to reconstruct the original fragments.*

The proof is given in section 5.8.3. In the hierarchical (4,3)-code in Fig. 5.4b, the condition (5.1) means that no more than 2 parity fragments can be chosen from $G_{2,1}$, no more than 2 parity fragments can be chosen from $G_{2,2}$ and no more than 4 parity fragments can be chosen from $G_{4,1}$[1] .

Using Proposition 2 we can compute the generic probability $P(failure|l)$, exploring all the possible configurations of losses and check in each case if there is still a possible choice of parity fragments that allows reconstruction, as explained in section 5.9.

Repair Degree In the case of Hierarchical Codes, as in the fragment replication, there does not exist a single number that expresses the repair degree required. In particular, the repair degree required changes accordingly to which parity fragment needs to be repaired and which parity fragments are still alive. For each situation we would like to know which is the right choice to prevent the code from degrading, i.e. to preserve the guarantees provided by the code before maintenance, as described in the previous paragraph.

We can use again the Theorem 1 to formulate:

Proposition 3. *Consider an* Information Flow Graph *of a hierarchical code at time step t. Consider a node p repaired at time step t. Denote as $G(b)$ the hierarchy of groups that contains p and as $R(p)$ the set of nodes in P_{t-1} that have been combined to repair p.*

***If** $\forall t$ and $\forall p$, $R(p)$ fulfills the following conditions:*

$$|G_{d,i} \cap R(p)| \leq d \ \ \forall G_{d,i} \ \text{belonging to the code} \tag{5.2}$$

and

$$\exists\, G_{d,i} \in G(p) : R(p) \subseteq G_{d,i}, |R(p)| = d \tag{5.3}$$

where, (i) condition (5.2) means that in the set of parity fragments combined $R(p)$ there can be a maximum of d parity fragments chosen from any group $G_{d,i}$ and (ii) condition (5.3) means that there must exist a group in the hierarchy $G(b)$ that contains all the combined parity fragments and that their quantity has to be equal to the combination degree used in that group.

***Then** the code does not degrade, i.e. preserves the properties of the code graph expressed in Proposition 2.*

[1] This last constraint is unnecessary, since $G_{4,1}$ represents in this case the whole code.

The proof is given in section 5.8.4. For the hierarchical (4,3)-code in Fig. 5.4b, this means that the parity fragment p_1 can be repaired in one of the following two ways:

1. Using other 2 parity fragments belonging to the same group $G_{2,1}$, i.e. parity fragments p_2 and p_3.
2. Using other 4 parity fragments belonging to the whole group $G_{4,2}$, paying attention not to pick more than two parity fragments from the group $G_{2,2}$, for example parity fragments p_3, p_7, p_4, and p_6.

When a repair is performed, according to the parity fragment that needs to be repaired, multiple repair degrees are allowed. The repair degree that is actually used will depend on the parity fragments that are available on the moment of the repair[2].

Using the Proposition 3 and exploring all the possible combinations of losses, we can compute the probability $P(d|l)$. The procedure to compute this probability closely resembles the procedure to compute the failure probability $P(\textit{failure}|l)$ explained in section 5.9.

$P(d|l)$ indicates what is the probability that, if we have l concurrent losses, the repair of a parity fragment, in the *worst case*, requires a degree d. Note that *worst case* means that among the l parity fragments that we could repair, we decided to repair the one that requires the highest repair degree.

Note that the *worst case formulation* of $P(d|l)$ is quite pessimistic. In the reality, the particular repair performed depends on the repair policy and its repair degree can be lower than d.

We collected the results obtained for the hierarchical (4,3)-code in Fig. 5.4b in the table 5.1:

	l (losses)		
	1	2	3
$P(d=2\|l)$	0.86	0.42	0
$P(d=4\|l)$	0.14	0.58	0.77
$P(\textit{failure}\|l)$	0	0	0.23

Table 5.1: $P(d|l)$ and P($\textit{failure}|l$) as function of the number of concurrent losses l for the hierarchical (4,3)-code of Fig. 5.4b.

The first two rows show the repair degree probability, while the last row shows $P(\textit{failure}|l)$. This last row represents the cases in which the original fragments cannot be reconstructed. Note that these cases correspond also to the cases in which there is at least one parity fragment that cannot be repaired. For these last cases, thus, the failure probability replaces the probability $P(d|l)$, since the repair in the *worst case* cannot be performed. This is also the reason for which the values in each column sum up to 1. The table covers up to 3 losses, because for a higher number of losses it is clear that repairs are never possible and the failure probability is 1.

An alternative graphical representation of table 5.1 can be given by the histogram plot in Fig. 5.5. Every bar in the plot corresponds to a column in the table, while the height of the

[2]In the example of Fig. 5.4b, if p_1 needs to be repaired and either p_2 or p_3 is not available, the repair degree must be $d = 4$.

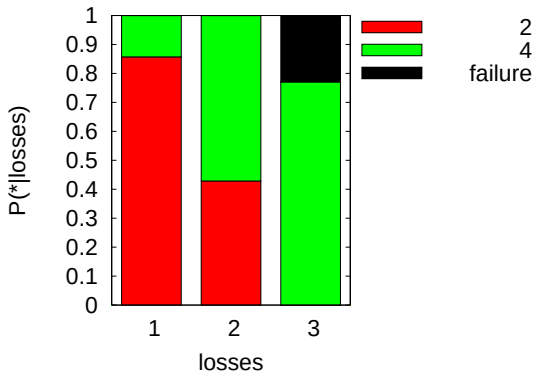


Figure 5.5: Examples of a Hierarchical (4,4)-code. $P(d|l)$ and P(*failure*|l) as function of the number of concurrent losses l.

sections in a bar represent the probabilities of repair degree or failure given the corresponding number of losses.

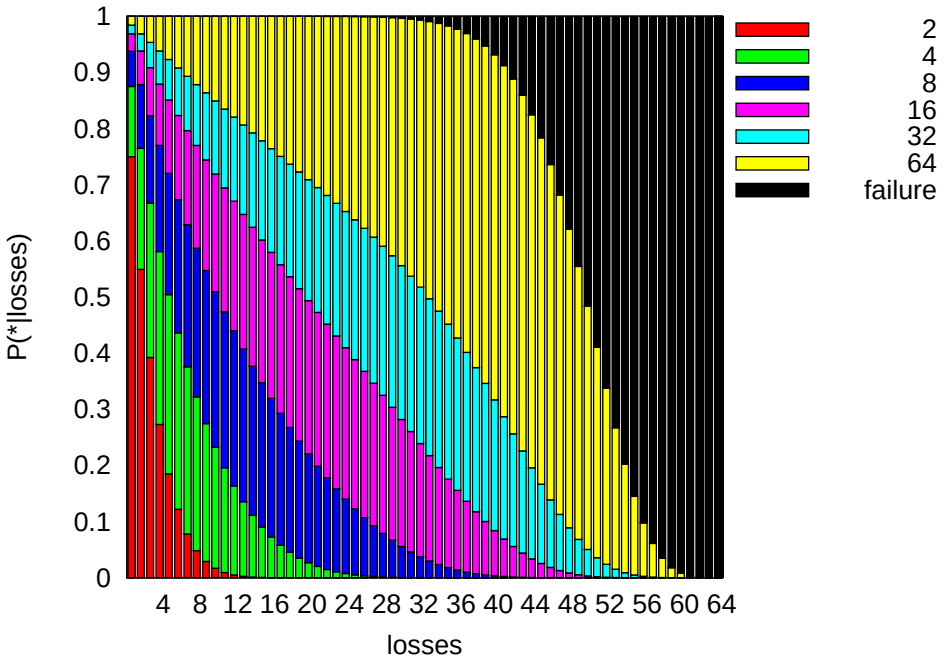
In Fig. 5.6a, we propose the graphical representation of the characteristics of a hierarchical code (64,64)-code, built using 6 levels and setting k_0=2, g_s=2 and h_s=1 for all the levels, except for the last level where h_5=2.

This figure nicely shows the properties of *Hierarchical Codes*. They are able to reduce the repair cost significantly: in a traditional (64,64)-code, the repair degree is always 64, while in this hierarchical (64,64)-code, it varies from 2 to 64. At a first look, the price to pay for this advantage seems to be reduced reliability; indeed a traditional (64,64)-code does never fail for fewer than 64 losses, while the hierarchical code may fail even for as low as 32 losses. However, by adjusting the repair policy, as we will explain in the next section, one can achieve the same reliability.

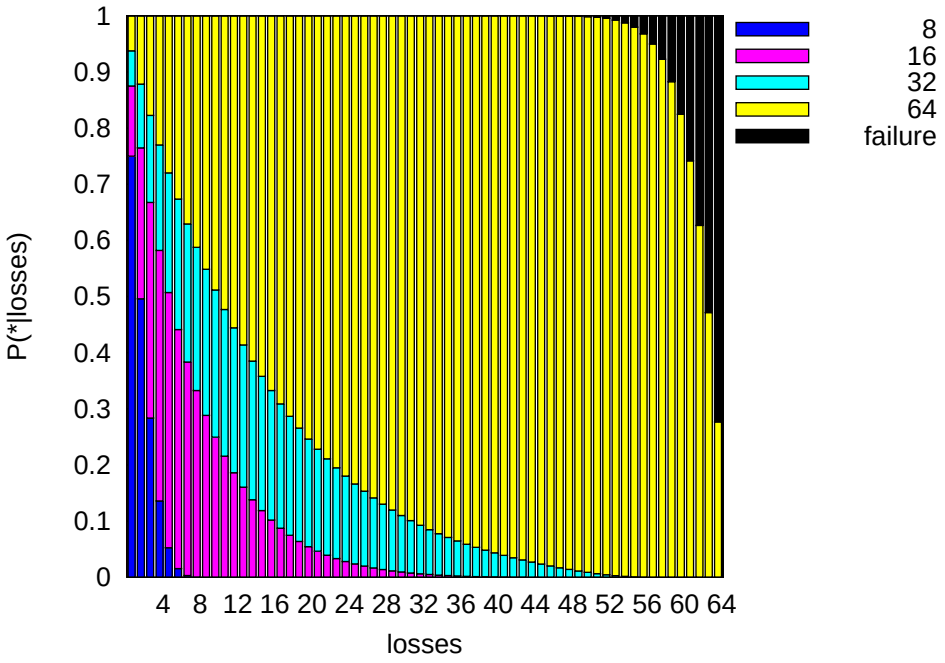
We believe that Hierarchical Codes give a new possibility to system designers to determine the right trade-off between costs and benefits with respect to the characteristics of the environment in which the system is going to operate. Note that different choices of the parameters $\{k_0, g_s, h_s\}$ produce different codes with the same level of redundancy, but with a different trade-off between reliability and repair degree.

In this sense, the configuration we proposed in Fig. 5.6a is just one of the many instances of Hierarchical (64,64)-codes. For some environments other choices of the parameters might be better. For example, if it is not acceptable to have a non-zero failure probability for 32 losses, one can choose the alternative configuration depicted in Fig. 5.6b. This configuration is built in 4 levels, setting k_0=8, g_s=2 and h_s=4 for all the levels, except for the last level where h_3=8. The histogram in Fig. 5.6b shows how failures occur for higher values of losses as compared to Fig. 5.6a. However, there is a price to pay in terms of a higher repair cost: the repair degree varies from 8 to 64 and the region corresponding to a repair degree of $d = 64$ is significantly larger.

In the present work we do not explore the tuning of the parameters, which we leave as future work. However, we will show through experiments how different parameter choices can impact the final cost.



(a) Hierarchical Code A.



(b) Hierarchical Code B.

Figure 5.6: Examples of Hierarchical (64,64)-codes. $P(d|l)$ and P($failure|l$) as function of the number of concurrent losses l.

5.5 Relation between Hierarchical Codes and LT Codes

Hierarchical Codes show some similarities with other classes of codes, such as Tornado Codes [70] and LT Codes [69]. We believe it is important to show these similarities and

clearly discuss the differences. In this section we refer in particular to LT Codes, however the concepts we discuss hold also for Tornado Codes and for all the other similar schemes.

The scenario in which LT codes are used is described as follows. There are k original symbols that need to be transmitted over a channel. These symbols are encoded in encoded symbols such that the receiver can reconstruct the original symbols as soon as it has received k of them (or slightly more). Every encoded symbol is the result of a XOR operation over a subset of the original symbols. The number of original symbols used is called combination degree and denoted with d.

In LT codes, for every new encoded symbol, the combination degree d and the original symbols to combine are chosen randomly. It can be shown that if d is drawn from a particular distribution (e.g. Soliton distribution), then with very high probability the following two properties hold:

1. Any k encoded symbols are sufficient to reconstruct the k original ones.
2. The decoding operation of any k encoded symbols consists of inverting a matrix which is triangular, i.e. which is very easy to invert.

When LT codes are used, the receiver waits to receive k encoded symbols and then it is able to reconstruct the original symbols very quickly. If some of the symbols are lost over the channel, the source will keep producing new encoded symbols *from the original symbols* until the receiver has received k of them.

The fact that LT codes produce encoded symbols from a subset of the original symbols, i.e. $d < k$, suggest that they are conceptually very similar to Hierarchical Codes. However, while LT codes combine original symbols in a completely random fashion, Hierarchical Codes define a precise tree structure, which confers to the code essential properties in the maintenance process. To understand this concept let us assume to use LT codes for storage purposes, as we do with Hierarchical Codes. In this case the k original symbols represent the original data we want to store, while the encoded symbols, say $k + h$, represent the redundant data we actually store on the peers. The question we need to ask is: what do we do when one or more of these k+h encoded symbols are lost because of peer failures? If we had the original symbols, it would be enough to draw a combination degree d from the Soliton distribution and combine the correspondent number of original symbols. The problem is that *the original symbols are not available*. This means that we have to guarantee that the distribution of the combination degrees follow the Soliton distribution by re-encoding already encoded symbols. Unfortunately there is no trivial way to do that.

Hierarchical Codes, thanks to their tree structure, are able, as described in the previous section, to define precise rules to recombine in case of losses already encoded blocks preventing a degradation of the parity blocks (i.e. preserving their original properties).

5.6 Experiments

The experiments are carried out via an event-driven simulator, which simulates a storage system for a set of peers whose behavior is described by availability traces that are provided as input.

The objective is to compare the reliability, the storage, and network communication costs of traditional erasure codes and *Hierarchical Codes*.

In both cases we chose a (64,64)-code. In particular, the traditional code is a Reed-Solomon code, while the Hierarchical Codes correspond to the two configurations presented in Fig. 5.6. This choice assures that the storage consumption is the same in all the scenarios[3].

The reliability provided depends on the repair policy adopted. We consider a hybrid timer/threshold policy. It assumes the presence of an entity able to monitor the availability of the participating peers and trigger a repair operation according to the following rules:

1. When a peer A disconnects and the number of available peers n is smaller or equal to TH: $n \leq TH$:

   perform immediately the repair of the parity fragment stored on peer A.

2. When a peer A disconnects and $n > TH$:

   wait for a time T and then if A is still unavailable perform a repair of the parity fragment stored on A.

The timer T is used to distinguish between transient and permanent failures. In the ideal case in which T is chosen as the maximum possible disconnection time of a peer, whenever a disconnected peer does not reconnect within T, we are sure that it has abandoned the system for ever. In the real world, disconnection times may be bigger than T. In such a case, the parity fragments stored on a reconnecting peer are discarded, because they have already been repaired[4]. This is a waste of resources that suggests to increase T. However, when T is increased, a higher number of peers is allowed to stay offline, in which case the set of online peers is not able to reconstruct the original fragments or is not able to perform repairs. To be quite insensitive to the choice of T, we introduced also the threshold, which has to be such that *availability* is provided, i.e. reconstruction is always possible.

In the case of Reed-Solomon codes, availability is provided if $n \geq k$, which requires that $TH > k$. we fix $TH = k + x$, which means that in the moment of minimum availability, i.e. in the moment of *maximum risk*, the system can still support x more losses. In the case of Hierarchical Codes, there is no a fixed threshold that guarantees availability. As shown in Fig. 5.6, the minimum number of online fragments that provides availability depends on the particular losses that occur in the system and varies, in the case of Fig. 5.6a, from 64 to about 96. To be comparable with Reed-Solomon codes, our approach is the following: whenever a loss occurs we recompute the probabilities $P(\textit{failure}|l)$, which indicate the probability of failure if additional l losses would occur, taking into account the specific losses that al-

[3] It is β=2, which means that every object consumes a space twice its size.

[4] For Hierarchical Codes, reintegration of this parity fragment is in some cases possible and would increase significantly the efficiency. However, identifying such cases is not trivial and it is left as future work.

ready have occurred. If we want that, in the moment of maximum risk, the system can still support additional x losses, we apply the following rule: a repair is performed whenever $P(\mathit{failure}|a) > 0$.

Two notable facts are that

1. The repair policy for Hierarchical Codes needs to maintain a larger number of available parity fragments and tends to perform more repairs.
2. The guarantees in terms of availability in the case of Hierarchical Codes **are stronger**: for Reed-Solomon codes, at the moment of maximum risk, if additional x losses would occur, the object will be unavailable with probability 1, while in Hierarchical Codes, the object will be unavailable with a probability that can be much smaller than 1.

In the experiments, we test different environments changing the stability of peers, and we measure the number of parity fragment transfers needed to maintain the code. We chose $a = 10$ and T to be three times bigger than the average disconnection time. In any case, we performed other experiments that showed that changing these parameters does not influence significantly the results.

5.6.1 Experiments with synthetic traces

In this set of experiments, the peer behavior is synthetically generated. In particular, every peer behaves according to a simple Markovian model depicted in Fig. 5.7.

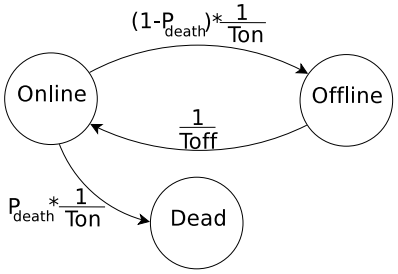
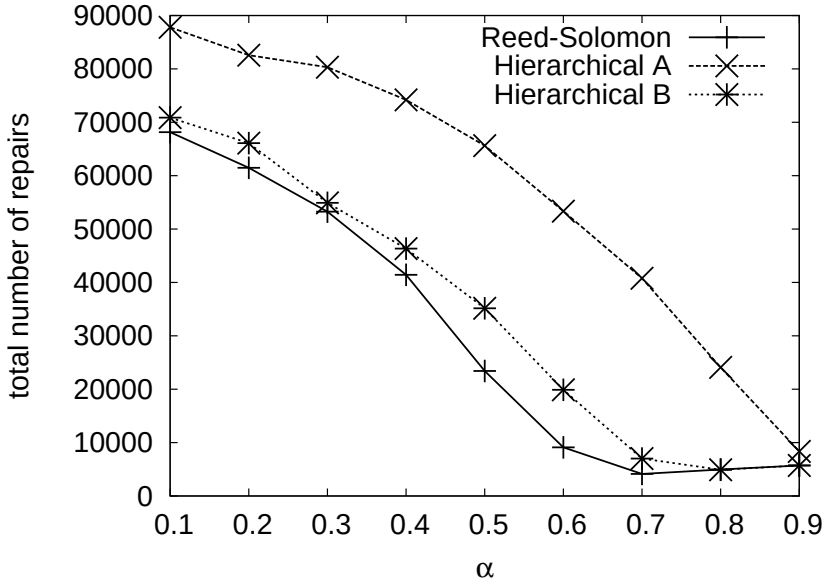


Figure 5.7: State machine modeling the behavior of a peer for Hierarchical Codes experiments.

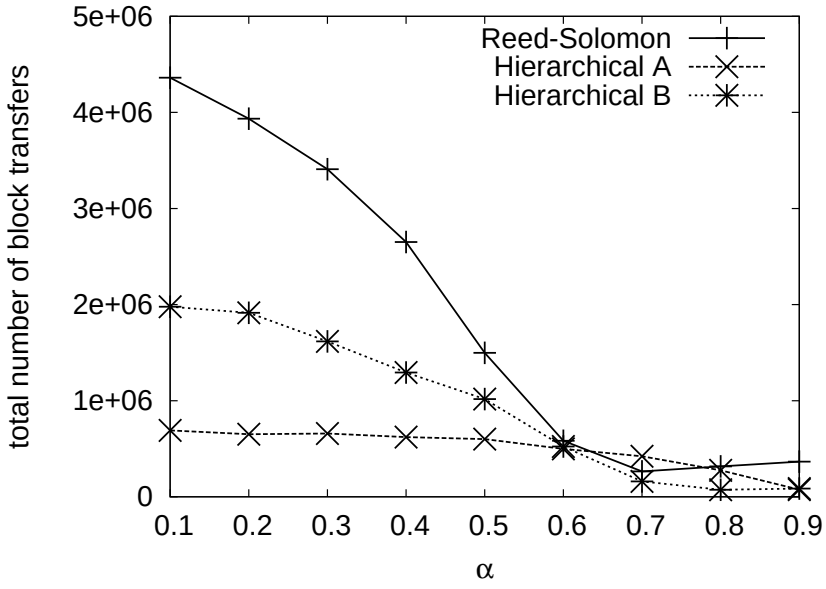
This model is very similar to the one introduced in section 3.2.2. A peer is available for an exponentially distributed time T_{on}, then upon disconnection it can abandon the system with probability P_{death} or can stay temporarily offline with probability $(1 - P_{death})$ for an exponentially distributed time T_{off}, after which it comes back online.

If one compares this model with the one in section 3.2.2 depicted in Fig. 3.1, one will notice that the parameter T_{life} has disappeared and has been logically replaced by the death probability P_{death}. However there is an easy mapping between the two parameters:

$$T_{life} = T_{on} + \frac{1 - P_{death}}{P_{death}}(T_{on} + T_{off})$$



(a) Total number of repairs.



(b) Total number of fragment transfers.

Figure 5.8: Cost of maintenance for Reed-Solomon and Hierarchical (64,64)-codes as function of the up ratio $\alpha = T_{on}/(T_{on} + T_{off})$. $a = 10$, $T = 3T_{off}$.

We tested our hierarchical (64,64)-codes and Reed-Solomon (64,64)-code, using different combinations of the three parameters. The results suggest that, while P_{death} does not have a strong influence, T_{on} and T_{off} are very important. In particular a fundamental role is played by the **up ratio** $\alpha = T_{on}/(T_{on} + T_{off})$, which represents the percentage of time that a peer spends online, or alternatively the ratio of peers that on average are online. It is clear that α has an influence on the number of repairs needed. This influence is different in Reed-Solomon codes and in Hierarchical Codes, since Hierarchical Codes need on average more peers to be online.

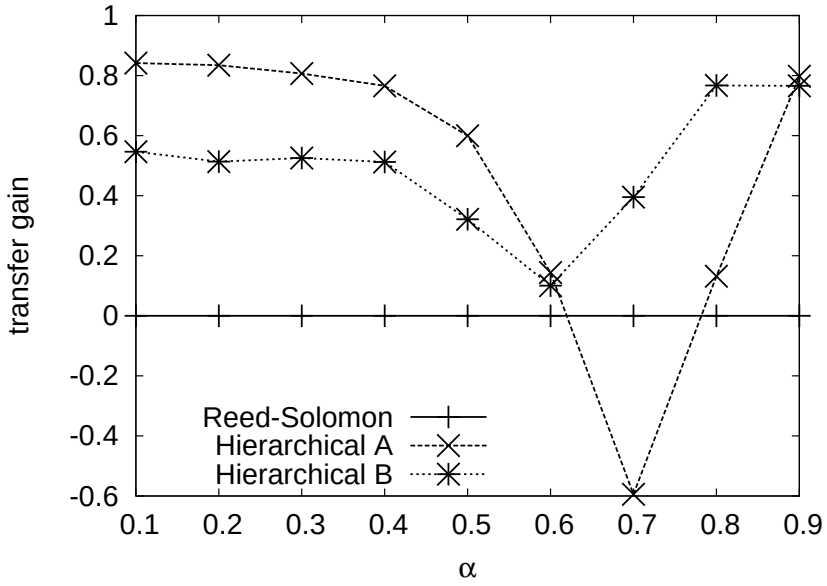


Figure 5.9: Gain of Hierarchical (64,64)-codes in terms of number of fragment transfers with respect of Reed-Solomon (64,64)-codes as function of the up ratio $\alpha = T_{on}/(T_{on} + T_{off})$. $a = 10$, $T = 3T_{off}$.

We run the simulation for 10000 time units, setting the disconnection time $T_{on} = 10$, the death probability $P_{death} = 0.001$ and selecting several values for T_{off} to test different values of the *up ratio*.

As already mentioned, we evaluate Reed-Solomon codes and the two instances of Hierarchical Codes shown in figure 5.6. Fig. 5.8 presents the results obtained; we label with 'Hierarchical A' the results obtained for the hierarchical code of Fig. 5.6a and with 'Hierarchical B' the results for the hierarchical code of Fig. 5.6b. In particular Fig. 5.8a shows the number of repairs needed by the three redundancy schemes, while Fig. 5.8b shows the total amount of fragment transfers executed for these repairs.

We see in Fig. 5.8a that, compared to Reed-Solomon codes, the hierarchical code 'A' requires a much larger number of repairs, while the hierarchical code 'B' requires only a slightly larger number of repairs. This is expected since (1) Hierarchical Codes need to be more reactive as they are more sensitive to losses (2) The trade-off between reliability and repair cost of the hierarchical code B in Fig. 5.6b is much closer to Reed Solomon with respect to the hierarchical code A of Fig. 5.6a, (i.e. a non-zero failure probability occurs for a number of losses closer to 64 and the repair degree is equal to 64 in a larger number of cases).

