ANNÉE 2013

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention : Informatique

Ecole doctorale MATISSE

présentée par

Alexandre van Kempen

préparée à l'unité de recherche INRIA
INRIA Rennes - Bretagne Atlantique
Université Rennes 1

Optimiser l'utilisation de la bande passante dans les systèmes de stockage distribué

Thèse soutenue à Rennes le 8 Mars 2013

devant le jury composé de :

Abstract

Modern storage systems have to face the surge of the amount of data to handle. In essence, they have to ensure a reliable storage of all their users' data, while providing a fast retrieval. At the current scale, it would be an illusion to believe that a single centralized storage device is able to store and retrieve all its users' data. While from the user's viewpoint the storage system remains a single interlocutor, its underlying architecture has become necessarily distributed. In others words, storage is no longer assigned to a centralized storage equipment, but is now distributed between multiple independent storage devices, connected via a network.

Therefore, when designing networked storage systems, bandwidth should now be taken into account as a critical resource. In fact, the bandwidth of a system is intrinsically a limited resource which should be handled with care to avoid congestion. In a distributed storage system, the system bandwidth is mostly consumed by (i) the data backup and restore operations, and (ii) the system maintenance.

(i) During backup and restore operations, data is exchanged via the network between users and the system. These exchanges consume a large amount of bandwidth, proportional to the size of data.

(ii) A storage system inevitably experiences either transient or permanent failures, typically disk crashes, involving some data loss. Lost data must be restored so that the system remains in a healthy state. These maintenance operations lead to multiple data transfers to repair the loss incurred by failures. While being essential to maintain the system reliability, repairs remain costly in terms of bandwidth consumption.

The problem of efficient bandwidth utilization becomes even more critical when designing systems relying on peer-to-peer architectures. In a peer-to-peer storage system, storage resources are provided by the users. Each of them share parts of their local storage capacity. The system aggregates this *eldorado* of space in order to offer a reliable storage service to its participants. While peer-to-peer architectures offer appealing properties such as scalability, and fault-tolerance, they also suffer from important limitations regarding the bandwidth. In fact, the available bandwidth of the system is dramatically reduced due to *churn* and failures.

In a peer-to-peer system, users can connect and disconnect from the system at will, without any warning. This turnover, usually referred to as churn, involves asynchrony between users' uptime period. Intuitively, data can be exchanged directly between two users only if both are connected at the same time. As a result, the effective amount of data they can transmit is reduced by churn. Churn thus leads to a significant reduction of the available bandwidth between users, increasing the time to backup and restore data. In addition, in a peer-to-peer system, the whole available bandwidth of the system is shared between the "useful data" transmission *i.e.*, backup and restore operations, and the data transfers related to the system maintenance. Maintenance operations consume an important part of the system bandwidth. Consequently, these maintenance operations decrease the amount of available bandwidth for backups and restores, underlining the importance of minimizing the impact of maintenance on the system bandwidth.

Contributions

The focus of this thesis is to **optimize the available bandwidth of distributed storage systems**, lowering the impact of churn and failures. The objective is twofold, on the one hand the purpose is to increase the available bandwidth for data exchanges and on the other hand, to decrease the amount of bandwidth consumed by maintenance. While addressing bandwidth consumption might be achieved in many ways, we focus on two specific parts:

- In the first part of this manuscript, we address the reduction of the impact of asynchrony, resulting from churn, on the available bandwidth during backup and restore operations. The first contribution of this thesis presents an hybrid peer-to-peer architecture taking into account the low level topology of the network *i.e.*, the presence of gateways between the system and the users. In this gateway-assisted system, the gateways are turned into active components, acting as a buffering layer to compensate the intrinsic instability of the machines. Due to their high availability, gateways provide a stable *rendez-vous* point between users' uptime, thus masking the asynchrony between them. This architecture is evaluated by simulation using real world traces for availability. Results show that the time required to backup and restore data is greatly reduced due to a smoother use of the available bandwidth.

- In the second part, we concentrate on the reduction of the impact of maintenance operations on the system bandwidth. Our contribution is twofold.