We also see that the number of repairs decreases when the *up ratio* increases. This is due to the fact that a higher *up ratio* corresponds to a higher percentage of peers on line, which in turn means fewer repairs.

The advantages of Hierarchical Codes are shown in Fig. 5.8b, where the actual communication costs introduced by the repairs is expressed in terms of the number of fragments transferred. Both hierarchical codes, in spite of their significantly higher number of repairs, generate in most of the cases much less fragment transfers than the Reed-Solomon code. In other words Hierarchical Codes require more repairs, but most of the repairs are quite cheap in terms of fragment transfers, which reduces the global repair traffic.

To have better understanding of the gain achieved in terms of fragment transfers, we compute the gain in terms of number of fragment transfers required by Hierarchical Codes with respect to Reed-Solomon codes. Formally, if we denote the number of fragment transfers of one instance of hierarchical codes as nt_{HC} and the number of fragment transfers of Reed-Solomon codes as nt_{RS}, the gain of the hierarchical code g_{HC} is defined as:

$$g_{HC} = 1 - \frac{nt_{HC}}{nt_{RS}}$$

Note that for this definition the gain of Reed-Solomon is equal to 0.

We show in Fig. 5.9 the values of the gain for the two instance of hierarchical codes. This picture clearly shows the trade-off offered by the two different Hierarchical Codes. The instance 'A' appears to be more suitable for small values of *up ratio*, i.e. when the nodes are more unstable, while for a more stable environment the reduction in repair traffic is smaller and in some cases (namely for $up = 0.7$) it is even higher than in Reed-Solomon codes. On the other hand, the hierarchical code 'B' produces a smaller gain for unstable environments (roughly 0.6 vs. 0.8), while it reduces the repair traffic much more in stable environments. Also this code always achieves a lower repair traffic than the Reed Solomon code (the curve is always below 1).

5.6.2 Experiments with real traces

To evaluate the codes for a wider set of operating conditions than the ones given by the synthetic traces, we also use availability data of real distributed systems. We use two different traces:

1. **KAD traces**: obtained crawling a KAD network. These traces [91] characterize the availability of about 6500 peers in the KAD network, sampling their status every 5 minutes for about 5 months.

2. **PlanetLab traces**: obtained monitoring the connectivity of PlanetLab nodes. These traces [56] describe the availability status of 669 nodes, which was obtained by means of pings sent every 15 minutes among all pairs of PlanetLab nodes, starting from January 2004 for about 500 days.

The table 5.2 depicts the results for the two traces comparing the total number of repairs and the total number of fragment transfers for the Reed-Solomon and the two instances of Hierarchical Codes.

The results confirm the trend that we saw in the previous subsection: *Hierarchical Codes require a higher number of repairs but result in a lower amount of fragments to be transferred*. Once again we see the impact of the two different configurations on the results.

5.7 Conclusion

We presented a new class of codes, called *Hierarchical Codes*, which offer a flexible way of adding redundancy in distributed storage systems. Hierarchical Codes combine the advan-

		Repairs	Transfers
PlanetLab	Reed-Solomon	472	30208
	Hierarchical A	637	4624
	Hierarchical B	487	6920
KAD	Reed-Solomon	765	48960
	Hierarchical A	3888	39710
	Hierarchical B	1072	20992

Table 5.2: Cost of maintenance for Reed-Solomon and Hierarchical (64,64)-codes using real traces of peer behavior.

tage of reduced repair traffic offered by replication with the higher resilience against failures offered by coding. We believe that Hierarchical Codes make coding a *practical* alternative to replication in peer-to-peer storage systems.

Experiments validated our claims, showing that for a given level of availability, a higher number of repairs needed by Hierarchical Codes results in most cases in a smaller amount of repair traffic.

Moreover, we saw that Hierarchical Codes with a given redundancy factor, allow to trade-off in multiple ways reliability and repair cost. Future developments will focus on better understanding these trade-offs, which will allow to determine the optimal configuration of the codes for a given environment.

5.8 Proofs

5.8.1 Useful lemmas

Lemma 1. *Consider an* Information Flow Graph *for a generic (k,h)-code at time step T. Consider a selection of k parity fragments P_1^k. Assume that there exists a condition **C** on this selection that guarantees that the original fragments can be reconstructed.*

***If** for any time step $t \leq T$, any selection of P_t^k that fulfills the condition **C** can be perfectly matched with a selection of k parity fragments P_{t-1}^k in time step $t-1$ that in turn fulfills the condition **C**,*

***Then** any selection P_T^k that fulfills the condition **C** allows the reconstruction of the original fragments.*

Proof. We proceed by steps:

step 1 Consider a selection P_1^k that fulfills the condition **C**. By assumption we know that the selection allows the reconstruction of the original fragments. This means, thanks to Theorem 1, that nodes in P_1^k have k distinct paths towards the original fragments F.

step 2 Consider a selection P_2^k that fulfills the condition **C**. By assumption we know that the nodes in this selection can be perfectly matched with a selection P_1^k that in turn fulfills the condition **C**. Thanks to previous step, we know that nodes in P_1^k have k distinct paths towards the original fragments O. This means that we can concatenate the perfect matching

between P_2^k and P_1^k and the k distinct paths between P_1^k and O, obtaining k distinct paths between P_2^k and O.

The last step can be repeated until the time step T, where thanks to Theorem 1, the lemma is proved. □

Lemma 2. *Consider a* code graph *of a Hierarchical Code. Consider a group $G_{d_s,i}$ and denote as $O_{d_s,i}$ the subset of original fragments that are connected with nodes in this group $G_{d_s,i}$. Consider a selection of nodes P_1^k and consider the subset of this selection that belongs to the group considered: $A_{d_s,i} = P_1^k \cap G_{d_s,i}$.*

If *$|A_{d_s,i}| \leq d_s$ and $\forall j : G_{d_{s-1},j} \subseteq G_{d_s,i}$, the nodes in $A_{d_{s-1},j}$ have already been perfectly matched with $|A_{d_{s-1},j}|$ nodes in $O_{d_{s-1},j}$,*

Then *it is possible to find a perfect matching between the nodes in $A_{d_s,i}$ and the nodes in $O_{d_s,i}$.*

Proof. Consider the nodes in $A_{d_s,i}$ that do not belong to the subgroups $G_{d_{s-1},j} \subseteq G_{d_s,i}$ and denote them as $\hat{A}$. Consider the fragments in $O_{d_s,i}$ that have not been matched with the nodes in the subgroups $G_{d_{s-1},j} \subseteq G_{d_s,i}$ and denote them as $\hat{O}$. The nodes in $\hat{A}$ are connected with all the nodes in $O_{d_s,i}$ and can be thus all matched with nodes in the subset $\hat{O}$, as long as $|\hat{A}| \leq |\hat{O}|$. Since nodes in the subgroups have already been matched, then $|A_{d_s,i}| - |\hat{A}| = |O_{d_s,i}| - |\hat{O}|$, where $|O_{d_s,i}| = d_s$. This implies that whenever $|A_{d_s,i}| \leq d_s$, $|\hat{A}| \leq |\hat{O}|$ and the perfect matching is possible. □

Lemma 3. *Consider an* Information Flow Graph *of a hierarchical code at time step t. Consider a selection P_t^k that fulfills the condition (5.1). Assume that a subset of α nodes $P_t^\alpha \subset P_t^k$ has already been perfectly matched with nodes in the previous step P_{t-1}^α that in turn fulfill the condition (5.1). Consider a node $p \in P_t^k \setminus P_t^\alpha$, i.e. that belongs to the selection but has not yet been matched.*

If *all the repairs in the graph are done fulfilling condition (5.2) and condition (5.3), and all the parity fragments $p_i \in P_t^\alpha$ are such that $|R(p_i)| \leq |R(p)|$,*

Then *it is possible to augment P_{t-1}^α with another node that is matched with p, without violating the condition (5.1) on the augmented set $P_{t-1}^{\alpha+1}$*

Proof. Let us use the following notation: $A_{d,i} = P_{t-1}^\alpha \cap G_{d,i}$ and $R_{d,i} = R(p) \cap G_{d,i}$. Assume that $G_{d_s,1}$ is the group in which condition (5.2) is fulfilled. This condition requires that $|R_{d_s,1}| = |d_s|$. Note that all the nodes in P_t^α have a repair degree $d \leq d_s$, which implies that all the nodes in $A_{d_s,i}$ are necessary matched with nodes in $P_t^\alpha \cap G_{d_s,1}$[5]. Since $p \in G_{d_s,1}$, thanks to condition (5.1), $|P_t^\alpha \cap G_{d_s,1}| < d_s$, which in turn implies $|A_{d_s,i}| < |d_s|$.

Consider two alternative cases:

case 1: $\exists j : 1 \leq j \leq g_s, |R_{d_{s-1},j}| > |A_{d_{s-1},j}|$: This means that there is a subgroup of the group $G_{d_s,1}$ (that belongs to $G(b)$) that has at least one *free* node that can be matched with the parity fragment p. Since $|R_{d_{s-1},j}| \leq |d_{s-1}|$, this node can be added without violating condition (5.1) and the lemma is proved.

[5]To be matched with a node p_o outside $G_{d_s,1}$, the repair degree of p_o must be bigger than d_s, which would violate the condition of the lemma.

case 2: $\forall j : 1 \leq j \leq g_s, |R_{d_{s-1},j}| \leq |A_{d_{s-1},j}|$: This means that there are no free nodes in the subgroups. This implies that: $\sum_{j=1}^{g_s} |R_{d_{s-1},j}| \leq \sum_{j=1}^{g_s} |A_{d_{s-1},j}|$. Consider the nodes in $A_{d_s,1}$ that do not belong to the subgroups and denote them as $\hat{A}$ (they are among the h_s additional nodes), then consider the nodes in $R_{d_s,1}$ that do not belong to the subgroups and denote them as $\hat{R}$. We can write $\sum_{j=1}^{g_s} |A_{d_{s-1},j}| = |A_{d_s,i}| - |\hat{A}|$ and $\sum_{j=1}^{g_s} |R_{d_{s-1},j}| = |R_{d_s,i}| - |\hat{R}|$. Since $|R_{d_s,1}| = |d_s|$ and $|A_{d_s,i}| < |d_s|$, we have that $|\hat{R}| > |\hat{A}|$. This means that there is at least one *free* node in $\hat{R}$ that can be matched with the parity fragments p without violating condition (5.1) and the lemma is proved. □

5.8.2 Proof of Proposition 1

Proof. Thanks to Lemma 1, proving Proposition 1 requires to prove that in a generic time step t, only if repairs are done with a repair degree $d \geq k$, then any selection of nodes P_t^k can be perfectly matched with a selection P_{t-1}^k.

Consider a *repaired* node $p \in P_t^k$. All the other $k-1$ nodes in P_t^k can be matched at most with $k-1$ nodes in P_{t-1}. If p has been repaired with a degree $d < k$, it is possible that all the nodes in $R(p)$ have already been matched with the $k-1$ nodes in P_t^k, preventing the matching of p. If $d \geq k$ there is at least one *free* node that can be matched with p. This can be repeated for all the repaired fragments proving, thanks to Lemma 1, the proposition. □

5.8.3 Proof of Proposition 2

Proof. Thanks to Lemma 1, proving Proposition 2 corresponds to prove that if a selection P_1^k is done fulfilling condition (5.1), then it is possible to find a perfect matching between the nodes in P_1^k and the original fragments in O. This can be proved using iteratively the Lemma 2 from the innermost group that nodes in P_1^k belong to, to the outest one. □

5.8.4 Proof of Proposition 3

Proof. Thanks to Lemma 1, proving Proposition 3 corresponds to prove that in a generic time step t, where repairs are done fulfilling the condition (5.2) and condition (5.3), any selection of nodes P_t^k that fulfills the condition (5.1) can be perfectly matched with a selection P_{t-1}^k that in turn fulfills the condition (5.1).

Thanks to Lemma 3, P_{t-1}^k can be found matching one by one the nodes in P_t^k proceeding from the nodes with the lowest repair degree to the nodes with the highest one. □

5.9 Computation of failure probability

Let us consider a Hierarchical (k,h)-code and assume that l losses occurred in this code, where $0 \leq l \leq (k+h)$. We first define the probability $P(k'|l)$, which is the probability

that, given that l losses occurred, k' is the maximum number of alive fragments in the code, which fulfills the condition (5.1). Note that the definition implies that $P(k'|l)$ exists only for $0 \leq k' \leq k$. Given these probabilities, computing the failure probabilities is straightforward:

$$P(\textit{failure}|l) = 1 - P(k' = k|l)$$

To compute the failure probability we proceed as follows:

1. We compute the probabilities $P_0(k'|l)$ for the Hierarchical (k_0, h_0)-code, represented by the level 0 (the innermost) in the hierarchy as explained in section 5.9.1.
2. We compute the probabilities $P_s(k'|l)$ for the Hierarchical (d_s, H_s)-code, represented by the generic level s, using the probabilities $P_{s-1}(k'|l)$ computed for the hierarchical (d_{s-1}, H_{s-1})-code, represented by the level $s-1$, as explained in section 5.9.2.

5.9.1 Probabilities for level 0

At the level 0 the probability computation is straightforward:

$$P_0(k'|l) = \begin{cases} 1 & \forall k' = k_0 + h_0 - l, k' < k_0 \\ 1 & \forall k' > k_0 + h_0 - l, k' = k_0 \\ 0 & \textit{otherwise} \end{cases}$$

5.9.2 Probabilities for level s

If we have l losses in a generic hierarchical (d_s, H_s)-code associated with the s-th level of the hierarchy we have many different ways in which these losses can be distributed among the g_s groups $G_{d_{s-1},i}$ this code is made of and the h_s fragments associated with this level s. Let us define the **Loss Configuration** of l losses denoted as $\overrightarrow{LC_l}$ as a vector of $g_s + 1$ elements $\overrightarrow{LC_l} = (l_0, l_1, \ldots, l_{g_s})$, where each element l_i indicates how many losses occur in the group $G_{d_{s-1},i}$ except from l_0, which indicates how many losses occur in among the h_s fragments of the level s. The constraints of $\overrightarrow{LC_l}$ are:

$$\begin{cases} l_0 < h_s \\ l_i < d_{s-1} + H_{s-1}, \quad \forall i = 1, \ldots, g_s \end{cases}$$

We denote as $P(\overrightarrow{LC_l})$ the probability that this configuration take place given that l losses occurred and we compute it as explained in section 5.9.3.

Every of the last g_s values in the configuration (all of them except l_0) indicates a number of losses l_i in a given subgroup and thus denotes a set of probabilities $P_{s-1}(k'|l_i)$, with $0 \leq k' \leq d_{s-1}$, which express the probability that k' is the maximum number of alive fragments from the subgroup i that fulfill the condition (5.1) for the level $(s-1)$.

If, for each of this subgroup we select a specific value k'_i, we define a **Fragment Configuration** of $K' = \sum_{i=1}^{g_s} k_i$ alive fragments denoted as $FC_{K'}$, whose probability is denoted as $P(FC_{K'})$ and given by:

$$P(FC_{K'}) = \prod_{i=1}^{g_s} P_{s-1}(k'_i|l_i)$$

The probability $P(FC_{K'})$ represents one of the components of the probability that, given the configuration analyzed $\overrightarrow{LC_l}$, K' is maximum number of alive fragments taken from the subgroups such that the condition (5.1) is fulfilled for the level s. To obtain the maximum number of alive fragments from the *whole group* that fulfill the condition (5.1), K' must be augmented with $h_s - l_0$ alive fragments of the level s, with the constraint: $K' + h_s - l_0 \leq d_s$.

Putting the pieces together we can finally define the probability $P_s(k'|l)$:

$$P_s(k'|l) = \sum_{\forall \overrightarrow{LC_l}} P(\overrightarrow{LC_l}) f(k', \overrightarrow{LC_l})$$

where the auxiliary function $f_s(k', \overrightarrow{LC_l})$ is defined as follows

$$f_s(k', \overrightarrow{LC_l}) = \begin{cases} 1 & k' < h_s - l_0 \\ \prod_{\forall FC_{k'-(h_s-l_0)}} P(FC_{k'-(h_s-l_0)}) & k' < k, k' \geq h_s - l_0 \\ \sum_{j=0}^{k_s-l_0} \prod_{\forall FC_{k'-j}} P(FC_{k'-j}) & k' = k, k' \geq h_s - l_0 \end{cases}$$

5.9.3 Loss Configuration probability

We can map the loss configuration problem to the following balls and bin problem. Consider a set of $g_s + 1$ colors, for each color i there are n_i balls, which are inserted in a bin. We extract form the bin a total of l balls, which will form a color configuration described by a vector of $g_s + 1$ elements, where each element l_i indicates how many balls of color i have been extracted. Considering the original loss configuration problem, our objective is to compute the probability of a given color extraction, where $n_0 = h_s$ and $n_i = k_{s-1} + h_{s-1}$ for $1 \geq i \geq g$. This probability can be computed dividing the number of possible configurations corresponding to the extraction by the total number of possible configurations, which gives:

$$P(\overrightarrow{LC_l}) = \frac{\prod_{i=0}^{g_s} \binom{n_i}{l_i}}{\binom{\sum_{i=0}^{g_s} n_i}{l}}$$

where $\overrightarrow{LC_l} = (l_0, l_1, \ldots, l_g)$ and :

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

CHAPTER 6

Adaptive Proactive Repair Policy

6.1 Introduction

In the previous chapters we focused our attention on redundancy schemes. In particular, we analyzed the efficiency of such redundancy schemes with respect to the resources they need: storage, communication, and computation. The key observation of all our work is that, in the economy of peer-to-peer file backup systems, the most critical element is the scarcity of communication bandwidth. From this perspective, the redundancy schemes we studied and proposed are designed to reduce the communication bandwidth utilization. This objective is achieved essentially by reducing the amount of data that need to be transferred upon repairs. In this chapter we try to pursue communication bandwidth savings from a different point of view. In particular, we investigate the impact of repair policies on the communication bandwidth.

While a redundancy scheme defines *how* to produce and maintain data redundancy, a repair policy decides *when* to perform the maintenance. In chapter 3 we explained that the maintenance process must essentially repair data at a rate that *in average* is equal to the rate of permanent losses. However, the exact instant in which repairs are done to guarantee such average rate can vary a lot. While the choice of these instants has in theory almost no impact on the durability of data, they can have a strong impact on the communication bandwidth requirements. In particular a bad choice may cause a bursty use of communication bandwidth, which can be unsustainable for the system. The reason for this bad impact is that bandwidth is a resource that must be used immediately and cannot be put aside for future use, in other words if the system does not use communication bandwidth for a given period of time, it is just wasted. If the repair process operates in bursts, the spikes in network bandwidth may not be sustainable by the peers and the success of the repair process may be delayed or compromised. Therefore, one should try to smooth as much as possible the repair rate by anticipating some of the repair work in order to prevent bursts to occur.

To give a simple example, let us assume a system where an erasure (k,h)-code is used and assume that every hour 10 parity blocks are permanently lost. Then, the maintenance process must repair 10 parity blocks per hour to keep up with the losses. If $h >> 10$, it does not

matter if all the 10 blocks are repaired together every hour or if one block is repaired every 6 minutes. However, if the bandwidth resources are limited, the system could be unable to perform a burst of 10 repairs at once and the solution of spreading them over time could be optimal.

In this chapter we introduce a new metric to evaluate the effectiveness of a repair policy, which is the smoothness of the repair rate and we propose a repair scheme that strives to maximize it.

Existing approaches for the repair of lost parity blocks are either **reactive** or **proactive**. Reactive schemes are able to follow changes in failure behavior of peers and provide availability, however they tend to perform more repairs than what is strictly needed and make a bursty use of resources.

Proactive schemes use a constant repair rate and are able to smooth the resource consumption for repairs. However, to provide durability, proactive schemes need an *a priori* knowledge of failure behavior. In case of imprecise or wrong knowledge, durability may be compromised.

We argue that proactive and reactive schemes represent two specific cases of a more general approach that tunes its reactivity with respect to the expected stability of the peers. In limit cases, it may result in a purely reactive scheme, if the peer behavior does not follow any predictable pattern, or a fixed repair rate, if the peer failure behavior remains constant.

We want to draw the attention to one important issue: Any scheme that does a repair to replace a parity block stored on a node that became temporarily unavailable may do *wasted work*. Indeed, if the newly created parity block was stored on a peer that later permanently leaves, that parity block may disappear before the system needs it. In this case, that parity block would be completely useless for improving either durability or availability.

Our main contribution in this chapter is a framework based on an ongoing estimation of the peer failure behavior. The rate R, at which repairs are performed, is periodically updated according to the changes in the statistical properties of failures. This framework is able to provide at the same time durability, adaptiveness, and a smooth use of the resources.

The design of this framework requires the solution of an adaptive control problem, presented in section 6.3, which is based on a period estimation of the peer behavior. In section 6.4 we discuss the impact of the estimation time, while in section 6.5 we propose a hybrid reactive-proactive scheme, which improves the availability guarantees of the system. The proposed system is first validated in section 6.6 and then evaluated using experiments as described in section 6.7.

6.2 Background

Providing availability or durability is a key issue to be addressed by any distributed storage system. The classical solution to replace a missing fragment is a purely **reactive scheme**, also known in literature as **eager repair scheme**. These systems [38, 50, 85] make no distinction between transient and permanent failures and the parity blocks stored on peers that become

again available after a temporary disconnection are not reintegrated in the system. This approach is extremely simple and it seems to be very effective, but it does not address at all the costs of the maintenance. The load due to maintenance may saturate in certain conditions the network or disk I/O bandwidth available and compromise data durability.

We know from traces of peer availability [28] that temporary disconnections are typically much more frequent than permanent ones. Therefore, reintegrating parity blocks should significantly reduce the number of repairs needed.

There exist a number of reactive schemes that use reintegration. These systems are more complex since they need to track the disconnected peers and react selectively to disconnection events, by means of a threshold. *Carbonite* [34] uses a repair threshold TH_L that corresponds to the minimal level of redundancy needed to face transient failures and provide availability. TH_L is considered as a lower-bound: any time the number of available parity blocks runs below TH_L, a single repair is performed. This threshold-based reactive scheme is the cheapest one in terms of resources consumed, since only the repairs strictly needed are performed. *Total Recall* [29] uses a lower-bound threshold TH_L as well, but fixes also another threshold TH_H, which is the amount of redundancy that is initially inserted and that is restored when the system runs below TH_L. This means that the system triggers multiple repairs at once to bring the number of replicas back to the initial number TH_H. This approach represents a first step towards a proactive approach, where part of the work is done in advance with respect to the real needs.

Datta and Aberer [41] are the first to address the evolution over time and the steady state characteristics of a storage system. They propose the use of a threshold TH_L but adopt a proactive approach called **random lazy repair** strategy, in which the number of repairs done increases as the system gets closer to the threshold TH_L. The main intuition behind random lazy repair is that if one waits until the threshold is reached before repairs are made, the occurrence of correlated failures may put in danger the durability of the objects. Besides, this may result in a very bursty use of the resources needed.

To our knowledge, *Tempo* [90] is the only one that tries to smooth the bandwidth used for repair. *Tempo* argues that reactive systems tend to perform repairs in bursts, alternating between periods of intensive bandwidth consumption and periods of inactivity. The spikes produced by such a behavior represent both an inefficient use of the resources and a danger for the object durability. The idea in *Tempo* is to have a constant repair rate that is not correlated to the *instantaneous condition* of the system. This rate is fixed by the system administrator in terms of a bandwidth budget per node, and if properly chosen, is able to assure durability. The main weakness of this method is that it is close to impossible to choose *a priori* the right repair rate. Even if the bandwidth is used as smoothly as possible, there is neither guarantee on the object durability nor on the optimality of the resource consumption, which may be a lot higher than what is strictly needed.

The present work, starting from the considerations made by *Tempo*, tries to develop a system that strives to meet three different objectives: provide data availability, consume an amount of resources comparable with the one consumed by reactive schemes, and *maximize the smoothness of the repair bandwidth needed*.

6.3 An adaptive control problem

A generic non-adaptive scheme uses a repair rate R that is constant in time to match a target number of available parity blocks.

In this work, we aim to build an *adaptive control scheme* that is able to adapt the repair rate to the changes of the system behavior.

We depict our adaptive control scheme in Fig. 6.1. The scheme is composed by three components:

- **The system** represents the evolution of the status of the peers storing a particular object, in terms of temporary disconnections, permanent disconnections, reconnections, and repairs. It is characterized by a system model explained in section 6.3.1, governed by three parameters: the disconnection rate μ, the reconnection rate λ, and the death probability P_{death}. The input of the system is the repair rate, which is a time-dependent signal $R(t)$, determined by the controller. The output of the system is the number of peers online $n(t)$ and an additional information, called **transition**, which signals the occurrence of a repair or of a reconnection.
- **The estimator** estimates the parameters that characterize the system, by observing the output of the system. The output of the estimator are the estimate $\hat{\mu}$ of the disconnection rate and estimate $\hat{P}_{death}$ of the death probability, which are obtained as explained in section 6.3.2
- **The controller** receives as input the estimates ($\hat{\mu}$ and $\hat{P}_{death}$) from the estimator and the target number n' of online peers set by the system administrator. The output of the controller is the repair rate $R(t)$ needed to match the target n', which is computed as explained in section 6.3.3.

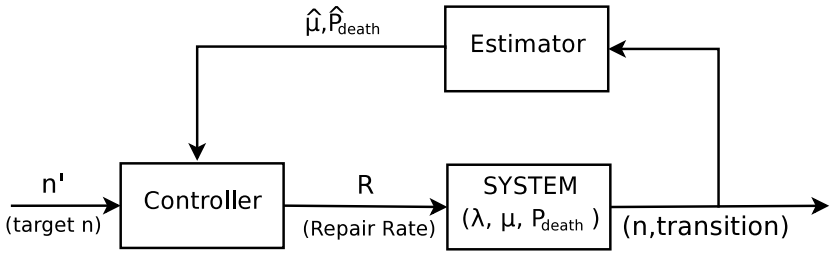


Figure 6.1: The adaptive control scheme.

The operations of estimation and control are performed periodically at a rate $1/\Delta T$, where ΔT is the observation period used by the estimator, and the update period, i.e. the interval between two updates of $R(t)$.

6.3.1 The System Model

The design of the estimator requires the definition of a mathematical model of the system that captures the behavior of peers.

We have already introduced such model in chapter 5, as a Markov model depicted as the state machine of Fig. 5.7. While in the previous chapters we assumed exponentially distributed session times and disconnection times, now we can relax this assumption, saying that those times follow a generic distribution. The model we obtain is a Continuous-Time Semi-Markov chain as depicted in Fig. 6.2. To ease the notation, we also introduce some auxiliary parameters, in particular we define the following transition rates:

- μ: **Single peer disconnection rate**, which corresponds to the inverse of the average session time. $\mu = 1/T_{on}$.
- λ: **Single peer reconnection rate**, which corresponds to the inverse of the average disconnection time $\lambda = 1/T_{off}$.

With this new notation the model can be depicted as in Fig. 6.2

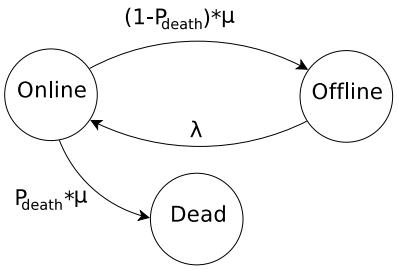


Figure 6.2: The Continuous-Time Semi-Markov chain of a peer life-cycle.

The behavior that describes the life-cycle of a single peer must be translated into a more complex model that takes into account the participating peers all together and the repair rate R.

Given the above assumptions, we can use a network of two $G/G/\infty$ queues depicted in Fig. 6.3 to represent the system behavior: $G/G/\infty$ queues represent pure delay elements where the delay, which corresponds to the service time, fits a generic distribution. In our case, Q_1 represents the peers in the *online* state and Q_2 represents the peers in the *offline* state.

We assume that the number of peers is sufficiently large so that every parity block can be stored on a different peer. With this assumption, the terms peer and parity block are equivalent and this model represents also the availability of the parity blocks in the system.

The customers of the first queue Q_1 represent the *number of available parity blocks* $n(t)$. Its arrival process, whose rate is denoted with γ_1, is given by the parity blocks that are newly introduced at rate R and the process of parity blocks that are becoming again online after a period of unavailability. The time spent in Q_1 is determined by the service rate μ. The departure process from Q_1 represents the parity blocks becoming unavailable and its rate is equal to the arrival rate. The customers in the second queue Q_2 represent the *number of temporarily unavailable parity blocks* $m(t)$ at time t. Its arrival process, whose rate is denoted with γ_2, is given by the parity blocks that have become unavailable and did not abandon permanently the system, which is: $\gamma_2 = (1 - P_{death})\gamma_1$. The time spent in Q_2 is determined by the service rate λ. Finally, the departure process from Q_2 represents the parity blocks becoming available again.

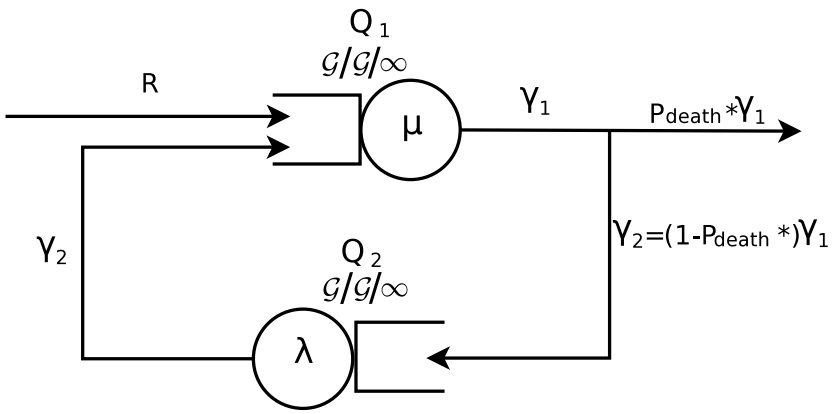


Figure 6.3: Queuing system representing the overall system behavior.

We want to make a clarification concerning the two parameters μ and λ, which are defined, in the Continuous-Time Semi-Markov chain of Fig. 6.2, as the single peer disconnection and reconnection rates. In the rest of the chapter, we will refer to them simply as **disconnection rate** and **reconnection rate**. In the queuing system of Fig. 6.3, μ and λ represent the service rates of the two queues and must not be confused with the global departure rates, which are referred to by γ_1 and γ_2.

To solve this network of queues, we write its balance equations:

$$\begin{array}{rcl} \gamma_1 &=& R+\gamma_2 \\ \gamma_2 &=& (1-P_{death})\gamma_1 \end{array} \Rightarrow \left\{ \begin{array}{rcl} \gamma_1 &=& \frac{1}{P_{death}} R \\ \gamma_2 &=& \frac{1-P_{death}}{P_{death}} R \end{array} \right. \tag{6.1}$$

Using Little's law [99] we can derive the average number of customers in each queue:

$$\overline{n} = \gamma_1/\mu = \frac{R}{\mu P_{death}} \qquad \overline{m} = \gamma_2/\lambda = \frac{(1-P_{death})R}{\lambda P_{death}} \tag{6.2}$$

In case the service times and the arrival times are exponentially distributed, Q_1 and Q_2 become $M/M/\infty$ queues and we can write the analytical expression [99] for the probability distribution of the number n of available parity blocks as:

$$f(n) = \frac{\overline{n}^n}{n!} e^{-\overline{n}} \sim \mathcal{N}(\overline{n}, \overline{n}) \tag{6.3}$$

Note that eq. 6.3 can be approximated by the Normal distribution with mean and a variance being $\overline{n}$. This particular model will be used exclusively in the experiments with synthetic data to validate the system and to get some insights into its functioning.

6.3.2 The Estimator

The estimator is an object that, collecting statistical information on the signal $n(t)$ and on the input flow in the first queue, is able to estimate the parameters μ and P_{death} needed by the controller. We obtain:

- The average number of available parity blocks $\hat{n}$:

$$\hat{n} = \frac{\sum_i n_i t_i}{\Delta T} \tag{6.4}$$

where t_i is the time spent by the system in the state n_i, which implies $\Delta T = \sum_i t_i$.

- The disconnection rate $\hat{\mu}$ using the relation:

$$\hat{\gamma_1} = \frac{\#Disconnections}{\Delta T}$$

and Little's law as in eq. 6.2:

$$\hat{\mu} = \frac{\hat{\gamma_1}}{\hat{n}} \tag{6.5}$$

- The death probability $\hat{P}_{death}$ can be computed in two equivalent ways. The first one is:

$$\hat{P}_{death} = 1 - \frac{\#Reconnections}{\#Disconnections}$$

while the second, which is the one used in our implementation, relies on the first balance equation in eq. 6.1 and is:

$$\hat{P}_{death} = \frac{R}{\hat{\gamma_1}} \tag{6.6}$$

6.3.3 The Controller

The controller receives the estimations $\hat{\mu}$ and $\hat{P}_{death}$ and the target number of available parity blocks n' and uses eq. 6.5 and eq. 6.6 to compute the repair rate R as:

$$R = \hat{\mu}\hat{P}_{death}n' \tag{6.7}$$

Table 6.1 summarizes the symbols used.

6.4 Impact of Estimation Time

The estimation time ΔT, which corresponds also to the time between two updates of the repair rate R, is the most crucial parameter of our model. In this section we discuss the tuning of ΔT and explain its implications on the performance of the system.

6.4.1 Impact on Bandwidth Usage

Proactive schemes work well in static environments that exhibit constant statistical properties, i.e. properties that once estimated never change. In such a case, the ideal choice would

symbol	description
μ	disconnection rate
λ	reconnection rate
P_{death}	death probability
R	repair rate
Q_1	queue of available parity blocks
Q_2	queue of temporarily unavailable parity blocks
γ_i	departure/arrival rate for queue Q_i
n	number of available parity blocks
m	number of temporarily unavailable parity blocks
n'	target number of available parity blocks
ΔT	estimation or update period

Table 6.1: Table of symbols used in the adaptive proactive repair policy .

be to perform a preliminary *offline* estimation and then select ΔT to be infinite. Any different choice would make the controller follow short-term fluctuations, which will result in unnecessary work and an uneven use of the bandwidth resources.

Reactive schemes follow instantaneously any fluctuation of the system in the belief that no properties describing the long-term behavior of the system can be found. This, in principle, corresponds to setting $\Delta T = 0$, in which case the controller promptly reacts to any change. An example of such a condition is when we see correlated failures of many nodes where most of the available parity blocks will suddenly disappear.

Our assumption is that, while real systems may lack long-term statistical properties, they may have short-term properties that can be still exploited to make a smoother use of the bandwidth while providing data durability.

If we consider a system in which the model parameters change continuously at a given rate, our challenge is to choose the maximum ΔT that divides the time in segments in which the system can be approximated as being statistically stable. This ideal choice would use the repair bandwidth in the smoothest possible way. In this case, the residual fluctuations in the repair bandwidth are those strictly necessary to provide durability.

If these fluctuations are not supported by the system, it just means that the available resources are not enough for what we are trying to achieve. At this point, reactive schemes appear as the safest and the most conservative choice: they may overshoot the instantaneous repair rate and require, in certain periods, an excessive amount of the repair bandwidth. Reactive systems have been very popular, because, without any complex tuning, they provide availability at the price of a bursty resource consumption. Purely proactive schemes, instead, received much less attention in the literature, because they only work in some theoretical cases.

6.4.2 Robustness of the Estimation

The speed of convergence is a key issue for a statistical estimator, and in our case puts a lower limit on the values of ΔT.

If one tries to push system reactivity too much, it would degenerate to a situation in which estimation does not make sense and a pure reactive scheme is more reliable.

Furthermore, the time needed to estimate the parameters depends on the parameters themselves. This means that in a dynamic environment a fixed choice of ΔT does not make sense. Therefore, in our implementation we do not fix ΔT, but D, the **average number of disconnections observed during an estimation period**: we use the last estimations of $\hat{\mu}$ and $\hat{n}$ to predict the time ΔT_D we expect to wait to observe D disconnections, with ΔT_D defined as:

$$\Delta T_D \triangleq \frac{D}{\hat{\gamma}_1} = \frac{D}{\hat{\mu}\hat{n}} \tag{6.8}$$

6.5 A Hybrid Scheme for Availability

Let us consider the control rule in eq. 6.7 using the real parameters and under the hypothesis that the average number of available parity blocks is exactly n':

$$R = \hat{\mu}\hat{P}_{death}n' \Rightarrow R = P_{death}(\mu\overline{n}) = P_{death}\gamma_1 \tag{6.9}$$

Eq. 6.9 says that the objective of the controller is to *make the repair rate equal to the rate of permanent failures*, which corresponds to an **oracle system** that is able to tell apart the permanent departures from the transient ones. This ability, however, provides only durability, but it cannot give any guarantee on the availability. Indeed, there might be periods in which a lot of peers are temporarily disconnected and some objects may temporarily not have enough available parity blocks to be reconstructed, while their existence in the system in the long run is not jeopardized.

If we assume, as in our case, that we need at least k parity blocks to replace any lost fragment, *we need availability to assure durability*. Therefore, even an oracle may run into situations in which it cannot respond to a permanent failure because there are not enough parity blocks left.

In practice this means that when the number of available parity blocks is below the availability level (i.e. less than k parity blocks) we can only infer, according to the current estimation of P_{death}, that the missing parity blocks are on machines that will reconnect later.

If we want to guarantee that a new parity block can be generated whenever required by our controller, we need to make sure that there are always at least k parity blocks. For this reason, we propose a **hybrid scheme** that switches to a purely reactive policy any time the number of available parity blocks hits a lower threshold TH_{pro}. During these *reactive* periods, the number of available parity blocks will not decrease as long as the system can keep up producing a new parity block each time a node disconnects. However, the price to pay are repair spikes. If TH_{pro} is properly chosen, these spikes are strictly needed and the hybrid scheme maintains in this case the best trade-off.

6.6 Validation

In the following we will validate the different pieces of our system, starting with the system model.

To validate the system, we designed and implemented an event-driven simulator that models a simplified version of a distributed storage system on a set of peers whose behavior is described by availability traces it receives as input.

For these experiments, we created synthetic traces of the peer behavior using exponentially distributed disconnection and reconnection times. This assumption is very useful to validate the system. Indeed, if the model is correct we will obtain a distribution of the number of available parity blocks matching the expected theoretical one in eq. 6.3. Note, however, that, since the estimation is independent of the distribution used, a choice of a different distribution would not affect the results.

6.6.1 System Model Validation

To validate the system model, we observe its behavior when the parameters are known. We choose a set of fixed parameters: $\mu = 1$, $P_{death} = 0.5$ and $\lambda = 2$ and the repair rate $R = 100 \cdot \mu \cdot P_{death} = 50$ which should provide, using eq. 6.2, an average number of available parity blocks $\overline{n} = 100$.

Fig. 6.4a shows that the distribution of the number of online parity blocks clearly fits a Normal distribution as we expected. Fig. 6.4b instead depicts the **instantaneous repair rate** denoted as R_{inst}, which reflects roughly the bandwidth consumption due to the maintenance process. This measure is computed as the *inverse of the time elapsed between two consecutive repair events*:

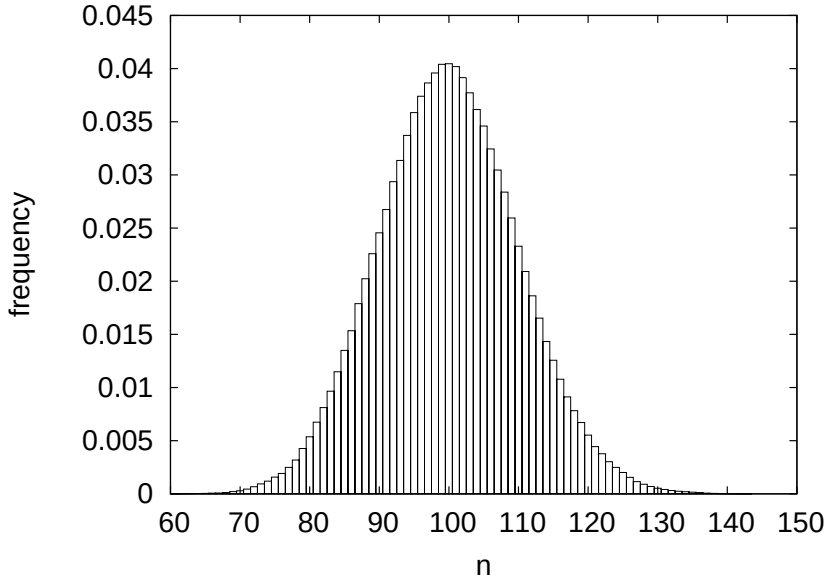
$$R_{inst} = \frac{1}{t_{R_i} - t_{R_{i-1}}}$$

Since repair times are exponentially distributed, the repair events are not equally spaced, producing a high variability in the repair rate.[1]. Such a high variability is an example of what we want to avoid and suggests that exponential distributed repair times are not a good choice, even if the average repair rate R is kept constant. A better solution would be to have constant repair times. To test their impact we repeated the experiments using constant repair times and we obtained roughly the same distribution as in Fig. 6.4a and a perfectly constant instantaneous repair rate $R_{inst} = R$.

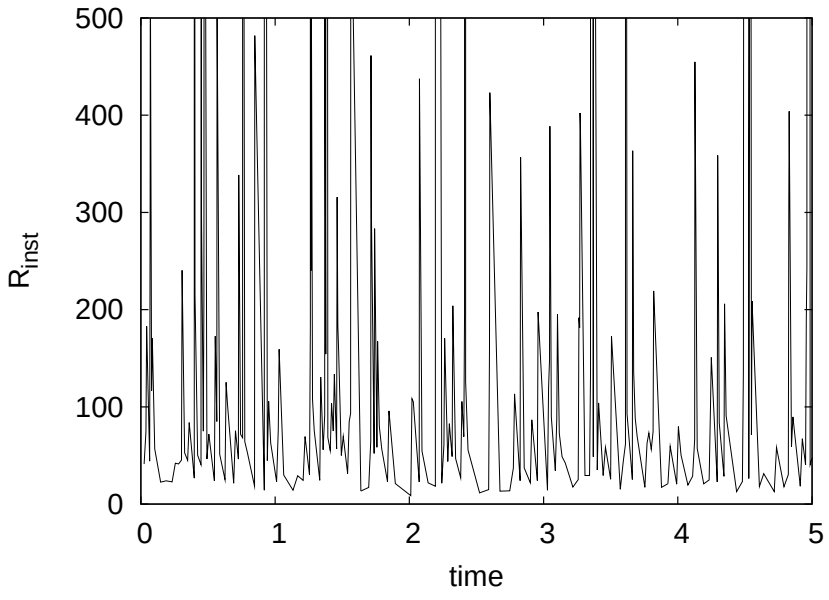
6.6.2 Estimator Validation

In this section we show the efficacy of the estimator. We run several experiments with different *fixed* values of the parameters and we observe the convergence of the estimation. In

[1]For graphical reasons we limited the y-range to 500. Rate spikes did attain values up to 50000 repairs per time unit.



(a) Distribution of the number of available parity blocks.

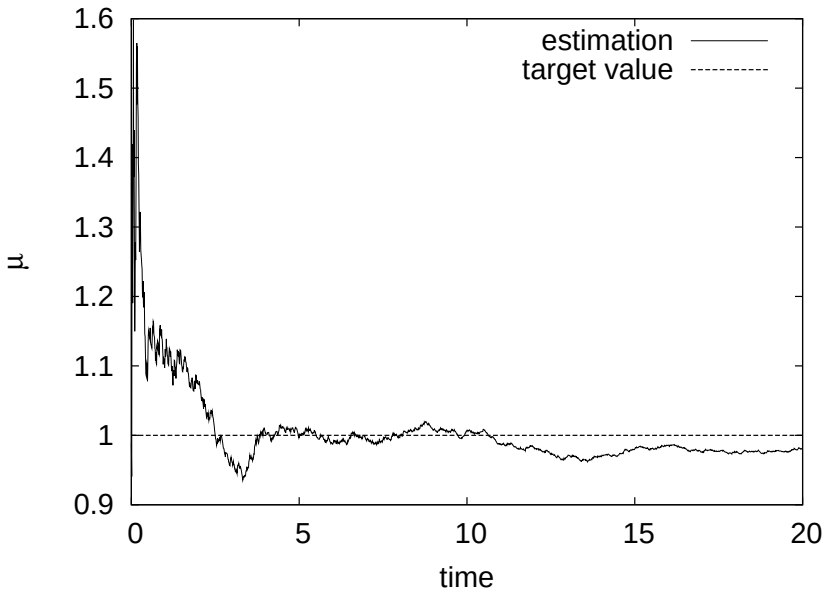


(b) Instantaneous repair rate.

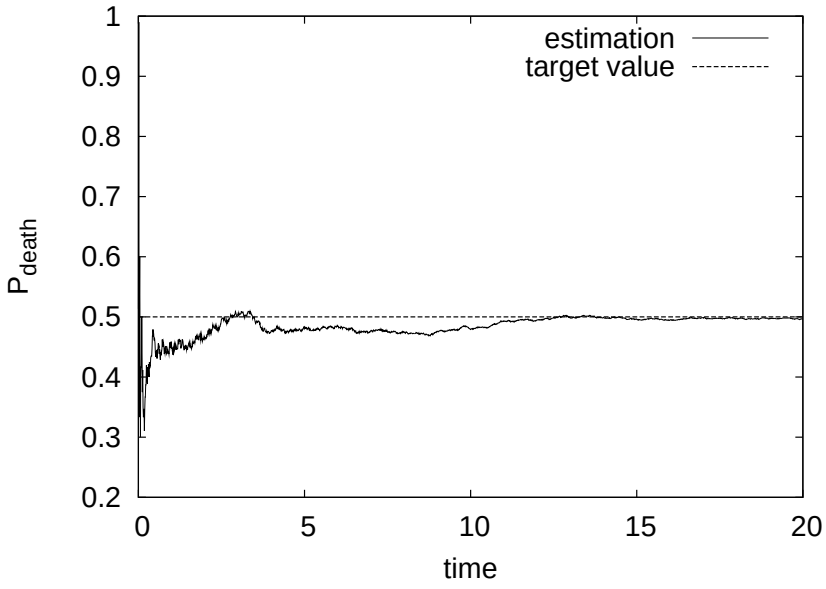
Figure 6.4: Simulation with synthetic data, fixed parameters and exponentially distributed repair times.

Fig. 6.5 we show only the results of a single case, where $\mu = 1$ and $P_{death} = 0.5$. The estimator is able to converge in about 5 time units. This convergence time is roughly proportional to γ_1, i.e. the number of samples (disconnections) observed per time unit, which is in turn proportional to μ.

All the experiments we ran pointed out two different issues that we already discussed in section 6.4.2:



(a) Estimation of the disconnection rate μ.



(b) Estimation of the death probability P_{death}.

Figure 6.5: Estimations with fixed parameters: $\mu = 1$ and $P_{death} = 0.5$.

1. Convergence of the estimation is not immediate. Even with fixed values it takes some time to obtain reasonable estimates. When we select a very small ΔT, we increase the reactivity of the system, but we are not able to infer its statistical properties.

2. The convergence time depends on μ. This leads us to say that in a changing environment we cannot use a constant estimation period; instead ΔT_D should be adapted to the order of magnitude of the parameter μ as we do in eq. 6.8.

6.6.3 Controller Validation

In this section we run experiments with parameters that vary as shown in Fig. 6.6 and with an ideal estimator that knows these parameters. This simulation aims to show that all the considerations made about a system with fixed parameters are still valid in a dynamic environment.

Results are shown in Fig. 6.7 and demonstrate clearly that the controller is able to maintain correctly the number of parity blocks. Moreover, the distribution of the number of available parity blocks in Fig. 6.8 still resembles a Gaussian.

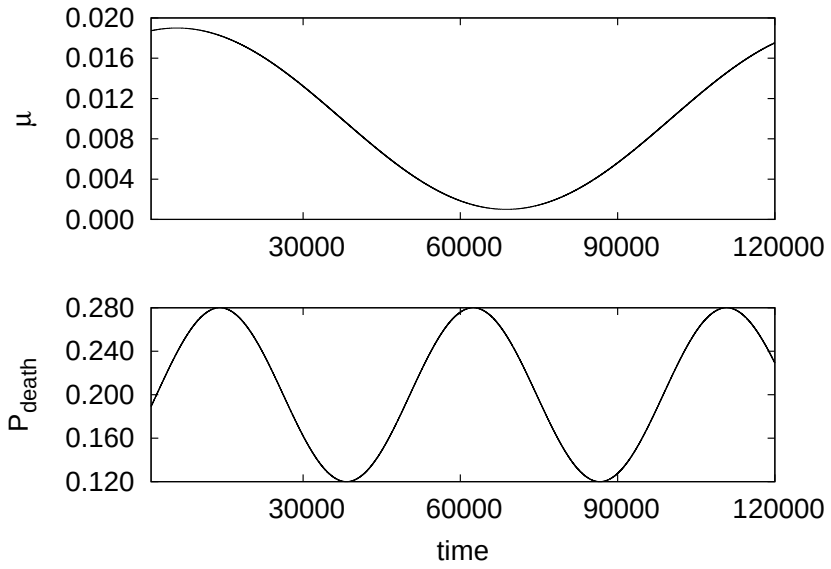


Figure 6.6: Simulation with varying parameters and an ideal estimator. Evolution of the input parameters μ and P_{death}.

6.7 Experiments

We are mainly interested in two aspects, the capacity to assure durability and the smoothness of the instantaneous repair rate. The durability can be easily evaluated looking at the distribution of the number of available parity blocks n.

Assessing the smoothness of the instantaneous repair rate is a bit more complex. Note that the ideal case is not necessarily the one in which the instantaneous repair rate is *constant*, but the one in which its variations are minimal given the variations in the system. This minimum corresponds to the ideal instantaneous repair rate, which is the rate we would select if we were able to know instantaneously the exact system parameters, as we did in section 6.6.3.

Formally speaking, the **ideal instantaneous repair rate** is a continuous signal $R_{ideal}(t)$ and is given by the following relation:

$$R_{ideal}(t) \triangleq \mu(t)P(t)n'$$

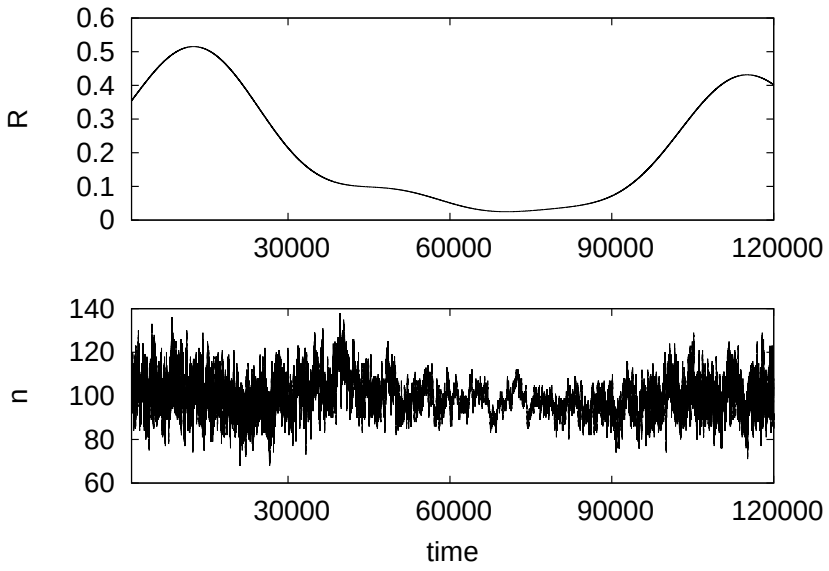


Figure 6.7: Simulation with varying parameters and an ideal estimator. Repair rate R and evolution of the number of available parity blocks.