  The second contribution of this thesis acts at the upstream level of the maintenance, during the failure detection process. In peer-to-peer systems, distinguishing permanent failures that require a repair from transient disconnections is challenging. In the latter case, a repair may turn out to be useless, consuming bandwidth unnecessarily. In order to assess whether a repair is required or not, an adaptive and user-level timeout mechanism is proposed. This mechanism, based on a Bayesian approach, is evaluated by simulation on real availability traces. Results show that, compared to classical timeouts, the number of useless repairs is significantly decreased, thus cutting back on the unnecessary consumed bandwidth.

The third contribution describes a repair protocol especially designed for erasure-coded stored data. Erasure codes are a promising way to replace classical replication in providing redundancy, for their savings in storage space. Repairing replicated data is straightforward as it only requires the transfer of a replica. However, it is well-known that repairing erasure-coded data is extremely bandwidth consuming. Previous works trying to reduce the repair bandwidth usually do so at the file level. The proposed repair mechanism enables to repair multiple files simultaneously, thus factorizing the bandwidth costs. The protocol is implemented and deployed on a public experimental testbed to demonstrate the savings in a real environment. Compared to most implemented mechanisms, results reveal that the necessary bandwidth is halved, while repair times are considerably lowered.

Organization

The remainder of this thesis is organized as follows:

Chapter 1 covers general concepts about storage systems, and provides definitions used in the following chapters of the manuscript. Chapters 2 and 3 present the three contributions which form the core of this work. Chapter 2 describes the gateway-assisted architecture, and presents its simulation results. Chapter 3 is devoted to the maintenance bandwidth reduction. The first part of Chapter 3 presents the timeout mechanism as well as its evaluation. The repair protocol and its implementation are described in the second part of this chapter. Finally, Chapter 4 discusses directions for future research.

Publications

(1) **Availability-based methods for distributed storage systems**. Anne-Marie Kermarrec, Erwan Le Merrer, Gilles Straub, Alexandre Van Kempen. *In SRDS 2012 : Proceedings of the International Symposium on Reliable Distributed Systems, 2012.*

(2) **Regenerating Codes: A System Perspective**. Steve Jiekak, Anne-Marie Kermarrec, Nicolas Le Scouarnec, Gilles Straub, Alexandre Van Kempen. *In DISCCO 2012 : Proceedings of the International workshop on Dependability Issues in Cloud Computing, 2012.*

(3) **On The Impact of Users Availability In OSNs**. Antoine Boutet, Anne-Marie Kermarrec, Erwan Le Merrer, Alexandre Van Kempen. *In SNS 2012 : Proceedings of the International workshop on Social Network Systems, 2012.*

(4) **Efficient peer-to-peer backup services through buffering at the edge**. Serge Defrance, Anne-Marie Kermarrec, Erwan Le Merrer, Nicolas Le Scouarnec, Gilles Straub, Alexandre Van Kempen. *In P2P 2011 : Proceedings of the International Conference on Peer-to-Peer Computing, 2011.*

(5) **Clustered Network Coding for maintenance in practical storage systems**. Anne-Marie Kermarrec, Erwan Le Merrer, Gilles Straub, Alexandre Van Kempen. *Technical report available on arXiv (under sumbmission), 2012.*

Contents

List of Figures

List of Tables

CHAPTER 1

Preliminaries on storage systems

In this chapter, we review the basics of a storage system. We define the notion of data availability/durability and the concept of data redundancy, data placement and data maintenance. We provide some insights about their respective implementation, while underlining their impact on the bandwidth consumption. Finally we describe the commonly deployed architectures for storage systems.

A study sponsored by the information storage company EMC [2] estimates that the world's data is more than doubling every two years, reaching 1.8 zettabytes[1] of data stored in 2011. Online storage systems that ensure the durability, integrity, and accessibility of digital data become increasingly important. However, designing a complete storage system is a challenging task, and requires the study of numerous tradeoffs depending on the targeted application.

In this thesis we focus on the lower layer of a storage system, below the application level. In other words, we only deal with how data is stored, and not how it is used by a potential application sitting on top. First, we define the basic storage services expected by a user. Second, we describe various concepts related to the design of such a system from a system viewpoint.

1.1 User's viewpoint

From the user's viewpoint, the storage system is seen as a black box and all the implementation and architectural aspects are ignored. The two basic primitives between the user and the system are only *"store data"* and to *"retrieve data"*. Typically, a user inserts data she wants the system to keep safe, while being able to retrieve it

[1]1 zettabyte = 10^{21} bytes

when needed. To achieve this, (i) data has to be effectively stored by the system, (ii) data can be retrieved at the time of a given request.