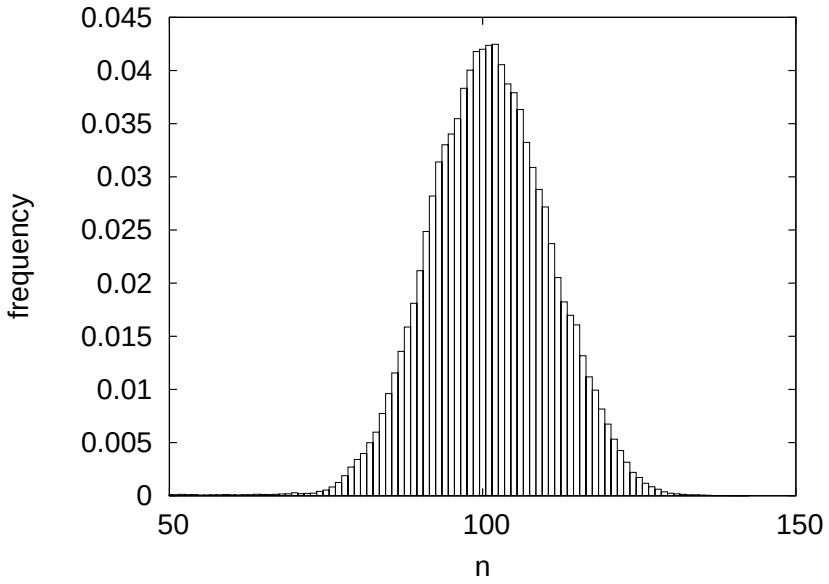


Figure 6.8: Simulation with varying parameters and an ideal estimator. Distribution of the number of available parity blocks.

For every repair event R_i we compute $R_{diff}(t_{R_i})$, which is the instantaneous repair rate we observe and the ideal instantaneous repair rate in that instant:

$$R_{diff}(t_{R_i}) \triangleq R_{inst}(t_{R_i}) - R_{ideal}(t_{R_i}) \tag{6.10}$$

The discrete sequence $R_{diff}(t_{R_i})$ measures how far is the instantaneous repair rate from the ideal one. To characterize this measure we use its standard deviation. The closer $std(R_{diff})$ is to zero, the closer our system is to the ideal case.

6.7.1 Evaluation with Synthetic Data

In these experiments we use synthetic traces, again with the assumption of exponentially distributed disconnection and reconnection times. The objective is to show that our approach obtains a higher smoothness than a reactive scheme and to evaluate the impact of the parameter D on the smoothness.

We choose the same sine waves for μ and P_{death} as in Fig. 6.6 . The duration is 300000 time units and we choose $n' = 100$, which we choose because it can nicely show all the dynamics of our system.

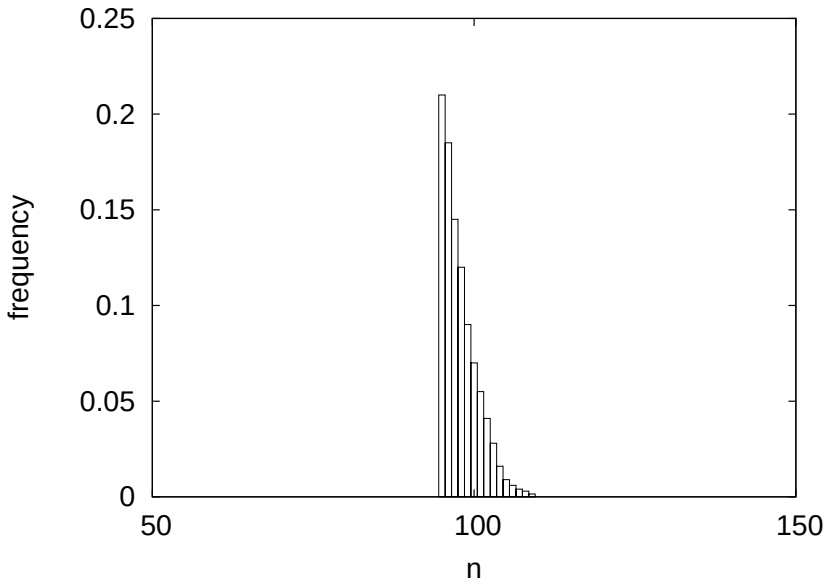
The reactivity is controlled through the parameter D, which in turn influences ΔT_D. Since for too small values of D the estimation is not reliable and for too big values the distribution of the number of available parity blocks degrades too much, we choose for D values between 50 and 2000.

The basis for our comparison is a generic reactive scheme that performs a single repair whenever the number of available parity blocks is below a threshold TH_{reac}. In these experiments, we choose the threshold $TH_{reac} = 95$, which is the value that produces an average number of online parity blocks equal to the one provided by our proactive scheme, namely $n' = 100$. In Fig. 6.9a we plot the distribution of the number of available parity blocks, which represents a very good result since the number of available fragments stays very close to the target number n'. However, the cost of the reactive scheme is a bursty repair activity shown in Fig. 6.9b, where the sequence of repair events is depicted.

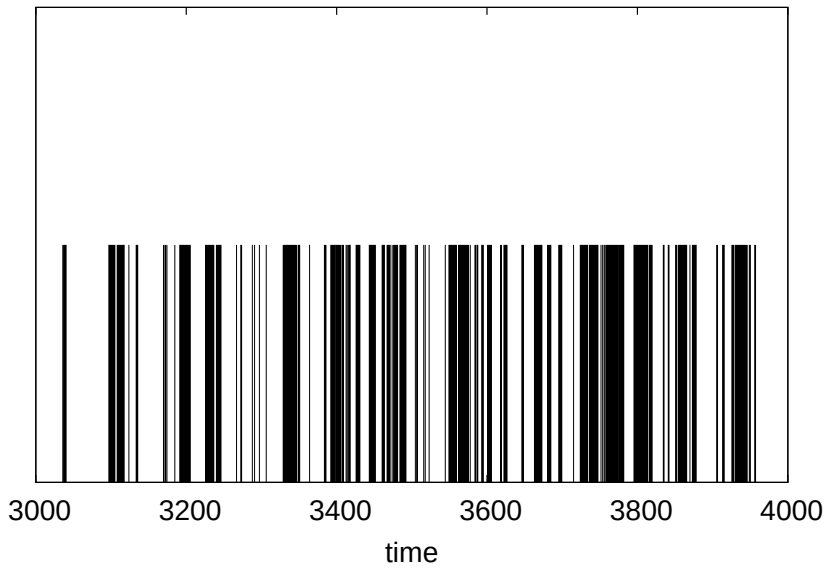
The objective of our scheme is to equally space the repair events, while still assuring a reasonable distribution of the number of available parity blocks. If we compare the distribution of the number available parity fragments obtained with $D = 50$ in Fig. 6.10a and the one obtained with $D = 2000$ in Fig. 6.10b, we can see that increasing D, the distribution gets worse. In particular, the results obtained with $D = 2000$ are clearly not acceptable, since the level of availability is very low. This low availability is due to the fact that D is too big with respect to the parameter dynamics and the estimator is not able to cope with their changes, as shown in Fig. 6.11b, where the instantaneous repair rate $R_{inst}(t_{R_i})$ is compared with the ideal one $R_{ideal}(t)$ in the case of $D = 2000$. However the higher availability obtained for $D = 50$ (Fig. 6.10a) translates in a much lower smoothness as shown by the spiky instantaneous repair rate showed in Fig. 6.11a.

The trends suggested by the cases $D = 50$ and $D = 2000$, are confirmed by the aggregate results for all the values of D shown in Fig. 6.12, where the mean, the 5-percentile and the 95-percentile of the number of available parity blocks are shown in Fig. 6.12a and the standard deviation of R_{diff}, as defined in eq. 6.10, is shown in Fig. 6.12b. Note that at $D = 0$ we associated the results for the reactive scheme. These results give a clear picture of the trade-off in the choice of D, namely that the increased smoothness has a cost in terms of the distribution of the number of available parity blocks.

The poor availability for D being too big motivates the need for a hybrid scheme that puts a lower-bound on the number of available parity blocks.



(a) Distribution of the number of available parity blocks.



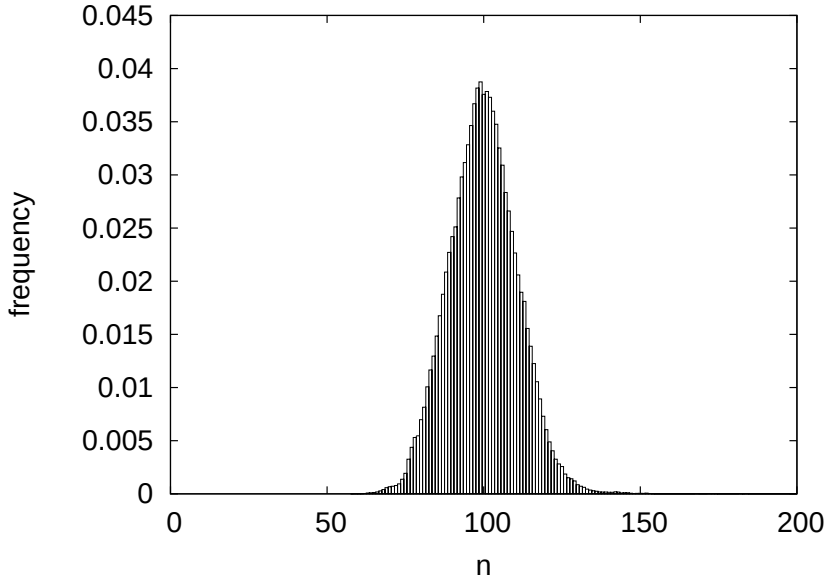
(b) Sequence of repair events.

Figure 6.9: Reactive Scheme: Simulation with synthetic data.

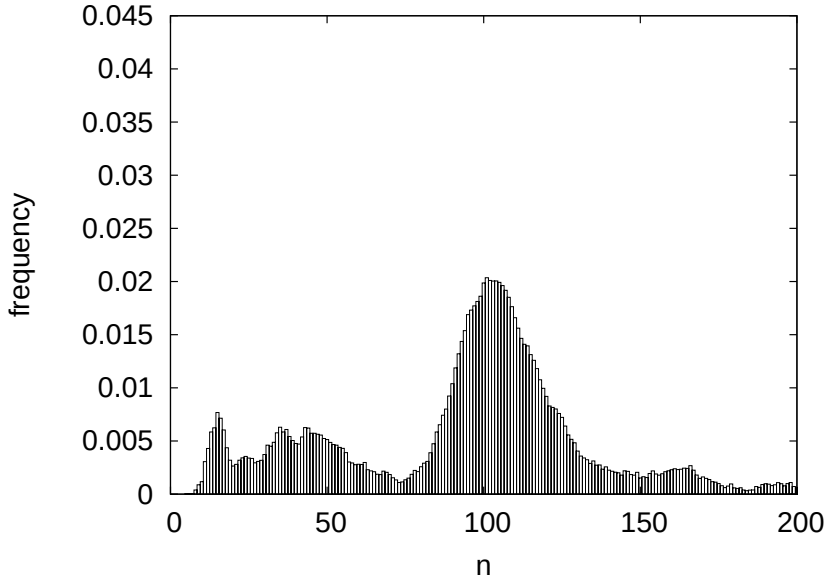
Experiments with the Hybrid Approach

In these experiments we evaluate the hybrid approach presented in section 6.5, and set the threshold $TH_{pro} = 50$.

The aggregate results for all the values of D are shown in Fig. 6.13, where the mean, the 5-percentile and the 95-percentile of the number of available parity blocks are shown in Fig. 6.13a and the standard deviation of R_{diff} is shown in Fig. 6.13b. Comparing these plots with those in Fig. 6.12, the impact of the hybrid approach is evident. On the one hand, the 5-percentile of the number of available parity blocks is raised above 50, i.e. TH_{pro}. On the



(a) D=50.



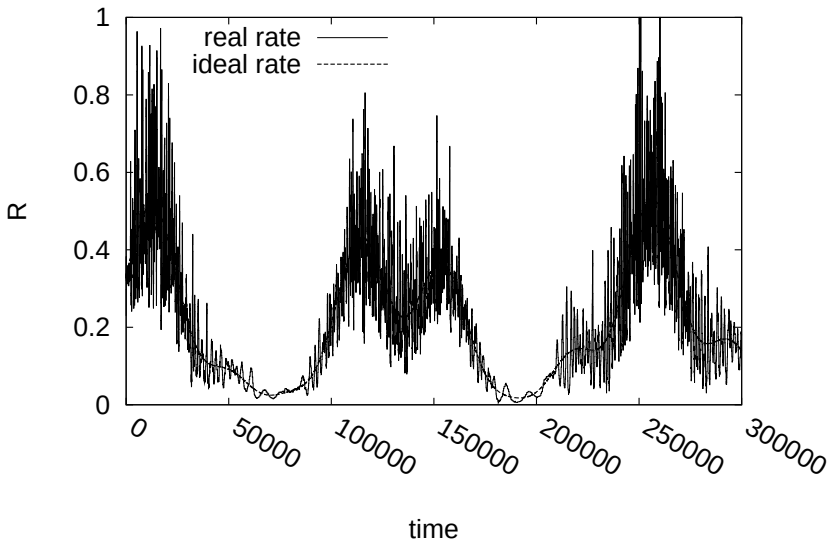
(b) D=2000.

Figure 6.10: Adaptive Proactive Scheme: Simulation with synthetic data. Distribution of the number of available parity blocks for $D = 50$ and $D = 2000$.

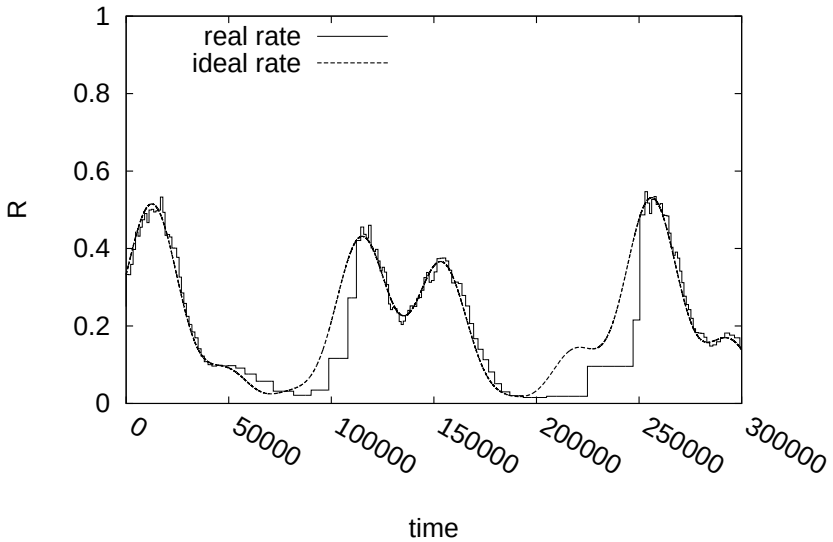
other hand, the standard deviation of R_{diff} increases, which means more fluctuations. Note that this effect is especially evident for bigger values of D, where the system, due to the lower reactivity, more likely runs under the threshold and triggers reactive periods.

6.7.2 Evaluation with real traces

We tested our scheme using real availability traces from PlanetLab and KAD.



(a) D=50.



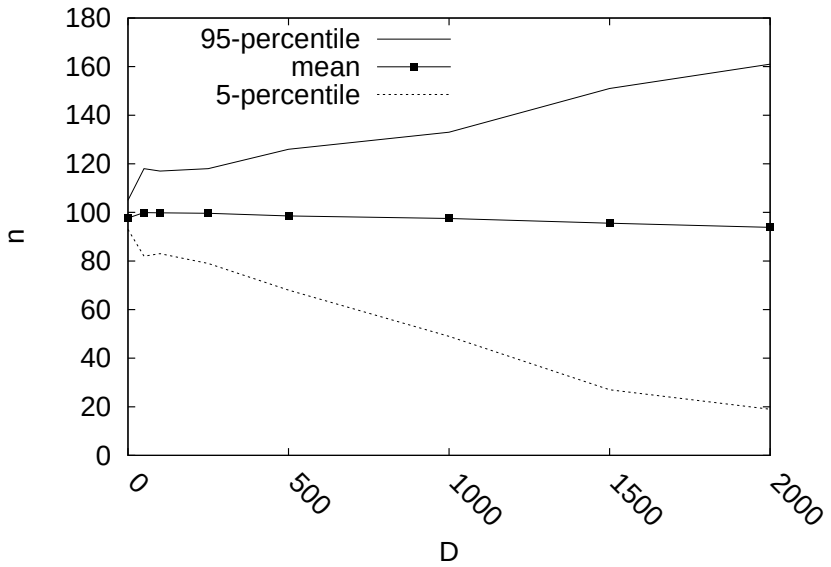
(b) D=2000.

Figure 6.11: Adaptive Proactive Scheme: Simulation with synthetic data. Comparison of R_{inst} and R_{ideal} for $D = 50$ and $D = 2000$.

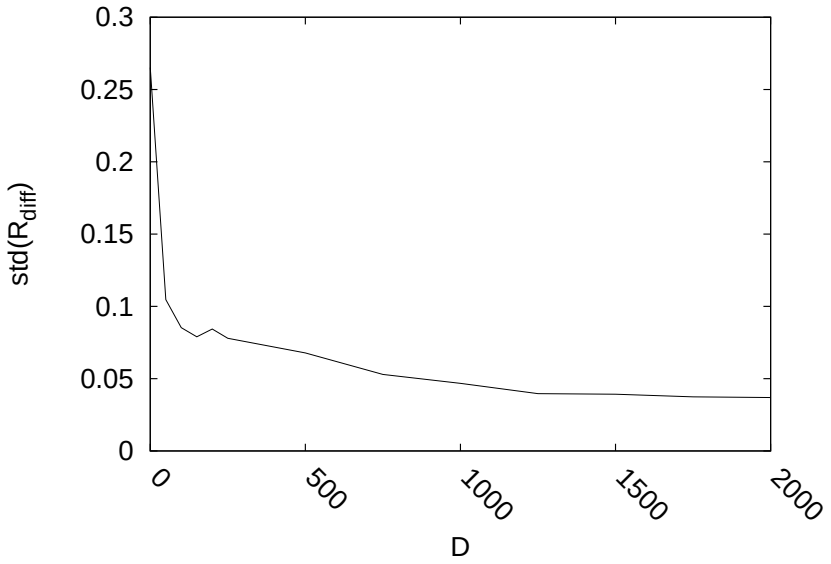
Since for real traces we do not know what are the *real* parameters of the system, we can neither compute R_{ideal} nor R_{diff}. To evaluate the smoothness of the repair rate in this case we use directly the cumulative amount of repairs done over time. This curve, already used in [90], gives us the total amount of work done by the different algorithms and its derivative expresses the instantaneous repair rate at which this work was done.

PlanetLab Traces

As already mentioned in chapter 5, these traces describe the availability status of 669 nodes and were obtained by means of pings sent every 15 minutes between all pairs of the con-



(a) Mean, 5-percentile and 95-percentile of the number of available parity blocks.



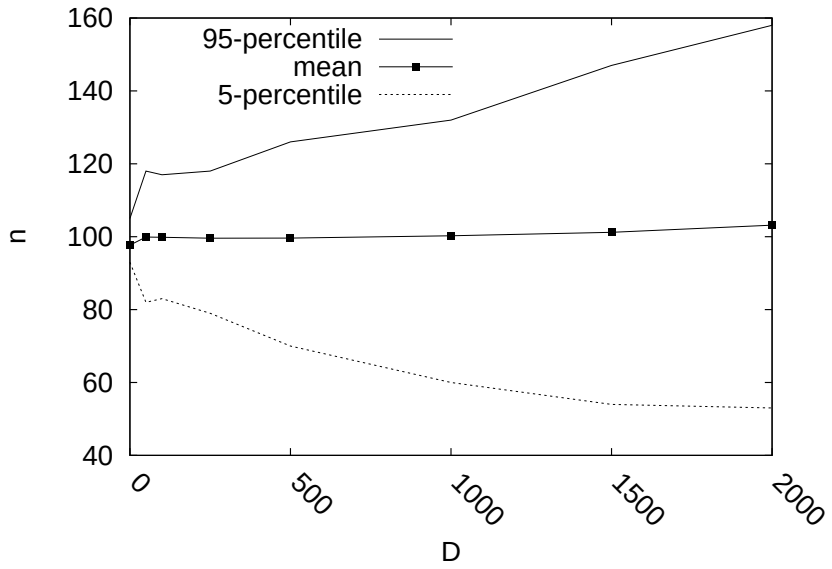
(b) Standard deviation of R_{diff}.

Figure 6.12: Adaptive Proactive Scheme: Simulation with synthetic data. Reactivity vs. parity blocks availability and rate smoothness.

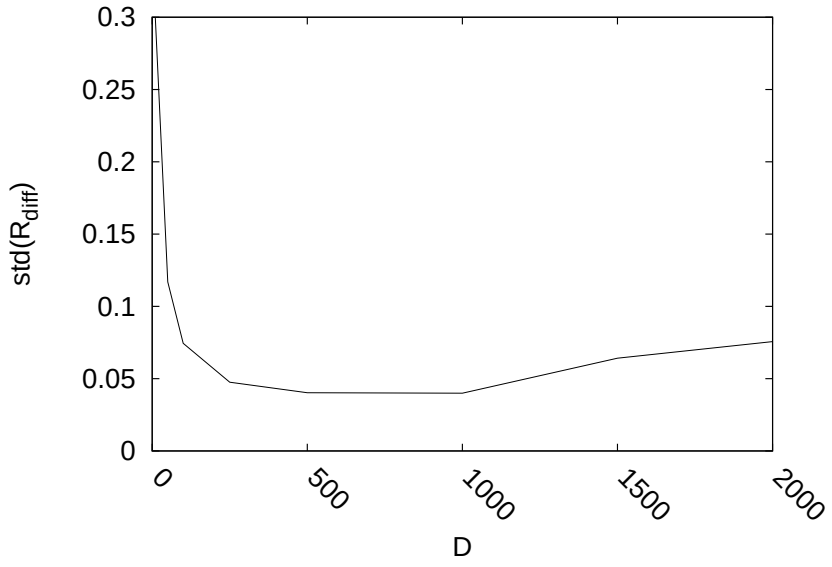
cerned PlanetLab nodes, starting from January 2004 for about 500 days. These traces are publicly available at [56]. We used data from the file `pl-app-cleaned.avt`.

We run and compare three different schemes:

- Our proactive scheme with the hybrid approach, a threshold of $TH_{pro} = 50$ and a target number of available parity blocks of $n' = 100$.



(a) Mean, 5-percentile and 95-percentile of the number of available parity blocks.



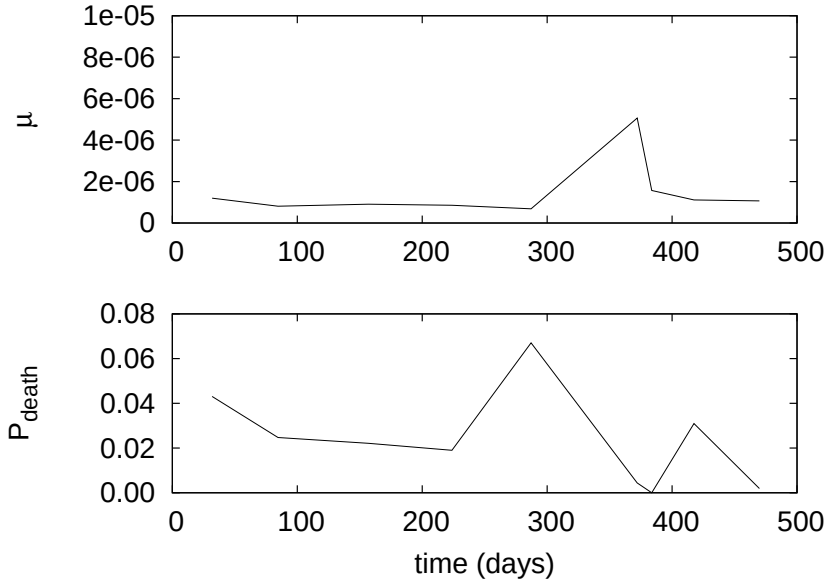
(b) Standard deviation of R_{diff}.

Figure 6.13: Adaptive Hybrid Scheme: Simulation with synthetic data. Reactivity vs. parity blocks availability and rate smoothness.

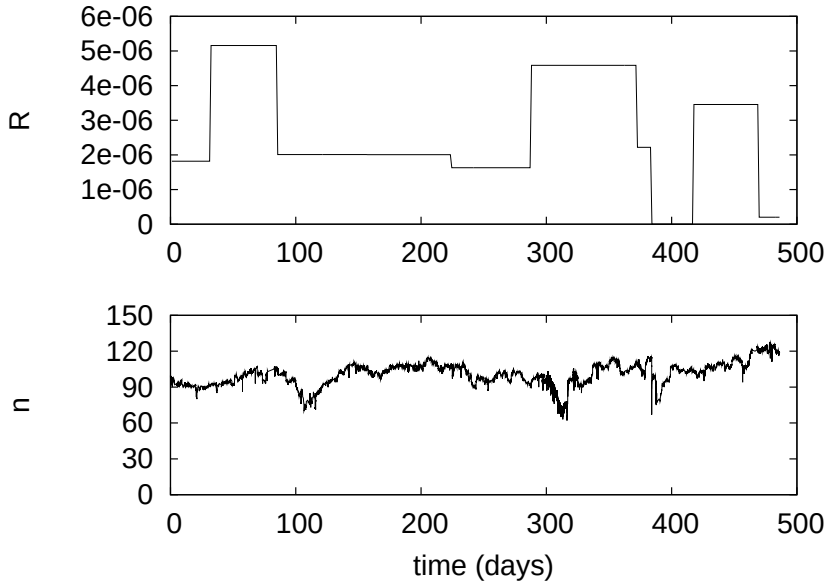
- A reactive scheme with a threshold $TH_{reac} = 80$. We choose experimentally the value of 80 that provides an average number of available parity blocks equal to the one provided by our proactive scheme ($n' = 100$).
- An oracle scheme which also starts with the same initial number of parity blocks as other schemes and represents a system that is able to distinguish transient from permanent failures, triggering a repair only in case of a permanent failure.

To easily compare the schemes we selected two values of D: $D = 500$ and $D = 1000$.

In Fig. 6.14 we show the time evolution of our proactive system for $D = 500$, where the estimations of μ and P_{death} are shown in Fig. 6.14a, while the repair rate selected and the evolution of the available parity blocks obtained are shown in Fig. 6.14b.



(a) Estimation of μ and P_{death}.



(b) Evolution of R and n.

Figure 6.14: Adaptive Hybrid Scheme: Simulation with PlanetLab traces for $D = 500$.

In Fig. 6.15a the distribution of the available parity blocks with the three schemes is shown, while in Fig. 6.15b the cost of the repair process is shown in terms of cumulative number of repairs performed.

The oracle, which is *a scheme that is practically not achievable*, requires the lowest number of repairs, since it creates new parity blocks only when peers permanently fail. Although an oracle system would not perform unnecessary work, its distribution of the number of

available parity blocks is shifted towards lower values of n running into the risk of being below the level required to assure availability.

The reactive scheme and our proactive scheme have a distribution of the number of available parity blocks with the same mean value, namely 100; the difference resides in the fact that in the reactive scheme the threshold mechanism prevents the number of available parity blocks to fall below 80, while in the proactive schemes we tolerate lower values. This situation is still acceptable and is compensated by the advantage of having a much smother repair rate as shown in Fig. 6.15b. While the total number of repairs is comparable, the curves in Fig. 6.15b show that the reactive scheme is more bursty than the proactive schemes. Moreover, as expected, a large D produces a better smoothness.

The differences in the dynamics of the two schemes may be understood looking at day 300, where a lot of transient failures took place. To face such an event, a reactive scheme simply performs a lot of repairs producing a step in the curve and a spike in the resource consumption. The proactive schemes, instead, up to that moment have already done a higher number of repairs, i.e. they have done part of the work in advance, and can absorb the massive disconnection without negative impact on the repair process. At the next parameter update, this higher disconnection rate is taken into account slightly increasing the repair rate. If the disconnection peak was bigger and was not sustainable by the chosen rate, the number of available parity blocks would have fallen below the threshold TH_{pro} triggering the hybrid scheme to initiate a *reactive period* in response of the excessive churn.

KAD Traces

In this case we used the traces obtained crawling a KAD network. These traces ([92], [91]) give the availability of about 6500 peers in the KAD network, sampling their status every 5 minutes for about 5 months.

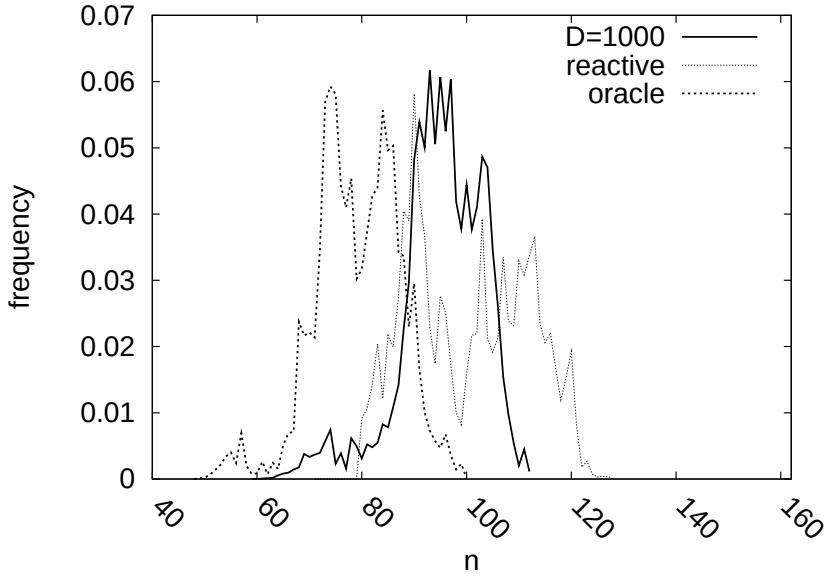
We run and compare the same three different schemes as in the case of PlanetLab. The only difference is that for the reactive scheme we chose a threshold $TH_{reac} = 90$.

In general, proactive schemes tend to have a higher number of repairs. This, as already suggested, is due to the fact that proactive schemes work in *advance* to spread the repairs over time. *This anticipation of repairs is the only way to smooth the rate without compromising the durability*. The price to pay, however, is that part of the parity blocks created in advance may be lost, because of permanent failures, before they are needed [34]. The higher the death probability P_{death} the more pronounced this effect will be.

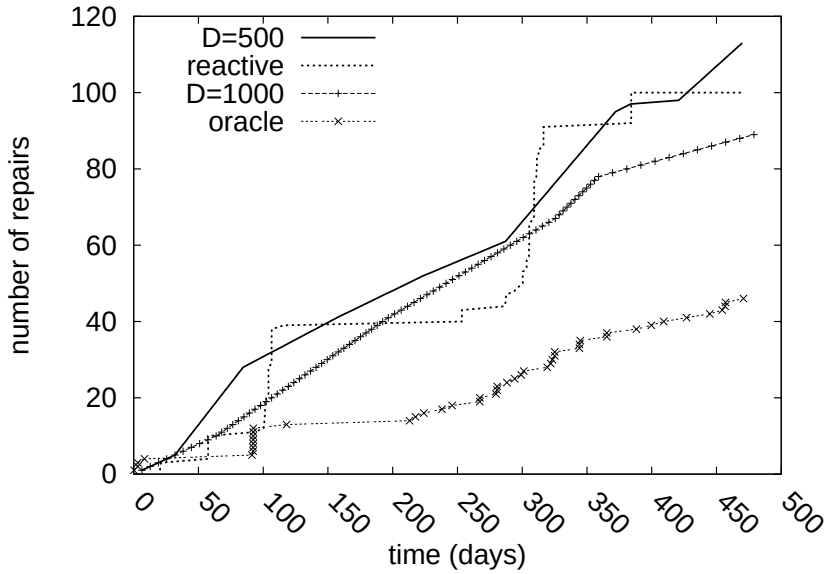
In this experiment we selected two values of D: $D = 1000$ and $D = 3000$.

In Fig. 6.16 we show the time evolution of our proactive system for $D = 1000$, where the estimations of μ and P_{death} are shown in Fig. 6.16a, while the repair rate selected and the evolution of the available parity blocks obtained are shown in Fig. 6.16b.

In Fig. 6.17a the distribution of the available parity blocks with the three schemes is shown. Although the reactive scheme achieve a better result, our proactive seems to be a quite good compromise. The oracle requires the lowest number of repairs, which is due mainly to two reasons: on the one hand the average number of available parity blocks is lower and, as



(a) Distribution of available parity blocks.

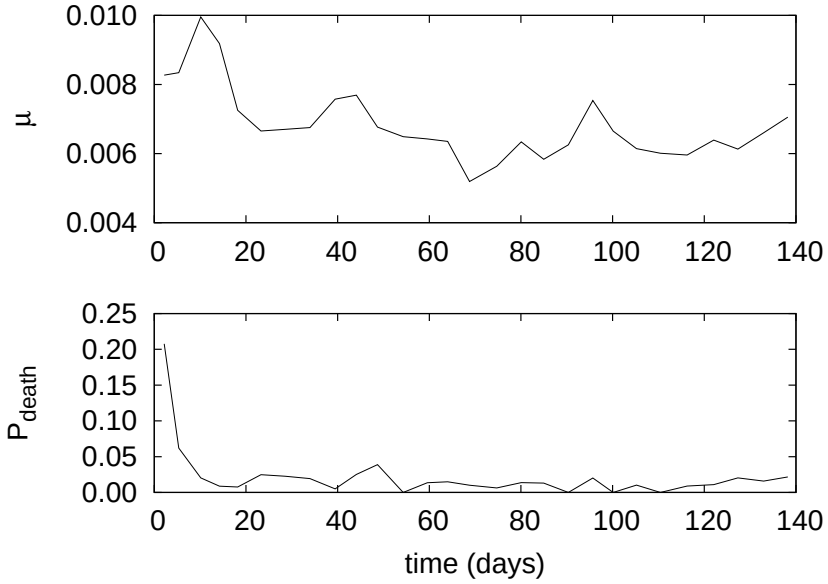


(b) Cumulative number of repairs.

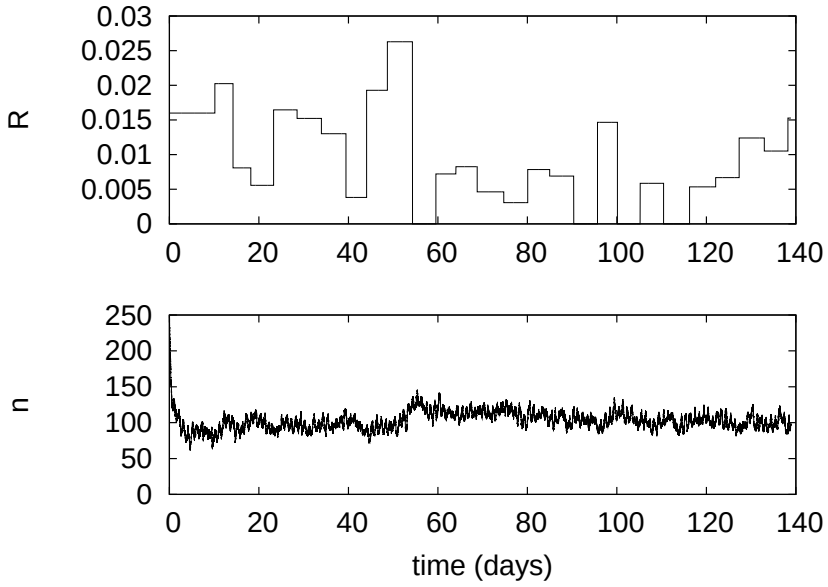
Figure 6.15: Simulations with PlanetLab traces.

already explained in section 6.5, it might even run in periods of unavailability; on the other hand part of the repairs done by proactive and reactive schemes is an additional work that does not always correspond to additional parity blocks, since part of the newly created parity blocks may have disappeared when needed, making their creation useless.

In Fig. 6.17b we show the cumulative repairs. While the oracle, as expected, does a very little number of repairs, the amount of work performed by the proactive and the reactive algorithm is roughly the same. The advantage of the proactive schemes is that they are able to provide a high level of availability for both values of D and to have a smoother rate as



(a) Estimation of μ and P_{death}.

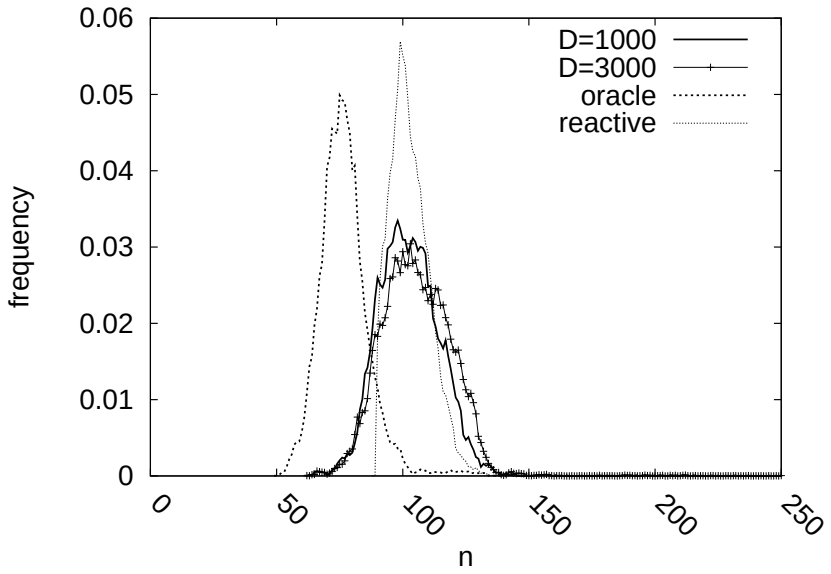


(b) Evolution of R and n.

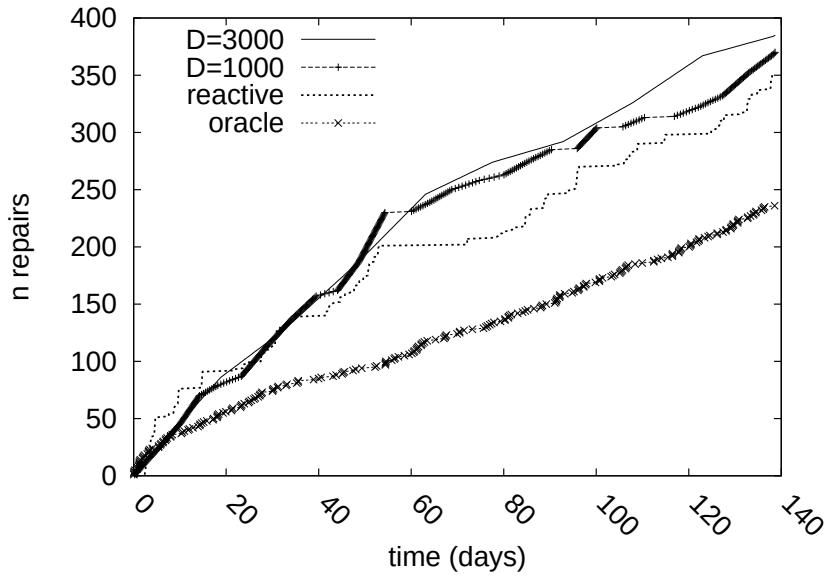
Figure 6.16: Proactive scheme with $D = 1000$ on KAD traces.

compared to the reactive scheme, which alternates between periods in which a high number of repairs are performed almost at the same time and periods of inactivity.

The fact that both curves for the proactive scheme end in a higher point, i.e. more repairs are done, can be interpreted considering that the property of a proactive scheme is to do work in *advance* to spread it over time. Having this in mind it is clear that is likely that the instantaneous number of repairs done by such a scheme is higher than the one done by a reactive scheme. Such a property is useful to have a safer *availability*, but as explained for the case of the oracle, may represent in part a *wasted* work, because of the parity blocks that disappear before their actual need.



(a) Distribution of available parity blocks.



(b) Cumulative number of repairs.

Figure 6.17: Simulation with KAD traces.

In general, proactive schemes tend to have a higher number of repairs. This, as already suggested, is due to the fact that proactive schemes work in *advance* to spread the repairs over time. *This anticipation of repairs is the only way to smooth the rate without compromising the durability*. The price to pay, however, is that part of the parity blocks created in advance may be lost, because of permanent failures, before they are needed [34]. The higher the death probability P_{death} the more pronounced this effect will be.

6.8 Conclusion

We proposed a novel framework for managing redundancy based on the estimation of the peer behavior.

Our system combines the resilience of reactive schemes with the smoothness of proactive schemes. It can be considered as a generalization of the purely proactive and reactive schemes, in which the duality reactive or proactive becomes a specific case of a wider approach tunable with respect to the dynamics of the failure behavior.

We validated the proposed scheme and demonstrated its effectiveness using synthetic data and availability traces of PlanetLab nodes.

Some of the open issues and the future work are outlined in the following.

More detailed experiments. It would be interesting to quantify metrics such as the number of (useless) parity blocks that were not needed to provide availability/durability or the number of (wasted) parity blocks that are lost before being used. To do so, an association between parity blocks and specific peers must be simulated. Such an association adds substantial complexity to the simulator and, moreover, would involve a model of the policy adopted to choose peers that store the newly generated parity blocks. These aspects could be easily investigated adopting the scheme presented in a real prototype.

Automatic tuning of the parameter D. Even if the ability of the system to provide durability is not very sensitive to the choice of D, it would be nice to find a way to auto-tune D. The objective would be to build a system that automatically finds a good tradeoff between reactivity and smoothness of repair.

Chapter 7
Conclusion

7.1 Summary

The need of cheap data backup for ordinary users motivates the research into peer-to-peer data backup systems. Such systems combine the practical advantages of online backup services with the economic advantages of peer-to-peer approach.

Our contributions focus on data redundancy and data maintenance. We looked at these two topics from a cost perspective with the objective of understanding the trade-offs that these two design aspects arose and how these trade-offs impact the system economy. From this point of view, we performed, as detailed in chapter 3, a cost analysis of an ideal peer-to-peer data backup system. In simple terms, this analysis shows that, while replication is able to consume little communication bandwidth for maintenance, it requires a lot of storage space; on the contrary, erasure coding is able to save storage space but requires a large amount of maintenance communication bandwidth.

Our goal is to find alternative redundancy schemes and maintenance algorithms able to overcome the existing duality between replication and erasure coding and provide both, storage efficiency and bandwidth efficiency. This goal guided our research, whose contributions are as follows.

Regenerating Codes

Regenerating Codes reduce the communication cost due to repair activity. The original work by Dimakis proved the existence of such codes and allowed to study their efficiency. We realized that Regenerating Codes are very interesting, but that a more applied and practical analysis is needed to understand properly the properties of Regenerating Codes.

We proposed an implementation of Regenerating Codes based on random linear codes and we performed an analytical and experimental evaluation of the costs generated by these codes. Our conclusions are as follows:

- Regenerating Codes are able to reduce significantly the communication bandwidth needed by repairs.
- Regenerating Codes may introduce in some configurations a significant computation cost that can become the system bottleneck.
- Regenerating Codes generalize the trade-off given by erasure codes and replication. This generalized trade-off includes not only communication bandwidth and storage but also computation.
- In the design space of Regenerating Codes it is possible to find points that optimally exploit the resources present in the system. These points, however, are able to provide reasonable computation costs only for repair operations, while insert and retrieval operations remain significantly expensive. This fact led us to conclude that Regenerating Codes are best suited for applications that do not retrieve data very often, which is the case of data backup systems.

Hierarchical Codes

We proposed Hierarchical Codes, which are a class of codes that aim, like Regenerating Codes, to reduce the repair communication cost. Hierarchical codes compute the parity fragments as random linear combinations of a *subset* of the original fragments. This choice allows to reduce the average repair degree needed for repairs.

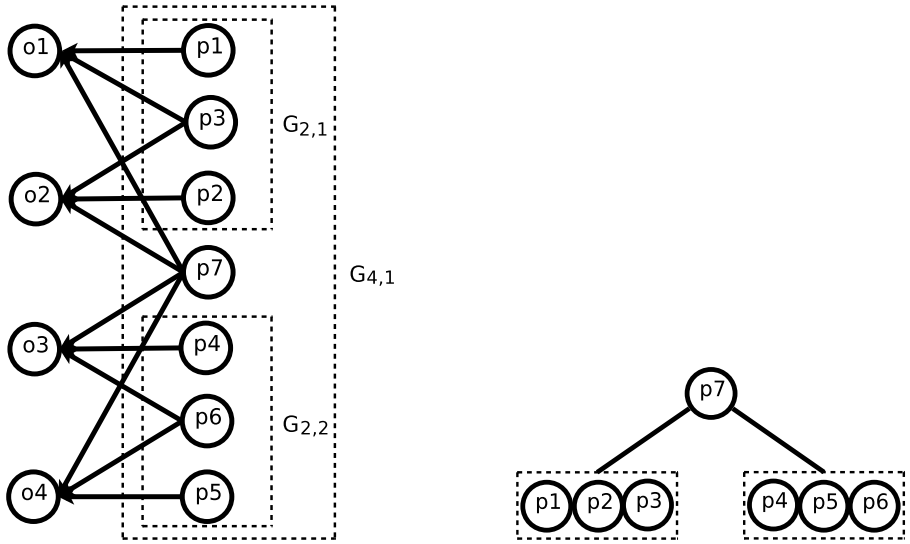
We presented the theoretic framework of Hierarchical Codes, proposed a formal analysis of their efficiency, and then valuated them experimentally. The results obtained are as follows:

- Hierarchical Codes are able in certain configurations to reduce the repair communication costs. They usually require a higher number of repair operations, which however in most cases entail only a small number of fragment transfers, which reduces the communication cost.
- Hierarchical Codes with a given redundancy factor allow a flexible trade-off between reliability and repair cost. The system designer may tune Hierarchical Codes to have a behavior that can approach either block replication side or traditional erasure codes side.

Adaptive Proactive Repair Policies

Previous work in the literature distinguished between two kinds of repair policies: reactive policies and proactive policies. While reactive policies are able to well-adapt to peer behavior, they tend to be bursty in the repair activity. Proactive schemes on the contrary make a smooth use of bandwidth, but they are not able to adapt to the changes in the peer behavior.

Our approach is to design an *adaptive proactive* repair scheme, able to overcome this duality between reactive and proactive policies. Our scheme is based on an ongoing estimation of the peer behavior.



(a) Code graph **(b)** Code hierarchy

Figure 7.1: Code graph and hierarchy for a hierarchical (4,3)-code

We tested our adaptive proactive repair scheme using synthetic and real connectivity traces, and showed that:

- Our policy represents a flexible and tunable solution that allows the system designer to select the behavior of the repair policy, trading reactivity for adaptiveness.
- The feedback control system we propose is able to provide data durability and to increase the smoothness of the bandwidth utilization as compared to reactive schemes.

7.2 Open Issues and Future Work

7.2.1 Data Placement and Hierarchical Codes

We already discussed the importance of data placement due to the fact that peers do not behave randomly. In particular, we mentioned that peers can be clustered accordingly to their availability or lifetime. In section 1.3.2 we discussed how data placement strategy can exploit the different behaviors of peers to increase data availability or durability. An idea that might be investigated is to exploit the features offered by Hierarchical Codes to conceive effective data placement.

One of the features of Hierarchical Codes is to define parity fragments with different *importance*. If we consider the sample hierarchical (4,3)-code we presented in section 5.4 and whose code graph is showed in Fig. 7.1a, we can define a hierarchy among the parity fragments: parity fragment p_7 is the most important one, since its loss requires always a repair degree of $d = 4$, while all the other parity fragments are less important, since if (only) one of them is lost the repair degree is $d = 2$. This hierarchy is represented by the tree structure in Fig. 7.1b.

Our intuition is that if we are able to put more important parity fragments on more stable peers, we can achieve higher reliability and we can reduce the repair bandwidth communication. In the example of Fig. 7.1b, if we could ideally put the parity fragment p_7 on a node that is always connected, the system could always use a repair degree $d = 2$ (as long as no concurrent failures occur).

The challenges that need to be addressed are twofold:

- It is not evident how to determine which are the nodes that are more stable (or with longer lifetime). We think that one solution could be to exploit the heavy-tailed distribution of peer online time [74], which means that peers that stayed connected longer have an higher probability to stay connected even longer. In this case, peers that have been connected the longest would store parity fragments that are on the top of code hierarchy.
- The concentration of important parity fragments on stable peers may have two collateral effects: (1) stable peers will be involved in a high number of repairs, creating an unbalanced use of resources. (2) when stable peers disconnect, a lot of high-cost repairs must be done, which can cause a bursty utilization of the network.

7.2.2 Growing Hierarchical Codes

In the design of Hierarchical Codes we considered parity fragments as static, in the sense that they occupy a specific position in the tree and never move. Actually, one could allow them to move down or up the hierarchy. We can say that a parity fragment that has moved down in the hierarchy is downgraded, since it loses importance, while a fragment that has moved up is upgraded. It is easy to understand that downgrading a fragment in any position in the hierarchical sub-tree (the fragment is root of) does not pose any problem: in the example of Fig. 7.1b, p_7 can be downgraded to any position, since it is given by a random linear combination of all the original fragments. On the contrary a fragment cannot be upgraded as it is, but it needs to be linearly combined with other fragments: in the example of Fig. 7.1b, p_3 must be combined also other two fragments out of p_4, p_5 and p_6 to be upgraded at the level of p_7.

As amply discussed in chapter 6, it is very important to spread bandwidth utilization over time, while repair policies tend to alternate periods of intense activity with idle periods. Our idea is that one can exploit the idle periods to upgrade fragments, combining them with a *growing* number of other fragments. This work, which can be done in the background, is not useless, since increasing the number of high-degree fragments, decreases the probability of *repairs* with high repair degree. In the example of Fig. 7.1b, if p_3 is upgraded to the level of p_7, even if p_7 is lost, the repair degree would be $d = 2$ instead of $d = 4$.

It would be interesting to evaluate the effect of this technique on the smoothness of the bandwidth utilization. Moreover this upgrade process would automatically store the high-degree fragments on peers that are connected longer.