(i) The first point relates to the notion of *durability*. When a user inserts data into a storage system, she expects this data never to be lost. This means that eventually, he will be able to retrieve it. This is the main function a storage system has to ensure.

(ii) The notion of durability is different from the one of *availability*. Availability only relates to the ability of retrieving the data upon the user's request at a given point in time. Data might be durably stored, while not being available when requested.

There are numerous reasons for not being able to store or retrieve data upon a request. Namely, the failure of servers, the congestion of a network link, the crash of a storage device, or a power outage are common issues.

1.2 System viewpoint

Distributed storage systems are designed to provide a reliable storage service over unreliable components [22, 37, 49, 75]. This unreliability involves challenging tradeoffs when designing a storage system.

1.2.1 Failures

To start with, one has to accept the fact that the failures of the underlying components of a storage system are unavoidable [34, 83]. In addition, the more components in a system, the more failures are to be expected.

A storage system would be 100% safe only if composed of storage devices which never fail with certainty. Such devices are not available yet (to the best of our knowledge) so we have to deal with the fact that the probability of losing data exists. In addition, the more data is kept, and the longer it is kept, the greater the chance that some of it will be unrecoverable when required [73].

However, the good news is that we do have good insights on how to make this probability arbitrarily low. The first idea to tolerate a failure is to replicate. Intuitively we can provide some general facts such as:

- the more copies, the safer;
- the more independent copies, the safer;
- the more frequently audited and replaced if failed, the safer.

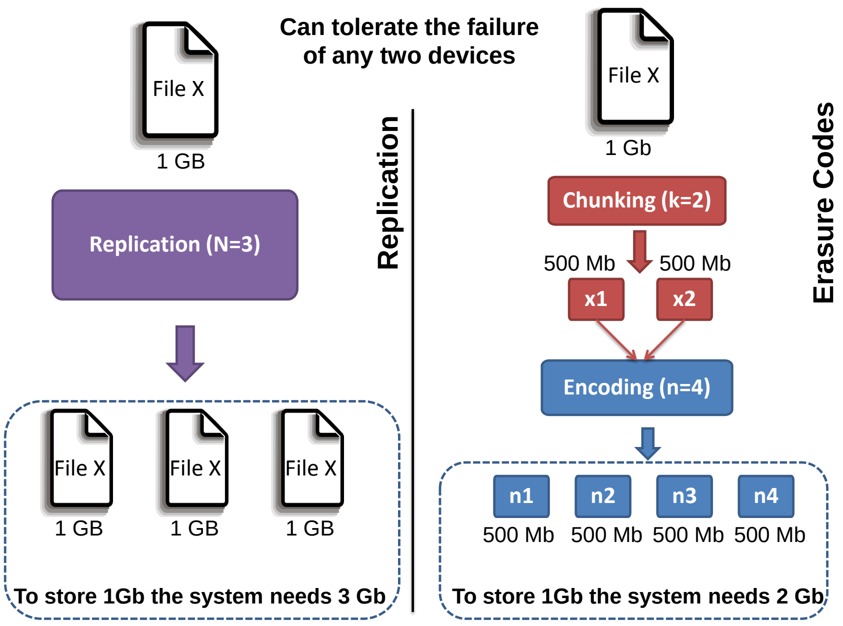


Figure 1.1: Redundancy creation process. On the left hand side classical replication for N=3, on the right hand side erasure codes (k=2,n=4). Both can tolerate the failure of any set of two storage devices.

At the system level, this is respectively related to data redundancy, data placement, and data maintenance. These three fundamental functions, as well as examples of their implementation, are discussed in the following three sections.

1.2.2 Data redundancy

A classical way to provide a system fault tolerance is to make its underlying components redundant. And as one can expect, the more redundancy is added, the more reliability can be obtained. *Replication*, *i.e.*, whole copy of data, is the canonical solution for data fault tolerance and is commonly used in current storage systems. While replication is the simplest means to obtain reliability with redundancy, it is now widely acknowledged that *erasure codes* can dramatically improve the storage efficiency. We describe hereafter these two redundancy strategies, while underlining the enhanced reliability of erasure codes.