7.3 Concluding remarks

The contributions presented in this dissertation focused on two specific topics in the design of a specific system, which is a peer-to-peer data backup system for ordinary users. We believe that these contributions help advance the state of the art:

- We offered a better understanding of the cost trade-offs of such a system.
- We proposed some tunable and reusable techniques to determine the optimum in these trade-offs.

One of the questions that we ask ourselves in retrospective is whether our contributions may be useful beyond the specific target context we addressed.

Big companies, like Google, Microsoft, and Amazon, provide the user with services that require an impressive amount of storage capacity and computation power. Examples of these services are search engines, mail services, social networks, web applications etc. For this reason, there has been an increasing interest in the construction of efficient data centers. The main trend in the design of such data centers is to use commodity hardware in large quantities, instead of high-quality and high-capacity server farms. This choice reduces sensibly the costs, but poses a number of challenging problems. For example, commodity hardware fails at a rate considerably higher than high-quality servers, and the communication among many independent units requires the design of complex network infrastructure, which may become highly loaded [5]. These issues make the design of data centers to some extent similar to the design of traditional peer-to-peer systems and for these reasons we believe that our contributions may be useful also in this domain. This is especially true if one considers that our main objective is the efficient utilization of the communication bandwidth, which is also one of the most critical resources of a data center.

Another important point is that data centers could in the near future use also resources that are at the edge of the network, i.e. they could exploit resources present in the user home, such as the home gateway. Some moves in this direction have already been done (see for example the NaDa project [14]): Internet service providers provide the users with devices (set-top boxes) that are very similar to personal computers and that are managed by the Internet service provider. The resources of these devices could be used to create peer-to-peer services, and a backup service could be one of them.

APPENDIX A

Synthèse en Français

A.1 Motivation

A.1.1 La nécessité d'un système de sauvegarde de données pour les utilisateurs

La quantité de données produite dans le monde augmente à un rythme exponentiel. Un'étude réalisée en 2003 [71] estime qu'environ 5 exaoctets ($5 \cdot 10^{18}$ octets) de données originales ont été produites en 2002, soit une augmentation de plus de 30% par rapport à l'année précédente. Une partie de cette croissance a été alimentée par les données numériques produites par les utilisateurs. La Fig. A.1 ([71]) montre que la quantité de données numériques originales stockés annuellement par les utilisateurs sur le disque dur de leur ordinateur personnel a augmenté de deux ordres de grandeur de 1996 à 2003. Cette prolifération est en grande partie due à la numérisation de l'information : photos numériques, vidéos numériques, et le courrier électronique sont devenus partie de la vie quotidienne de tout le monde.

Tout cela dit, il est évident que la sauvegarde des données est devenue une nécessité urgente pour les utilisateurs. De toute évidence, les techniques de sauvegarde de fichiers ne sont pas une nouveauté : les entreprises ont été confrontés au problème de perte de données pendant des décennies et sont équipées avec des systèmes de sauvegarde. Toutefois, les coûts supportés par les entreprises pour ces systèmes ne sont pas abordables par les utilisateurs ordinaires, qui doivent trouver des solutions à bas prix pour stocker leurs données en toute sécurité.

A.1.2 Le pair-à-pair est la solution

Il existe un besoin croissant de solutions de sauvegarde des données, et les solutions existantes ont beaucoup de limites en termes de fiabilité et de coûts. Ces deux facteurs ont pousse à chercher des nouvelles façons pour sauvegarder les données des utilisateurs.

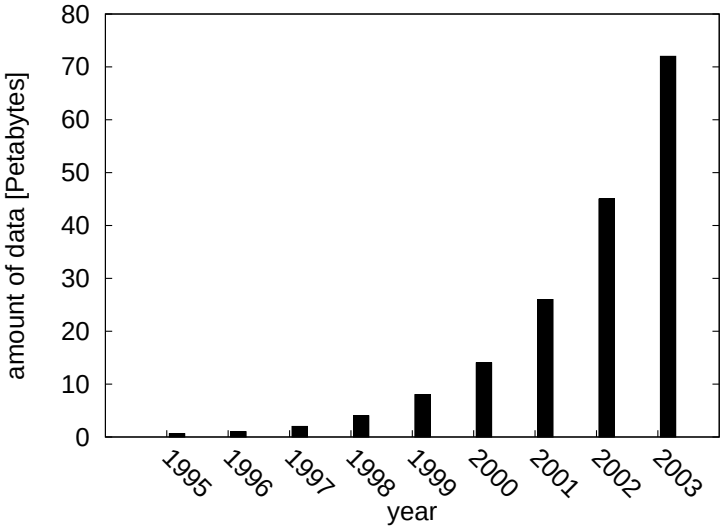


Figure A.1: Quantité de données numériques produites par an par les utilisateurs et stockées sur les disques dures (in Petaoctets : 10^{15} octets).

Depuis plusieurs années les chercheurs ont proposé de développer des systèmes pair-à-pair pour le stockage et la sauvegarde des données.

L'architecture traditionnelle des applications distribuées suit le paradigme client-serveur, qui se compose de deux entités distinctes :

- **Le serveur**, qui fournit le service et toutes les ressources nécessaires pour le service.
- **Le client**, qui utilise le service et exploite les ressources fournies par le serveur.

La principale caractéristique des réseaux pair-à-pair est la fusion de ces deux rôles. Dans un réseau pair-à-pair tous les pairs jouent à la fois le rôle du serveur et du client : tous les pairs contribuent au service et partagent une partie de leur ressources et, au même temps, ils sont utilisateurs du service. L'approche pair-à-pair permet d'avoir deux propriétés très intéressantes :

- ***self-scaling***, qui signifie que la quantité des ressources disponibles augmente avec la demande.
- ***fault-tolerance***. L'organisation décentralisée des réseaux pair-à-pair profite de l'indépendance des pairs pour fournir un bon niveau de fiabilité.

Dans un tel système de stockage, chaque pair consacre une partie de son espace de stockage à la communauté et en échange la communauté (les autres pairs) stocke de manière fiable ses données.

A.1.3 Faisabilité d'un système de stockage pair-à-pair

Un système de stockage pair-à-pair ne peut être réalisé que si les pairs participants ont suffisamment d'espace de stockage. La Fig. A.2 ([8]) montre l'évolution de la capacité des dis-

ques durs au cours des 30 dernières années, indiquant une croissance exponentielle de la capacité de stockage. Cette croissance n'a pas été associée à une croissance correspondante des prix, qui implique une diminution exponentielle du coût par octet. La disponibilité de stockage bon marché suggère que les utilisateurs peuvent facilement avoir beaucoup d'espace de stockage non utilisé. Cette intuition est confirmée par une étude menée sur les ordinateurs de bureau au sein de Microsoft [46], qui montre que les disques durs en moyenne ne sont qu'à moitié plein.

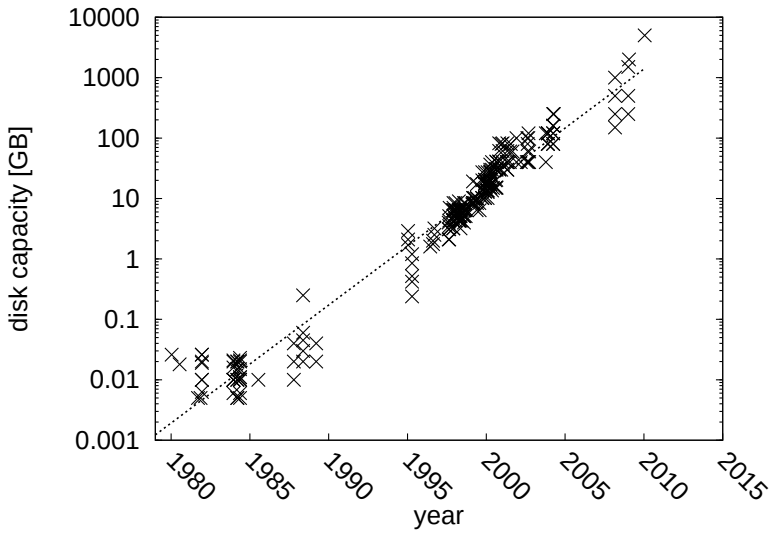


Figure A.2: Evolution de la capacité des disques dures dans le temps.

A.2 Description d'un système de stockage pair-à-pair

La conception d'un système de sauvegarde de fichiers pair-à-pair est une tâche complexe, ayant un grand nombre d'aspects différents et pose un certain nombre de problèmes. Avant de discuter les différentes questions impliquées dans une telle conception, nous proposons une description du système à partir de deux points de vue différents.

Premièrement, nous définissons les propriétés qui sont attendues du système. Nous proposons une sorte de contrat de service qui indique quelles sont les fonctionnalités qu'un système de stockage doit offrir à l'utilisateur.

Deuxièmement, nous décrivons les contraintes que nous avons sur la construction du système. En pratique, nous décrivons la nature pair-à-pair du système et nous expliquons en détail le comportement attendu et les caractéristiques des pairs participants.

A.2.1 Définition d'un service de stockage de données

Essentiellement, un système de stockage de données devrait fournir un stockage fiable et sécurisé des données. En particulier, l'utilisateur aura deux primitives très simples : *Store Data* et *Retrieve Data*, qui correspondent à l'insertion des données dans le système et à leur

récupération. Le système doit garantir les propriétés suivantes par rapport à ces deux primitives :

- **Durabilité des données** Les données qui sont insérées dans le système sont stockés de façon fiable et *jamais perdues*.
- **Disponibilité des données**. La disponibilité des données implique la capacité de récupérer les données à la demande des utilisateurs. Notez que cette propriété diffère de la durabilité : un objet stockée pourrait être durable, mais pas disponible à certaines périodes.
- **Confidentialité des données**. Les données insérées ne peuvent être lues que par un groupe d'utilisateurs autorisés.

A.2.2 Contraintes de l'environnement pair-à-pair

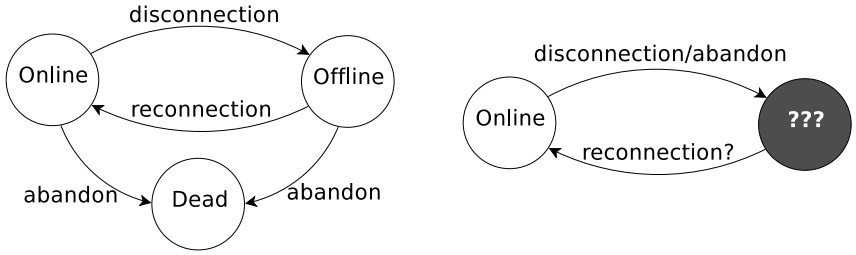
La propriété essentielle d'un système pair-à-pair est que les ressources sont fournies par les pairs qui participent dans le système. Dans le cas des systèmes de stockage de données, la ressource la plus importante est la capacité de stockage. Chaque pair fournit une partie de son espace de stockage, utilisé par le système pour fournir un service de sauvegarde fiable.

Les pairs participants au système, cependant ne sont pas sous le contrôle du système lui-même. Cependant, les propriétés du service doit être garanties en dépit de ce manque de contrôle. Le comportement des pairs peut être caractérisé comme suit :

- **Connectivité intermittente**. Les pairs ne sont pas connectés au système tout le temps des raisons différentes : les utilisateurs peuvent se déconnecter de l'Internet, les machines peuvent être redémarrées, ou à cause des pannes temporaire de réseau. Cette connectivité intermittente implique que les données stockées peuvent être périodiquement indisponibles.
- **Déconnexions permanentes**. Au-delà des déconnexions temporaires, les pairs peuvent quitter le système et les données stockées sur ces pairs sont perdus. L'événement de perte de données peut survenir en raison de départs volontaires, mais aussi en raison de défaillances ou de suppression des données (accidentelle ou volontaire).
- **Mauvais comportement des pairs**. Les pairs ne sont pas fiables. Ils peuvent se comporter mal ou de manière inattendue.
- **Bande passante limitée**. Les pairs sont connectés au système via une connexion avec une bande passante limitée. Cette limitation peut être due à des contraintes réelles sur le lien d'accès ou à un plafond d'utilisation de bande passante que l'utilisateur peut poser sur l'application pair-à-pair .

Pour comprendre le comportement des pairs, nous introduisons dans la Fig. A.3a un automate fini qui modélise l'évolution du cycle de vie d'un pair. Chaque pair alterne des périodes dans lesquelles il est en ligne (*online*) et des périodes où il est déconnecté (*offline*). Cette alternance constitue ce que nous avons défini comme connectivité intermittente. Après une période, appelée **lifetime**, les pairs peuvent abandonner le système et deviennent morts (*dead*).

Nous appelons cet événement **déconnexion permanente**, et, comme déjà mentionné, une déconnexion permanente se traduit par la perte de données.



(a) Comportement reel **(b)** Comportement observé

Figure A.3: Automate fini modélisant le comportement d'un pair.

Il est important de remarquer que le comportement réel des pairs n'est pas observable, car le système n'est pas en mesure de distinguer les déconnexions temporaires des déconnexions permanentes. Dans la Fig. A.3b, nous montrons l'automate fini comme il est observable par le système. Le manque de connaissance sur l'état réel des pairs, et donc des données qu'ils stockent, rend la maintenance des données difficile.

A.2.3 Systèmes de fichiers vs. systèmes de sauvegarde de données

Bien que l'accent principal de cette thèse porte sur les systèmes pair-à-pair de sauvegarde de données, ces systèmes partagent de nombreuses caractéristiques avec les systèmes de fichiers pair-à-pair . En effet, une partie considérable de la littérature aborde des questions qui se rapportent au domaine générique de systèmes de stockage pair-à-pair . Pourtant, les systèmes de sauvegarde des données et les systèmes de fichiers diffèrent beaucoup en termes d'hypothèses et de propriétés requises et ces différences peuvent avoir une forte influence sur les choix de conception. Dans cette section, nous soulignons ces différences et discutons de la manière dont elles se reflètent au niveau de la conception.

A.2.4 Disponibilité et durabilité des données

Nous avons introduit les notions de disponibilité et de durabilité des données. Il est évident que la disponibilité implique la durabilité, alors que l'inverse n'est pas toujours vrai : un objet durable ne sera pas disponible à certaines périodes. Une première différence entre les systèmes de fichiers et les systèmes de sauvegarde consiste dans l'importance différente donnée à la disponibilité des données et à la durabilité des données. Il est évident que la disponibilité est essentielle dans les systèmes de fichiers, puisque la lecture et l'écriture des fichiers sont des opérations qui ne peuvent pas tolérer de retard, ce qui implique qu'un fichier doit être disponible en pratique à n'importe quel moment. Dans les systèmes de sauvegarde de fichiers, par contre, le propriétaire des données doit accéder à ses données qu'en cas de perte et peut accepter un certain retard pour leur reconstruction.

A.2.5 Mises à jour de données

Dans les systèmes de fichiers, les fichiers ne sont pas seulement lus, mais également modifiés. Un grand défi dans la conception de systèmes de stockage pair-à-pair est de gérer les mises à jour des données et garantir leur cohérence. Par contre, dans des systèmes de sauvegarde, les données pourraient être considérées comme immuables (voir par exemple [81]).

A.3 Thème central de la thèse

L'idée d'un système de stockage distribué basé sur le paradigme pair-à-pair n'est pas nouvelle. Elle a intéressé la communauté des chercheurs depuis une dizaine d'années et il existe un bon nombre de publications sur ce sujet. Ces publications peuvent être mises dans l'une des deux catégories suivantes :

- **Des systèmes complets**, qui décrivent des systèmes de stockage complets et représentent une *proof-of-concept*. Bien qu'ils proposent des solutions et des techniques intéressantes, la complexité de la conception empêche souvent les auteurs de faire un examen détaillé des choix de conception qu'ils font.
- **Solutions de sous-problèmes bien définis** . Ces publications se concentrent plutôt sur des aspects spécifiques et fournissent une analyse et des résultats qui sont réutilisables par d'autres qui veulent construire un système complet.

Nous avons décidé de suivre cette deuxième approche : les contributions de cette thèse visent à fournir des *blocs de base* capables de résoudre des problèmes spécifiques de manière efficace.

Parmi les aspects impliqués dans la conception de systèmes de sauvegarde pair-à-pair , nous concentrons notre attention sur des aspects spécifiques à la fiabilité des données stockées, tels que les schémas de redondance et les politiques de réparation. Nos contributions se portent sur un point essentiel : la bande passante est une ressource limitée. Ce fait a été ignoré dans de nombreux travaux de recherche qui ont enquêté sur la redondance des données et sa maintenance.

Beaucoup de schémas de redondance se concentrent sur l'utilisation efficace du stockage, tout en payant un prix élevé en termes de communication. Les codes correcteurs classiques en sont un exemple. Ils sont capables de consommer très peu d'espace de stockage, mais ils consomment beaucoup de bande passante pour la réparation des données. Notre intuition est qu'il est essentiel d'étudier des systèmes de redondance spécifiquement conçus pour le stockage distribué, qui tiennent compte à la fois de l'espace de stockage et de la bande passante. Par rapport à ce sujet nous proposons et étudions des codes correcteurs pour la redondance capables de combiner l'efficacité en bande passante de la réplication à l'efficacité en stockage des codes correcteurs classiques. En particulier, nous présentons et analysons deux nouvelles classes de codes: Regenerating Codes et Hierarchical Codes.

Comme dans le cas des schémas de redondance, les politiques de réparation ont besoin de réduire la consommation de bande passante. Dans ce cas, cependant, la question n'est pas *combien* de bande passante faut-il pour les réparations (car cela est déterminé par les schémas de redondance), mais *quand* cette bande passante est nécessaire. Une observation importante est

que la bande passante *ne peut pas être conservée pour une utilisation future* : la bande passante non utilisée est simplement perdue. Les politiques de réparation plus utilisées sont réactives et basées sur des seuils : quand la quantité de redondance se réduit à un niveau trop bas (elle atteint le seuil), le système remplace les données perdues. Ces politiques ont tendance à faire un usage très irrégulière de la bande passante. Un'autre catégorie est représenté par les politiques proactives, qui sont en mesure de consommer la bande passante d'une façon régulière, mais ils ne sont pas capables de s'adapter correctement aux fluctuations de comportement de pairs. Pour cette raison, nous nous concentrons sur la conception d'algorithmes de maintenance de données qui visent à la fois à avoir une utilisation régulière de la bande passante et à s'adapter au comportement réel des pairs. Nous proposons un système de réparation, nommé "adaptive proactive repair scheme", qui combine l'adaptabilité des systèmes réactifs avec l'utilisation régulière de la bande passante des systèmes proactifs, en généralisant les deux approches existantes.

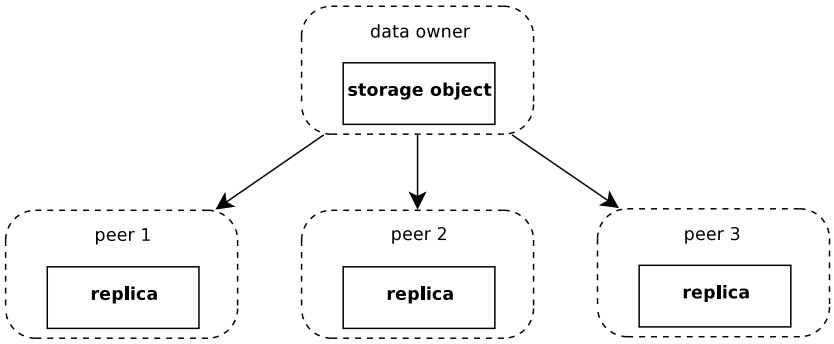
A.3.1 redondance des données

La redondance des données consiste à stocker plusieurs instances des mêmes données sur des pairs différents. Grâce à cette technique, même si une partie des données stockées est indisponible, le reste devrait être suffisant pour reconstruire les données d'origine.

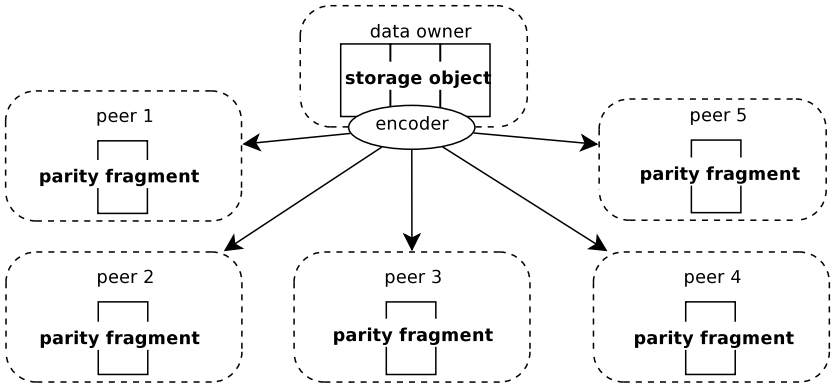
Il existe plusieurs de techniques différentes pour ajouter de la redondance aux données et ils sont généralement appelés **schémas de redondance**. Chaque schéma de redondance détermine (i) comment créer les données redondantes et (ii) la façon de reconstruire les données redondantes lorsqu'elles sont perdues. Ces deux opérations génèrent des coûts qui varient d'un schéma à l'autre. Ici, nous introduisons les schémas de redondance les plus largement utilisés : la réplication et les codes correcteurs.

La façon la plus simple de stocker des données de manière redondante est la **réplication** : Si nous stockons N_{rep} répliques d'un même fichier, même si $N_{rep} - 1$ de ces répliques sont stockées sur des pairs qui ne sont pas disponibles, nous sommes encore en mesure de récupérer notre fichier. L'insertion de données pour la réplication est représentée dans la Fig. A.4a pour le cas $N_{rep} = 3$. Dans le cas de la réplication, la reconstruction d'une réplique perdue est simple, puisqu'il suffit de créer une copie exacte d'un autre réplique, comme le montre la Fig. A.5a.

Un autre schéma de redondance est représenté par les **codes correcteurs**. Un objet qui doit être stocké dans le system est divise en k morceaux, que l'on nomme **fragment d'origine**, puis ces fragments d'origine sont codés pour obtenir $k + h$ **fragments de parité**, tel que chaque k d'entre eux sont suffisantes pour reconstruire l'objet original. L'insertion de données pour les codes correcteurs est décrite dans la Fig. A.4b dans le cas où $k = 3$ et $h = 2$. Notez que les codes correcteurs sont capables de consommer moins d'espace de stockage par rapport à la réplication, offrant le même niveau de fiabilité. En effet, considérons le schéma de réplication (avec $N_{rep} = 3$) et le code correcteur (avec $k = 3$ et $h = 2$) de la Fig. A.4. Dans les deux cas, le système peut accepter jusqu'à deux pairs déconnectés sans perdre les données originales : dans ce cas la réplication consomme de l'espace de stockage correspondant à 3 fois la taille des données d'origine, tandis que le code correcteur ne consomme que $5/3$ de la taille des données d'origine.



(a) Réplication avec 3 répliques.



(b) Code correcteur avec $k = 3$ et $h = 2$

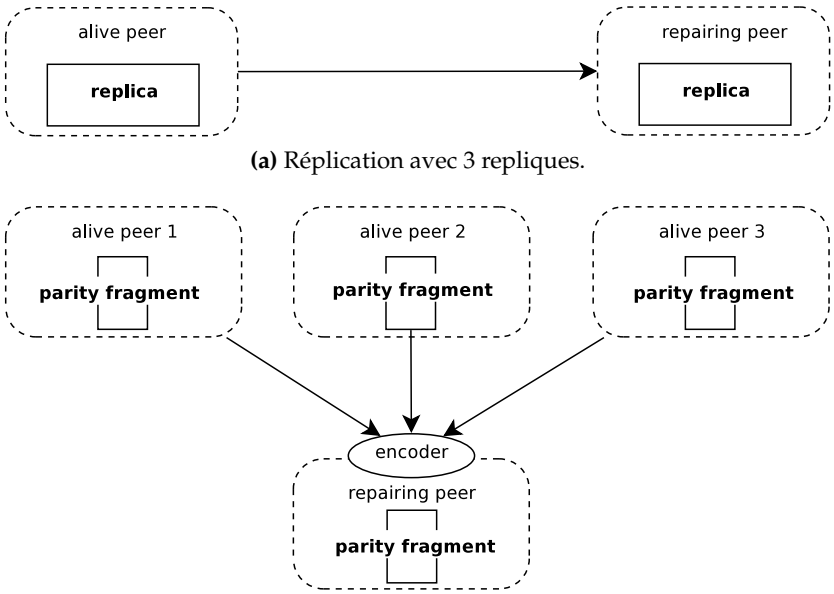
Figure A.4: Schémas de redondance : Insertion des données.

Dans les codes correcteurs, les réparations sont plus complexes que dans la réplication. Quand un fragment de parité est perdu, sa reconstruction nécessite l'accès à k autres k fragments de parité. Notez que pour exploiter la redondance des données, il est essentiel que les répliques ou les fragments de parité soient stockés sur des pairs différents. Notre choix, implicitement introduit dans la Fig. A.4, est que *chaque réplique ou fragment de parité est stocké sur un pair différent*. Ce choix permet de maximiser la probabilité de la disponibilité des données, car il exploite au maximum la diversité des pairs, mais implique que toutes les lectures de données correspondent à des *transferts de données*. Quand une réparation d'un fragment de parité est réalisée, le système a besoin de télécharger k autres fragments de parité, qui se traduit par le transfert de données d'une quantité de données égale à la taille de l'*objet entier*. Cette procédure de réparation pour les codes correcteurs est décrite dans la Fig. A.5b dans le cas où $k = 3$ et $h = 2$.

Les politiques de réparation

A cause da la perte de données il est nécessaire que les données stockées soient périodiquement réparées.

La politique de réparation définit la période de réparations, c'est-à-dire le moment où de nouvelles répliques ou de nouveaux fragments de parité doivent être créés. Cette décision n'est pas facile , car elle doit être prise sans avoir une connaissance complète de l'état des



(a) Réplication avec 3 repliques.

(b) Code correcteur avec $k = 3$ et $h = 2$

Figure A.5: Schémas de redondance : Réparation des données.

données : le système est en mesure d'indiquer si des données sont indisponibles, mais ne peut pas affirmer si ces données seront disponibles à nouveau ou si elles sont définitivement perdues.

Une politique de réparation doit au moins garantir que les données stockées sur les pairs qui abandonnent le système, c'est-à-dire les déconnexions permanentes, soient réparées. Pour cette raison, une approche très triviale est de réparer toutes les données qui ne sont plus disponibles, peu importe si elles seront de nouveau disponibles ou pas. Cette politique, que l'on appelle **"eager policy"**, est cependant très coûteuse en termes : (i) de stockage, car elle consomme plus d'espace que ce qui est strictement nécessaire et (ii) de bande passante, étant donné que chaque réparation correspond au transfert d'une quantité de données égal à la taille de l'objet entier.

Des politiques de réparation plus intelligentes s'efforcent de réduire ces coûts en évitant les réparations inutiles. Une approche est basée sur un *minuteur*, ce qui signifie que le système considère comme définitivement perdues les données qui restent inaccessibles pour une période de temps qui dépasse un seuil de temps. Dans cette politique, que l'on appelle **"lazy policy"**, le choix de ce seuil est essentiel, car une valeur trop courte peut causer des réparations inutiles, tandis qu'un valeur large peut détecter les pertes permanentes trop tard, mettant en péril la durabilité des données.

Une autre "lazy policy" est basé sur un seuil de redondance, qui fixe le montant minimum de redondance qui doit toujours être disponible. Dans ce cas, les réparations sont déclenchées lorsque ce seuil est atteint. Cependant, cette approche, rend le processus de réparation très irrégulière, et par conséquent de nombreuses réparations seront effectuées dans une petite fenêtre de temps, alors que dans d'autres périodes aucune réparation ne sera effectuée. Ce comportement irrégulier provoque une utilisation inefficace de la bande passante de communication, qui devrait idéalement être utilisée le plus régulièrement possible.

Enfin, les politiques proactives essayent de choisir un taux fixe de réparation pour parvenir à une utilisation régulière de la bande passante. Toutefois, le choix du taux correct est très difficile et un mauvais choix peut causer soit des réparations inutiles soit des pertes de données.

A.3.2 Structure de cette synthèse

Dans cette synthèse, nous donnons un bref aperçu des deux sujets adressés dans cette thèse : les schémas de redondance des données et les politiques de réparation.

Pour ce qui concerne les schémas de redondance, nous avons sélectionné un des nos deux contributions : les "Regenerating Codes". Dans la section A.4 nous proposons une brève description des "Regenerating Codes" et nous décrivons quelles sont les principaux résultats de nos analyses et nos expériences.

Pour les politiques de réparation, dans la section A.5, nous décrivons notre politique de réparation d'adaptation proactive, illustrant les détails essentiels de notre schéma de contrôle.

A.4 Regenerating Codes

Les **"Regenerating Codes"** sont une classe de codes correcteurs qui fournissent presque la même efficacité de stockage que les codes correcteurs classiques, avec une réduction importante de la bande passante nécessaire pour les réparations.

La description originale des "Regenerating Codes" ne définit pas une façon pratique de les mettre en oeuvre et surtout n'étudie pas tous les coûts de tels codes et leur faisabilité dans le monde réel.

Notre contribution vise à combler cette lacune. Tout d'abord, nous donnons une description détaillée des propriétés théoriques des "Regenerating Codes", puis nous décrivons notre implémentation basée sur les codes linéaires aléatoires et, enfin, nous procédons à une évaluation analytique et expérimentale des différents coûts.

A.4.1 Notation pour le codes correcteurs

Avant d'entrer dans les détails des "Regenerating Codes", il est essentiel d'établir une notation unique pour les codes correcteurs.

Considérons un fichier, dont la taille est notée par $|file|$. L'application d'une code correcteur (k,h) au fichier consiste à créer à partir de ce fichier $k + h$ **blocs de parité** d'une façon telle que n'importe quels k blocs de parité de ces $k + h$ blocs soient suffisants pour reconstituer le fichier original. Une remarque importante est que dans cette définition générique, il n'y a pas de contrainte ni dans la façon dont ces blocs sont construits ni sur leur taille. En particulier,

la taille d'un bloc de parité est de $|block|$ avec la seule contrainte suivante :

$$|block| \geq \frac{|file|}{k}$$

A.4.2 Le cycle de vie d'un fichier

Dans cette section, nous décrivons le cycle de vie d'un seul fichier inséré dans le système. Ce cycle de vie est composé de trois étapes :

1. **Insertion :** L'insertion consiste à créer $(k + h)$ blocs de parité à partir du fichier et à les distribuer sur $(k+h)$ pairs. Quel que soit le schéma redondance utilisé, comme expliqué dans la section précédente, la propriété de ces blocs de parité est que n'importe quels k blocks sont suffisants pour reconstituer ce fichier.

2. **Maintenance :** La maintenance consiste en la reconstruction de la redondance perdue. La maintenance est effectuée avec les réparations. Une réparation nécessite la coopération de d pairs qui envoient des données à un nouveau pair, appelé **"newcomer"**, qui à son tour traite les données reçues pour obtenir un nouveau bloc de parité. Le paramètre d est appelé **"repair degree"**. Si la réparation est correctement exécutée, le nouveau bloc de parité a les mêmes propriétés que tous les autres, c'est-à-dire qu'avec n'importe quels $(k-1)$ blocs de parité, il forme un ensemble de blocs de parité suffisant pour reconstituer le fichier original.

3. **Reconstruction :** Si le propriétaire du fichier veut son fichier, une reconstruction doit être effectuée. La reconstruction consiste à télécharger des données à partir de k pairs et à les traiter pour obtenir le fichier original.

A.4.3 La quantification des coûts

Nous pouvons maintenant proposer une description formelle des coûts induits par chaque opération séparément. En particulier, il y a trois types de coûts :

1. **Stockage :** La redondance implique que le fichier stocké consomme plus d'espace de stockage que le fichier original. Les besoins de stockage sont facilement calculés par la formule ci-dessous:

$$|storage| = (k + h) \cdot |block| > |file|$$

2. **Communication :** Toutes les trois phases du cycle de vie d'un fichier nécessitent un transfert de données entre pairs. À l'*insertion*, tous les blocs de parité doivent être transférés, ce qui correspond à un volume de données de $|storage|$. À la *maintenance*, pour chaque réparation, chacun des d pairs transfère une quantité de données égal à $|repair_{up}|$ sur le "newcomer", pour un total de $|repair_{down}|$, où :

$$|repair_{down}| = d \cdot |repair_{up}|$$

   À la *reconstruction*, le propriétaire du fichier doit télécharger au moins une quantité de données égale à $|file|$

3. **Calcul :** Quand un code correcteur est utilisé, toutes les trois phases décrites nécessitent d'un traitement de données. À l' *insertion*, tous les blocs de parité doivent être codées avec un coût de *CPU(encoding)*. À la *réparation*, une partie du traitement est réalisé sur les d pairs participants, noté $CPU(repair)_{up}$, et une partie est réalisé sur le "newcomer", désignées par $CPU(repair)_{down}$. À la *reconstruction*, le fichier d'origine doit être reconstruit à partir de k blocs de parité avec un coût *CPU(reconstruction)*.

Le schéma de redondance définit la façon dont les données redondantes sont générées et manipulées ainsi que le coût en termes de calcul, communication et stockage. À titre d'exemple, considérons les codes correcteurs classiques. Ces codes ont les deux contraintes suivantes à l'égard du "repair degree" d et la taille des blocs de parité :

$$\begin{aligned} d &= k \\ |block| &= |file|/k \end{aligned} \tag{A.1}$$

En termes de maintenance, les coûts de communication sont les suivants :

$$\begin{aligned} |repair_{up}| &= |block| \\ |repair_{down}| &= |file| \end{aligned}$$

Cela signifie que pour chaque nouveau bit créé au cours d'une réparation, k autres bits doivent être transférés. Enfin, les coûts de calcul dépendent de l'implémentation.

A.4.4 Description des "Regenerating Codes"

Dans cette section, nous décrivons les propriétés principales des "Regenerating Codes" tels qu'ils ont été décrits à l'origine par Dimakis et al. [42, 43].

Les "Regenerating Codes" répondent à la question suivante : Si les contraintes définies pour les codes correcteurs classiques (eq. A.1) sont relâchées, quel serait l'impact sur le coût de communication?

Considérons un code correcteur (k, h); les "Regenerating Codes" peuvent prendre $k \cdot h$ valeurs différentes pour la paire de paramètres $(d, |block|)$. En effet les "Regenerating Codes" peuvent être considérés comme une généralisation des codes correcteurs classiques, qui font un compromis entre le coût de stockage et le coût de communication.

Plus formellement, un Regenerating Code générique, noté par $RC(k, h, d, i)$, fixe les contraintes suivantes sur le "repair degree" d et la taille des blocs de parité :

$$\begin{aligned} d &\in [k, k+h-1] \\ |block| &= p(d,i) \cdot |file| \quad i \in [0, k-1] \end{aligned} \tag{A.2}$$

Étant donné un "repair degree" d, le paramètre i, appelé **"block expansion index"** détermine la taille des blocs de parité à travers la fonction $p(d, i)$, qui est définie ci-dessous :

$$p(d,i) = 2\frac{d-k+i+1}{2k(d-k+1)+i(2k-i-1)}$$

Il peut être prouvé que pour un Regenerating Code $RC(k, h, d, i)$, chacun des d pairs participants à une réparation doit transférer au "newcomer" une quantité de données au moins égal à

$$|repair_{up}| = r(d, i) \cdot |file| \tag{A.3}$$

où $r(d, i)$ est définie ci-dessous :

$$r(d, i) = \frac{2}{2k(d - k + 1) + i(2k - i - 1)}$$

par conséquent :

$$|repair_{down}| = d \cdot r(d, i) \cdot |file| \tag{A.4}$$

la Fig. A.6 décrit comment la taille des blocs de parité et la quantité de données transférées lors d'une réparation $|repair_{down}|$ évoluent en fonction de d et i pour un code où $k = 32$ et $h = 32$. En particulier, toutes les valeurs sont normalisées par la taille des blocs de parité et le volume du trafic de réparation nécessaire pour un code correcteur classique qui, dans le cadre des "Regenerating Codes" correspond à $RC(32, 32, 32, 0)$, c'est-à-dire avec $d = 32$ et $i = 0$. Ces valeurs de référence sont :

$$\begin{aligned} |block| &= |file|/32 \\ |repair_{down}| &= |file| \end{aligned}$$

La Fig. A.6 montre que l'augmentation du "repair degree" d et celle de las taille des blocs de parité (augmentation du "block expansion index" i) impliquent *une très fort réduction du trafic de réparation*.

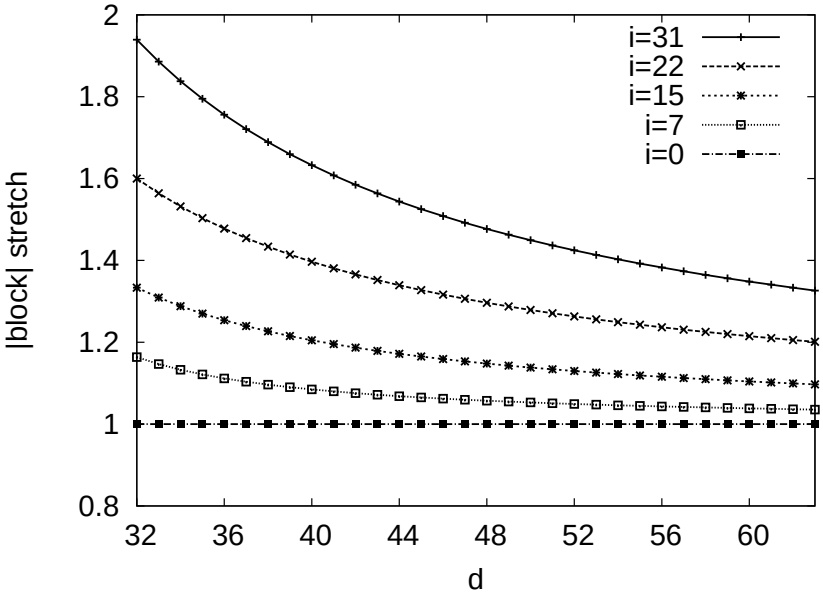
A.4.5 Evaluation expérimentale

Dans cette section, nous évaluons les besoins en ressources de calcul des "Regenerating Codes". Nous avons conçu et implémenté une version optimisée des "Regenerating Codes" basée sur les codes linéaires aléatoires et nous l'avons testée sur un processeur Intel Core 2 Duo à 2,66 GHz.

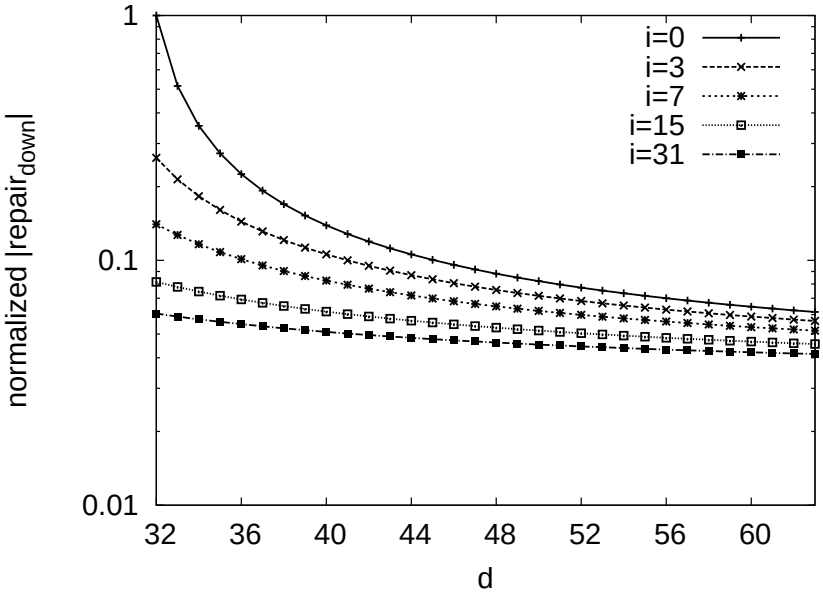
Toutes les opérations dans le cycle de vie d'un fichier sont exécutées et le temps nécessaire pour effectuer ces opérations est mesuré pour un fichier de 1 Mo. Toutes les expériences ont été faits pour un fichier de 1 Mo. Les paramètres des "Regenerating Codes" sont fixés à $k = 32$, $h = 32$, et nous faisons varier i et d.

Pour avoir une base de comparaison des différentes configurations des "Regenerating Codes", nous décrivons d'abord les résultats obtenus pour un code correcteur classique, (soit un "Regenerating Code" $RC(32, 32, 32, 0)$) lorsqu'un fichier de 1 Mo est stocké.

Si on note par $t_{d,i}$ le temps nécessaire par une opération pour un Regenerating Code $RC(32, 32, d, i)$, Table A.1 indique le temps $t_{32,0}$ nécessaire pour chaque opération.



(a) $|block|$ normalisé



(b) $|repair_{down}|$ normalisé

Figure A.6: Taille des blocs de parité et coût de communication pour les réparations normalisés par les valeurs d'un un code correcteur classique.

Notez que la réparation n'exige aucun temps de calcul pour un participant car celui-ci par définition envoie tout simplement un bloc de parité entier au "newcomer".

Décrivons maintenant les résultats obtenus pour le cas général des "Regenerating Codes" $RC(32, 32, d, i)$. Pour comprendre les coûts de calcul de ces codes, nous considérons le rapport entre le temps $t_{d,i}$ et le temps $t_{32,0}$ mesurés pour les codes correcteurs classiques. Nous

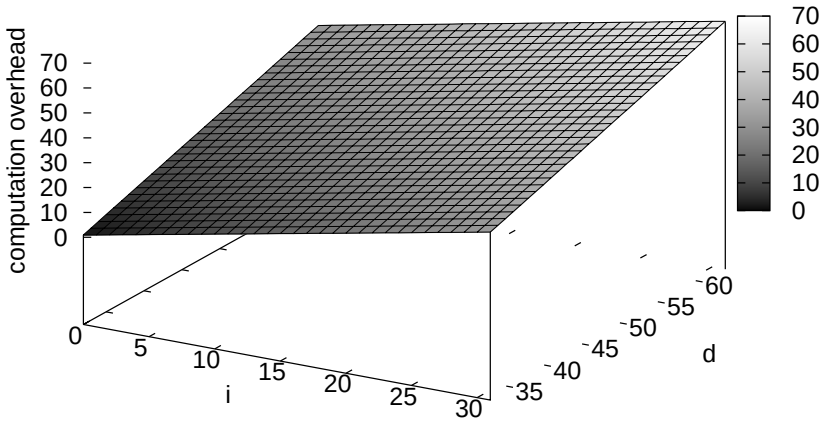
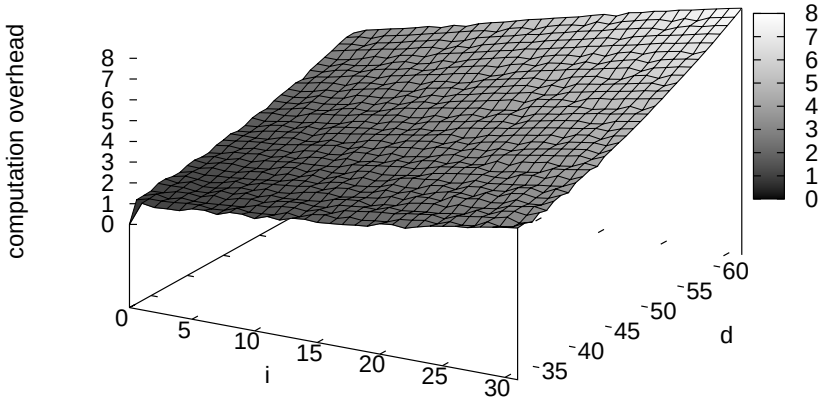
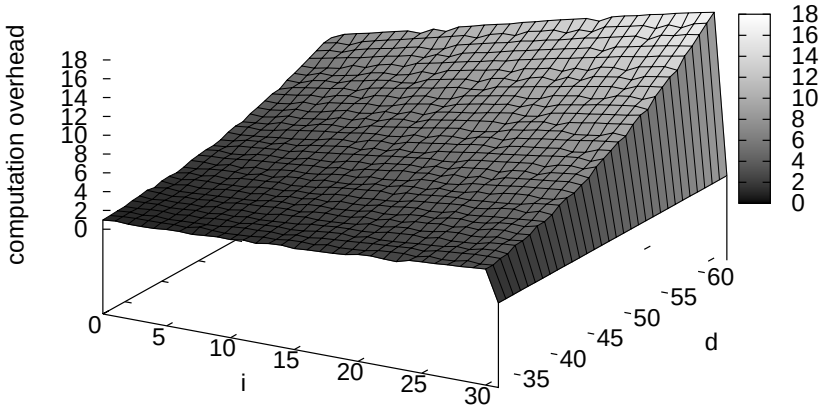


Figure A.7: *Computation overhead* pour l'encodage pour $RC(32, 32, d, i)$.



(a) Cote participant.

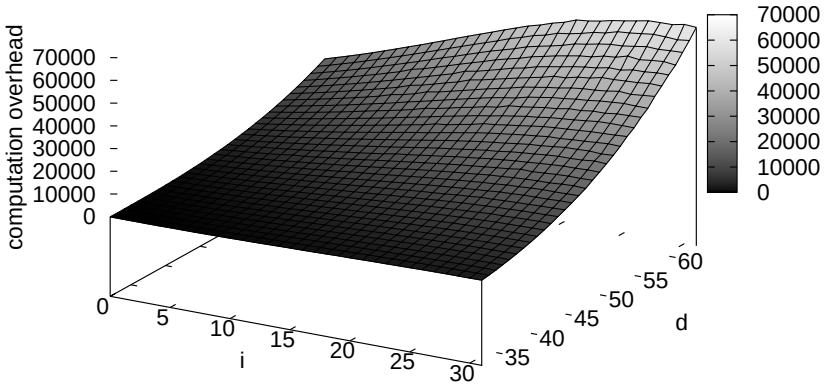


(b) Cote "newcomer".

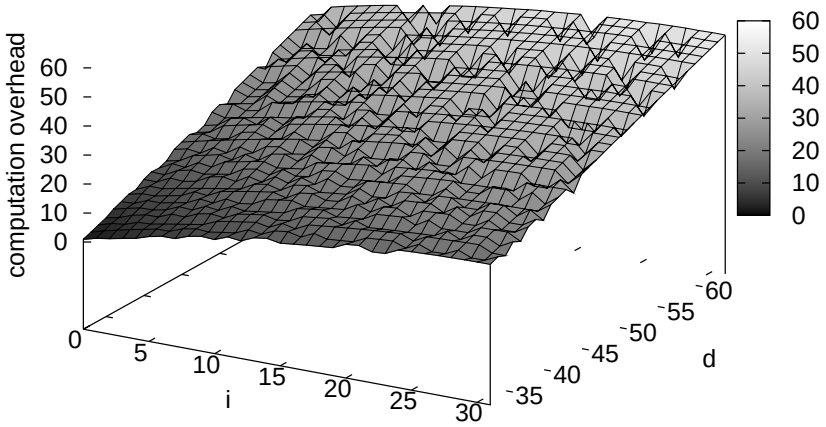
Figure A.8: *Computation overhead* pour les réparations pour $RC(32, 32, d, i)$.

appelons ce rapport **"computation overhead"** *coh* :

$$coh_{d,i} = \frac{t_{d,i}}{t_{32,0}}$$



(a) Inversion de matrice.



(b) Decodage.

Figure A.9: *Computation overhead* pour la reconstruction pour $RC(32, 32, d, i)$.

	$t_{32,0}[sec]$
Encodage	0.52
Réparation - côté participant	0
Réparation - côté "newcomer"	0.01
Inversion de matrice	0.002
Décodage	0.25

Table A.1: Temps nécessaire pour les opérations dans le cas d'un code correcteur classique.

Considérons le cycle de vie d'un fichier :

1. **Insertion :** la Fig. A.7 décrit le "computation overhead" pour l'encodage initial du fichier. Le graphique montre que le *overhead* augment linéairement avec i et d.

2. **Maintenance :** la Fig. A.8a montre le "computation overhead" pour une réparation du côté des participants, dans ce cas, ce coût augmente un peu plus que linéairement avec d et i. La Fig. A.8b décrit le "computation overhead" pour une réparation du côté du "newcomer".

3. **Reconstruction :** La reconstruction nécessite l'inversion d'une matrice de coefficients, puis le décodage du fichier. La Fig. A.9a décrit le "computation overhead" pour l'inversion, qui peut être très coûteuse, en particulier pour des valeurs de d et i grandes. La Fig. A.9b décrit le "computation overhead" pour le décodage.

A.4.6 Conclusion

Les "Regenerating Codes" sont capables de réduire la bande passante nécessaire pour les réparation; ils peuvent cependant introduire dans certaines configurations, un coût de calcul important qui peut devenir le goulot d'étranglement du système. Ils peuvent être considérés comme une généralisation des codes correcteurs classiques et de la réplication. Ils permettent de trouver un bon compromis non seulement entre les coûts de communication et de stockage, mais aussi de calcul. Nous avons schématisé cet aspect dans la Fig. A.10.

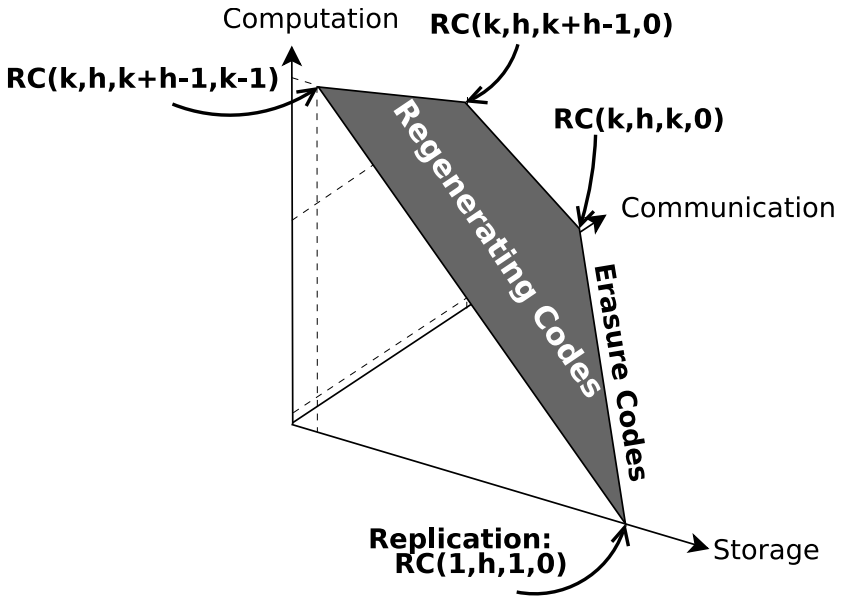


Figure A.10: Illustration du *trade-off* posé par le "Regenerating Codes".

A.5 Adaptive Proactive Repair Policy

Dans cette section, nous introduisons un nouveau paramètre permettant d'évaluer l'efficacité d'une politique de réparation : la régularité du taux de réparation. Nous proposons ensuite un système de réparation qui s'efforce à le maximiser.

Les approches existantes pour la réparation des blocs de parité sont soit **réactives** soit **proactives**. Les politiques réactives sont capables de suivre les changements dans le comportement de pairs, mais elles ont tendance à effectuer plus de réparations de ce qui est strictement nécessaire et de faire une utilisation irrégulière des ressources. Les politiques proactives utilisent un taux de réparation constant et sont en mesure de lisser la consommation de ressources pour les réparations. Cependant, pour fournir la durabilité, les systèmes proactifs

ont besoin d'une connaissance *a priori* du comportement de pairs. Dans le cas des connaissances imprécises ou erronées, la durabilité peut être mise en péril.

Nous soutenons que les systèmes proactifs et réactifs représentent deux cas spécifiques d'une approche plus générale qui accorde sa réactivité avec la stabilité prévue des pairs.

Notre principale contribution dans ce domaine est un schéma de réparation basée sur un processus continu d'estimation du comportement des pairs. Le taux R de réparation est périodiquement mis à jour en fonction des changements détectés. Cet approche est en mesure de fournir la durabilité de données, un bon niveau d'adaptation au comportement de pairs et un'utilisation régulière des ressources.

A.5.1 Un problème de contrôle adaptatif

Une politique proactive non adaptative utilise un taux de réparation R qui est constant dans le temps. Dans notre solution, nous cherchons à construire un *système de contrôle adaptatif* qui est capable d'adapter le taux de réparation aux changements de comportement des pairs.

Nous représentons notre schéma de contrôle adaptatif dans la Fig. A.11. Trois composantes sont définies :

- **Le système** (*System*) représente l'évolution de la situation des pairs qui stocke un objet, en termes de déconnexions temporaires, déconnexions permanentes, reconnexions et réparations. Il est caractérisée par un modèle régi par trois paramètres : le taux de déconnexion μ, le taux de reconnexion λ, et la probabilité de décès P_{death}. Le paramètre d'entrée du système est le taux de réparation, qui est un signal $R(t)$ qui évolue dans le temps, déterminé par le contrôleur. Les paramètres de sortie du système sont le nombre de pairs en ligne $n(t)$ et une information supplémentaire, appelée **transition**, qui signale l'événement d'une réparation ou d'une reconnexion.
- **L'estimateur** (*Estimator*) estime les paramètres qui caractérisent le système. Les valeurs de sortie de l'estimateur sont l'estimation $\hat{\mu}$ du taux de déconnexion et la probabilité de mort $\hat{P}_{death}$.
- **Le contrôleur**(*Controller*) reçoit les estimations ($\hat{\mu}$ et $\hat{P}_{death}$) de l'estimateur et le nombre ciblé n' des pairs en ligne sélectionné par l'administrateur système. L'output du régulateur est le taux de réparation $R(t)$ nécessaire pour obtenir l'objectif n'.

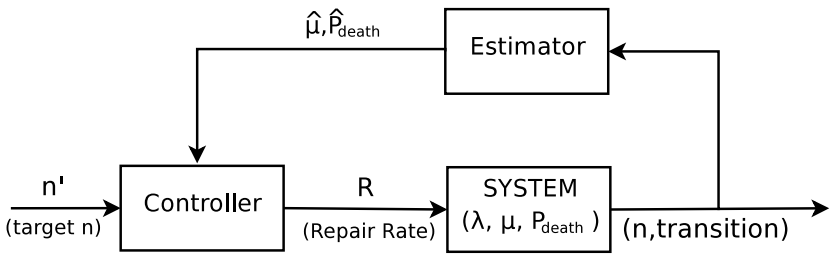


Figure A.11: Le schéma de contrôle adaptatif

Les opérations d'estimation et de contrôle sont effectuées périodiquement à un taux de $1/\Delta t$, où Δt est la période d'observation utilisée par l'estimateur et la période de mise à jour, c'est-à-dire l'intervalle entre deux mises à jour de $R(t)$.

Le modèle du système

La conception de l'estimateur exige la définition d'un modèle mathématique du système qui enregistre le comportement des pairs.

Le modèle que utilisons est un réseau de files d'attente, qui consiste en deux files d'attente $G/G/\infty$, comme illustré dans la Fig. A.12. Les files d'attente $G/G/\infty$ représentent des éléments retard, qui correspond au temps de service et suit une distribution générique. Dans notre cas, Q_1 représente les pairs qui sont connectés et qui se déconnectent avec un taux μ; Q_2 représente les pairs temporairement déconnectés qui se reconnectent avec un taux λ.

Nous supposons que le nombre de pairs est suffisamment large pour que chaque bloc de parité puisse être stocké sur un pair différent. Avec cette hypothèse, ce modèle représente aussi la disponibilité des blocs de parité dans le système.

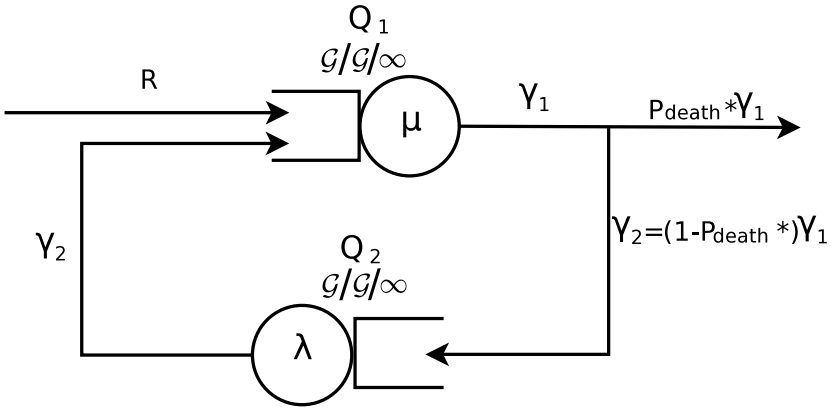


Figure A.12: Réseau de files d'attente modélisant le stockage d'un objet.

Les clients de la première file d'attente Q_1 représentent le *nombre de blocs de parité disponibles* $n(t)$. Le flux d'arrivée, dont le taux est notée avec γ_1, est donné par (i) les blocs de parité qui sont réparés avec un taux R, et (ii) le flux de blocs de parité qui reviennent de nouveau en ligne après une période d'indisponibilité. Le temps passé en Q_1 est déterminé par le taux de service μ. Le flux de départ de Q_1 représente les blocs de parité qui se déconnectent avec un taux égal au taux d'arrivée.

Les clients dans la deuxième file Q_2 représentent le *nombre de blocs de parité temporairement indisponibles* $m(t)$. Le flux d'arrivée, dont le taux est notée avec γ_2, est donnée par les blocs de parité qui sont devenus indisponibles et n'ont pas abandonné définitivement le système, qui est : $\gamma_2 = (1 - P_{death})\gamma_1$. Le temps passé en Q_2 est déterminé par le taux de service λ. Enfin, le flux de départ de Q_2 représente les blocs de parité qui deviennent de nouveau disponibles.

Pour résoudre ce réseau de files d'attente, on écrit ses équations d'équilibre :

$$\begin{array}{rcl} \gamma_1 &=& R+\gamma_2 \\ \gamma_2 &=& (1-P_{death})\gamma_1 \end{array} \Rightarrow \left\{ \begin{array}{rcl} \gamma_1 &=& \frac{1}{P_{death}} R \\ \gamma_2 &=& \frac{1-P_{death}}{P_{death}} R \end{array} \right. \tag{A.5}$$

En utilisant la loi de Little [99], nous pouvons calculer le nombre moyen de clients dans chaque file d'attente :

$$\overline{n} = \gamma_1/\mu = \frac{R}{\mu P_{death}} \qquad \overline{m} = \gamma_2/\lambda = \frac{(1-P_{death})R}{\lambda P_{death}} \tag{A.6}$$

L'estimateur

L'estimateur, en collectant des informations statistiques sur le signal $n(t)$ et sur les flux d'entrée dans la première file, est en mesure d'estimer les paramètres μ et P_{death}. On obtient :

- Le nombre moyen de blocs de parité disponibles $\hat{n}$:

$$\hat{n} = \frac{\sum_i n_i t_i}{\Delta T} \tag{A.7}$$

  où t_i est le temps passé par le système dans l'état n_i, ce qui implique $\Delta T = \sum_i t_i$.

- Le taux de déconnexion $\hat{\mu}$ en utilisant la relation :

$$\hat{\gamma_1} = \frac{\#Déconnections}{\Delta T}$$

  et la loi de Little :

$$\hat{\mu} = \frac{\hat{\gamma_1}}{\hat{n}} \tag{A.8}$$

- La probabilité de mort $\hat{P}_{death}$ peut être calculée de deux manières équivalentes. La première est :

$$\hat{P}_{death} = 1 - \frac{\#Reconnections}{\#Déconnections}$$

  la seconde est :

$$\hat{P}_{death} = \frac{R}{\hat{\gamma_1}} \tag{A.9}$$

A.5.2 Le contrôleur

Le contrôleur reçoit les estimations $\hat{\mu}$ et $\hat{P}_{death}$ et le nombre ciblé de blocs de parité disponibles n' et utilise eq. A.8 et eq. A.9 pour calculer la taux de réparation R :

$$R = \hat{\mu}\hat{P}_{death}n' \tag{A.10}$$

A.6 Conclusion

Pour conclure cette syntèse, nous rappelons nos contributions principales :

Regenerating Codes Les “Regenerating Codes”, dont la théorie a été proposée par Dimakis, sont une classe de codes correcteurs capables de réduire les coûts de communication dues à l’activité de réparation. Nous croyons que les “Regenerating Codes” sont très intéressants, mais qu’une analyse plus pratique est nécessaire pour bien comprendre leur propriétés. Nous proposons une implémentation des “Regenerating Codes” fondés sur les codes aléatoires linéaires et effectuons une évaluation analytique et expérimentale des coûts générés par ces codes.

Hierarchical Codes Nous proposons une classe de codes correcteurs, nommés “Hierarchical Codes”, qui visent, comme les “Regenerating Codes”, à réduire la communication nécessaire pour les réparations. Les “Hierarchical Codes” calculent les fragments de parité comme des combinaisons aléatoires linéaire d’un *sous-ensemble* des fragments d’origine. Ce choix permet de réduire la quantité de données téléchargées lors d’une réparation. Nous présentons le cadre théorique des “Hierarchical Codes” et nous proposons une évaluation analytique et expérimentale de leur efficacité.

Politique proactive adaptative de réparation Nous proposons une politique de réparation qui est en mesure de surmonter la dualité entre les politiques réactives et proactives. Notre système est fondé sur une estimation continue du comportement des pairs. Notre politique représente une solution flexible et ajustable qui permet au concepteur du système de sélectionner sa réactivité et d’augmenter la régularité du processus de réparation.

Bibliography

[1] Allmydata. http://www.allmydata.com/.

[2] Allmydata tahoe. http://www.allmydata.org/.

[3] Amazon simple storage service (s3). http://aws.amazon.com/s3/.

[4] Carbonite. http://www.carbonite.com.

[5] Cisco data center infrastructure 2.5 design guide. www.cisco.com/application/pdf/en/us/guest/netsol/ns107/c649/ccmigration_09186a008073377d.pdf.

[6] The farsite project. http://research.microsoft.com/en-us/projects/farsite/.

[7] Gnutella website. http://www.gnutella.com.

[8] Historical notes about the cost of hard drive storage space. http://www.alts.net/ns1625/winchest.html.

[9] ibackup. http://www.ibackup.com/.

[10] Idrive. http://www.idrive.com/.

[11] Microsoft skydrive. http://skydrive.live.com/.

[12] Mozy. mozy.com.

[13] The myth of the 100-year cd-rom. http://www.rense.com/general52/themythofthe100year.htm.

[14] The NaDa project. http://www.nanodatacenters.eu.

[15] The Napster website. http://www.napster.com.

[16] The OceanStore project. http://oceanstore.cs.berkeley.edu/.

[17] Ubistorage. http://www.ubistorage.com/.

[18] Wuala. www.wuala.com.

[19] S. Acedacnski, S. Deb, M. Medard, and R. Koetter. How good is random linear coding based distributed networked storage? In *Workshop on Network Coding, Theory, and Applications (NetCod)*, 2005.

[20] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer. Farsite: federated, available, and reliable storage for an incompletely trusted environment. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[21] R. Ahlswede, N. Cai, S.-Y. R. Li, and R. W. Yeung. Network information flow. *IEEE Transactions on Information Theory*, 46(4), 2000.

[22] S. Al-Kiswany, A. Gharaibeh, E. Santos-Neto, G. Yuan, and M. Ripeanu. StoreGPU: exploiting graphics processing units to accelerate distributed storage systems. In *ACM Symposium on High Performance Distributed Computing (HPDC)*, 2008.

[23] T. E. Anderson, M. D. Dahlin, J. M. Neefe, D. A. Patterson, D. S. Roselli, and R. Y. Wang. Serverless network file systems. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1995.

[24] G. Ateniese, R. Burns, R. Curtmola, J. Herring, L. Kissner, Z. Peterson, and D. Song. Provable data possession at untrusted stores. In *ACM Conference on Computer and Communications Security (CCS)*, 2007.

[25] M. Baker, M. Shah, D. S. H. Rosenthal, M. Roussopoulos, P. Maniatis, T. Giuli, and P. Bungale. A fresh look at the reliability of long-term digital storage. In *European Conference on Computer Systems (EuroSys)*, 2006.

[26] C. Batten, K. Barr, A. Saraf, and S. Trepetin. pStore: A secure peer-to-peer backup system. Technical Report MIT-LCS-TM-632, MIT Laboratory for Computer Science, 2001.

[27] R. Bhagwan, D. Moore, S. Savage, and G. M. Voelker. Replication strategies for highly available peer-to-peer storage. Technical Report CS2002-0726, University of California at San Diego, 2002.

[28] R. Bhagwan, S. Savage, and G. M. Voelker. Understanding availability. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2003.

[29] R. Bhagwan, K. Tati, Y.-C. Cheng, S. Savage, and G. M. Voelker. Total recall: system support for automated availability management. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.

[30] C. Blake and R. Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[31] W. J. Bolosky, J. R. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS*, 2000.

[32] M. Castro and B. Liskov. Practical byzantine fault tolerance. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 1999.

[33] A. L. Chervenak, V. Vellanki, and Z. Kurmas. Protecting file systems: A survey of backup techniques. In *Joint NASA and IEEE Mass Storage Conference*, 1998.

[34] B.-G. Chun, F. Dabek, A. Haeberlen, E. Sit, H. Weatherspoon, M. F. Kaashoek, J. Kubiatowicz, and R. Morris. Efficient replica maintenance for distributed storage systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[35] I. Clarke, S. Oskar, O. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[36] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2002.

[37] L. P. Cox and B. D. Noble. Samsara: honor among thieves in peer-to-peer storage. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[38] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with cfs. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2001.

[39] A. Dandoush, S. Alouf, and P. Nain. P2P storage systems modeling, analysis and evaluation. Research Report RR-6392, INRIA, 2007.

[40] A. Dandoush, S. Alouf, and P. Nain. Simulation analysis of download and recovery processes in p2p storage systems. Research Report RR-6858, INRIA, 2009.

[41] A. Datta and K. Aberer. Internet-scale storage systems under churn – a study of the steady-state using markov models. In *IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2006.

[42] A. G. Dimakis, B. Godfrey, Y. Wu, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. *Computer Research Repository (CoRR)*, arXiv:0803.0632v1 http://arxiv.org/abs/0803.0632, 2008.

[43] A. G. Dimakis, P. B. Godfrey, M. J. Wainwright, and K. Ramchandran. Network coding for distributed storage systems. In *IEEE INFOCOM*, 2007.

[44] R. Dingledine, M. J. Freedman, and D. Molnar. The free haven project: Distributed anonymous storage service. In *Workshop on Design Issues in Anonymity and Unobservability*, 2000.

[45] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2002.

[46] J. R. Douceur and W. J. Bolosky. A large-scale study of file-system contents. In *ACM SIGMETRICS*, 1999.

[47] J. R. Douceur and R. Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *Symposium on DIStributed Computing (DISC)*, 2001.

[48] J. R. Douceur and R. Wattenhofer. Modeling replica placement in a distributed file system: Narrowing the gap between analysis and simulation. In *European Symposium on Algorithms (ESA)*, 2001.

[49] J. R. Douceur and R. Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2001.

[50] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2001.

[51] A. Duminuco, E. Biersack, and T. En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 1–12, Dec. 2007.

[52] C. Fragouli, J.-Y. L. Boudec, and J. Widmer. Network coding: An instant primer. *ACM Computer Communication Review (CCR)*, 36(1), 2006.

[53] R. Gallager. Low density parity check codes. *IRE Transactions of Information Theory*, 8, Jan 1962.

[54] A. J. Ganesh, A.-M. Kermarrec, and L. Massoulié. Peer-to-peer membership management for gossip-based protocols. *IEEE Transactions on Computers*, 52(2), 2003.

[55] S. Ghemawat, H. Gobioff, and S.-T. Leung. The google file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2003.

[56] B. Godfrey. Repository of availability traces. `http://www.cs.berkeley.edu/~pbg/availability/`, 2006.

[57] P. Golle, K. Leyton-Brown, I. Mironov, and M. Lillibridge. Incentives for sharing in peer-to-peer networks. In *Workshop on ELectronic COMmerce (WELCOM)*, 2001.

[58] A. Haeberlen, A. Mislove, and P. Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2005.

[59] R. Hasan, W. Yurcik, and S. Myagmar. The evolution of storage service providers: Techniques and challenges to outsourcing storage. In *ACM Workshop on Storage Security and Survivability (StorageSS)*, 2005.

[60] T. Ho, M. Médard, R. Koetter, D. R. Karger, M. Effros, J. Shi, and B. Leong. A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10), 2006.

[61] J. Ioannidis, S. Ioannidis, A. D. Keromytis, and V. Prevelakis. Fileteller: Paying and getting paid for file storage. In *International Financial Cryptography Conference*, 2002.

[62] F. Junqueira, R. Bhagwan, K. Marzullo, S. Savage, and G. M. Voelker. The phoenix recovery system: rebuilding from the ashes of an internet catastrophe. In *USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2003.

[63] R. Koetter and M. Médard. An algebraic approach to network coding. *IEEE/ACM Transactions On Networking (TON)*, 11(5), 2003.

[64] J. Kubiatowicz, D. Bindel, Y. chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[65] G. Lefebvre and M. J. Feeley. Separating durability and availability in self-managed storage. In *ACM SIGOPS European Workshop (EW11)*, 2004.

[66] S.-Y. R. Li, R. W. Yeung, and N. Cai. Linear network coding. *IEEE Transactions on Information Theory*, 49(2), 2003.

[67] M. Lillibridge, S. Elnikety, A. Birrell, M. Burrows, and M. Isard. A cooperative internet backup scheme. In *USENIX Annual Technical Conference (ATEC)*, 2003.

[68] S. Lin and C. Jin. RepStore: A self-managing and self-tuning storage backend with smart bricks. In *IEEE International Conference on Autonomic Computing (ICAC)*, 2004.

[69] M. Luby. LT codes. In *IEEE Symposium on FOundations of Computer Science (FOCS)*, 2002.

[70] M. G. Luby, M. Mitzenmacher, M. A. Shokrollahi, D. A. Spielman, and V. Stemann. Practical loss-resilient codes. In *ACM Symposium on Theory Of Computing (STOC)*, 1997.

[71] P. Lyman, H. R. Varian, P. Charles, N. Good, L. L. Jordan, and J. Pal. How much information? 2003. http://www2.sims.berkeley.edu/research/projects/how-much-info-2003/, 2003.

[72] P. Maymounkov and D. Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[73] M. Mitzenmacher. Digital fountains: a survey and look forward. In *Information Theory Workshop*, 2004.

[74] Moritz Steiner, Taoufik En Najjary, and Ernst W Biersack. Analyzing peer behavior in KAD. Technical Report 2358, EURECOM, Sophia Antipolis, France, 2007.

[75] S. Nath, H. Yu, P. B. Gibbons, and S. Seshan. Subtleties in tolerating correlated failures in wide-area storage systems. In *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2006.

[76] N. Oualha, M. Önen, and Y. Roudier. A security protocol for self-organizing data storage. In *International Information Security Conference (IFIPSEC)*, 2008.

[77] N. Oualha and Y. Roudier. Reputation and audits for self-organizing storage. In *Workshop on Security in Opportunistic and SOCial Networks (SOSOC)*, 2008.

[78] N. Oualha and Y. Roudier. Validating peer-to-peer storage audits with evolutionary game theory. In *International Workshop on Self-Organizing Systems (IWSOS)*, 2008.

[79] D. A. Patterson, G. Gibson, and R. H. Katz. A case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM International Conference on Management of Data (SIGMOD)*, 1988.

[80] J. S. Plank. A tutorial on Reed-Solomon coding for fault-tolerance in RAID-like systems. *Software Practice and Experience*, 27(9), 1997.

[81] S. Quinlan and S. Dorward. Venti: A new approach to archival data storage. In *USENIX Conference on File And Storage Technologies (FAST)*, 2002.

[82] S. Ramabhadran and J. Pasquale. Analysis of long-running replicated systems. In *IEEE INFOCOM*, 2006.

[83] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *ACM SIGCOMM*, 2001.

[84] I. S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2), 1960.

[85] S. Rhea et al. OpenDHT: A public DHT service and its uses. In *ACM SIGCOMM*, 2005.

[86] R. Rodrigues and B. Liskov. High availability in dhts: Erasure coding vs. replication. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[87] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms*, 2001.

[88] R. Sandberg, D. Golgberg, S. Kleiman, D. Walsh, and B. Lyon. *Design and implementation of the Sun network filesystem*, pages 379–390. Innovations in Internetworking. Artech House, Inc., Norwood, MA, USA, 1988.

[89] T. S. J. Schwarz and E. L. Miller. Store, forget, and check: Using algebraic signatures to check remotely administered storage. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2006.

[90] E. Sit, A. Haeberlen, F. Dabek, B.-G. Chun, H. Weatherspoon, R. Morris, M. F. Kaashoek, and J. Kubiatowicz. Proactive replication for data durability. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2006.

[91] M. Steiner. Kad traces. http://www.eurecom.fr/~btroup/kadtraces/, 2007.

[92] M. Steiner, E. W. Biersack, and T. En-Najjary. Actively Monitoring Peers in Kad. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[93] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, 2001.

[94] C. A. Thekkath, T. Mann, and E. K. Lee. Frangipani: a scalable distributed file system. In *ACM Symposium on Operating Systems Principles (SOSP)*, 1997.

[95] J. Tian and Y. Dai. Understanding the dynamic of peer-to-peer systems. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2007.

[96] J. Tian, Z. Yang, and Y. Dai. A data placement scheme with time-related model for p2p storages. In *IEEE International Conference on Peer-to-Peer Computing (P2P)*, 2007.

[97] L. Toka and P. Michiardi. Analysis of user-driven peer selection in peer-to-peer backup and storage systems. In *ACM-Valuetools International Workshop on Game theory in Communication Networks (GameComm)*, 2008.

[98] L. Toka and P. Michiardi. Uncoordinated peer selection in P2P backup and storage applications. In *IEEE Global Internet Symposium*, 2009.

[99] K. S. Trivedi. *Probability and statistics with reliability, queuing, and computer science applications*. John Wiley & Sons, 2nd edition, 2001.

[100] G. Utard and A. Vernois. Data durability in peer to peer storage systems. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID)*, 2004.

[101] H. Weatherspoon. *Design and evaluation of distributed wide-area on-line archival storage systems*. PhD thesis, University of California at Berkeley, 2006.

[102] H. Weatherspoon, B.-G. Chun, C. W. So, and J. Kubiatowicz. Long-term data maintenance in wide-area storage systems: A quantitative approach. Technical Report UCB/CSD-05-1404, 2005.

[103] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[104] H. Weatherspoon, T. Moscovitz, and J. Kubiatowicz. Introspective failure analysis: Avoiding correlated failures in peer-to-peer systems. In *IEEE Symposium on Reliable Distributed Systems (SRDS)*, 2002.

[105] H. Weatherspoon, C. Wells, P. R. Eaton, B. Y. Zhao, and J. D. Kubiatowicz. Silverback: A global-scale archival system. Technical Report CSD-01-1139, University of California at Berkeley, 2001.

[106] H. Weatherspoon, C. Wells, and J. Kubiatowicz. Naming and integrity: Self-verifying data in peer-to-peer systems. In *Future Directions in Distributed Computing*, 2003.

[107] F. Wu, T. Qiu, Y. Chen, and G. Chen. Redundancy schemes for high availability in DHTs. In *International Symposium on Parallel and distributed processing and Applications (ISPA)*, 2005.

[108] Y. Wu, A. Dimakis, and K. Ramchandran. Deterministic regenerating codes for distributed storage. In *Annual Allerton Conference*, 2007.

[109] Z. Zhang, Q. Lian, and Y. Chen. XRing a robust and high-performance P2P DHT. Technical Report MSR-TR-2004-93, Microsoft Research, 2004.

[110] Z. Zhang, Q. Lian, S. Lin, W. Chen, Y. Chen, and C. Jin. BitVault: a highly reliable distributed data retention platform. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2007.

[111] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: a resilient global-scale overlay for service deployment. *IEEE Journal on Selected Areas in Communications*, 22(1), 2004.

List of Publications

[1] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, pages 1–12, December 2007.

[2] Alessandro Duminuco and Ernst Biersack. Hierarchical Codes: How to make erasure codes attractive for peer-to-peer storage systems. In *IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 89–98, September 2008.

[3] Alessandro Duminuco and Ernst Biersack. Hierarchical Codes: A flexible trade-off for erasure codes in peer-to-peer storage systems. *Journal of Peer-to-Peer Networks and Applications*, vol. 2, April 2009.

[4] Alessandro Duminuco and Ernst Biersack. A practical study of regenerating codes for peer-to-peer backup systems. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 376–384, June 2009.