Replication

Replication is the most intuitive way to introduce redundancy, as it merely requires copying the data. Typically N instances of the same data are needed to tolerate $N-1$ simultaneous failures. While plain replication is easily implemented, and has been proposed in numerous designs so far [20, 21, 37, 49, 50, 55, 75], it suffers from a high *storage overhead*. The storage overhead is defined as the total quantity of data including redundancy over the original quantity of data to store. For example, to store a 1 Gigabyte (GB) file, the system requires three times that amount so as to cope with two failures. This storage overhead ($N = 3$ in the previous example) is the price to pay to tolerate failures. However this high overhead is a growing concern, especially as the scale of storage systems keeps increasing.

Erasure codes

Erasure codes have been widely acknowledged as much more efficient than replication [84] with respect to storage overhead. They have been the focus of many studies, and numerous codes representing different points in the code design space have been proposed. In this thesis we only focus on optimal codes, as they provide the best reliability for a given storage overhead. More specifically, we describe the principles of Maximum Distance Separable (MDS) codes known as being optimal. The basics of an MDS code (n, k) are: a file to store is split into k chunks, encoded into n blocks with the property that any subset of k out of n blocks is sufficient to reconstruct the file. Thus, to reconstruct a file of $\mathcal{M}$ Bytes one needs to download *exactly* $\mathcal{M}$ Bytes. This corresponds to the same amount of data as if plain replication were used. Reed-Solomon codes are a classical example of MDS codes and are already deployed in existing storage systems [17, 34].

The name "erasure codes" expresses the ability to sustain up to $n - k$ erasures (*i.e.*, failures) without losing any data. For example, for a given code ($k = 2$, $n = 4$), a file is split into $k = 2$ blocks, x_1 and x_2, then encoded into $n = 4$ blocks n_1, n_2, n_3 and n_4 with the property that any set of two encoded blocks is sufficient to recover the original file (See Figure 1.1). In this example, the system can thus tolerate any $n - k = 2$ failures before losing data. However, compared to replication, the storage overhead defined as the ration between n and k is only $\frac{n}{k} = 2$ with the code ($k = 2$, $n = 4$). This means that to tolerate two failures when storing a 1 GB file, replication needs 3 GB while the previous code ($k = 2$, $n = 4$) only requires 2 GB. Note that the replicas, as well as the encoded blocks, need to be stored on distinct storage components.

In order to better understand this tradeoff between the storage overhead, and the reliability we describe a commonly used model to compare the efficiency of replication and erasure codes.

Reliability model

In this section, we describe a commonly used model [56, 84] to estimate the reliability one can expect depending on the amount of redundancy introduced. This straightforward model enables to compare the probability to lose data depending on the storage overhead for replication and erasure codes. In essence, this model only provides an

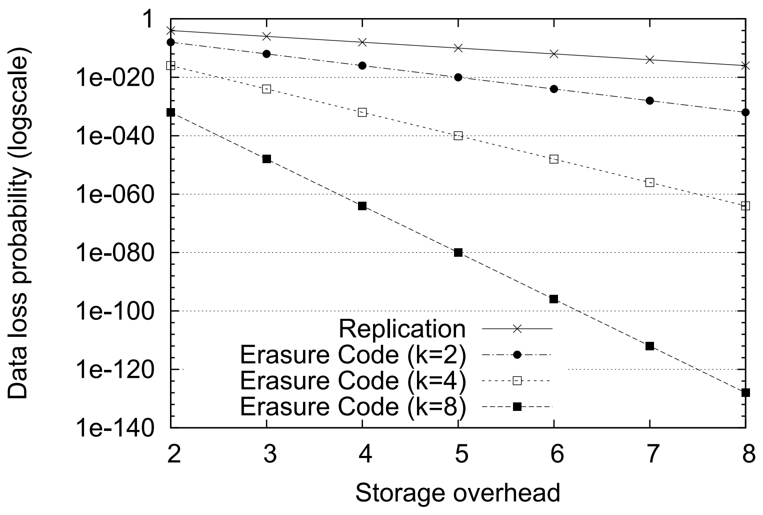


Figure 1.2: Probability to lose data depending on the storage overhead introduced for replication and erasure codes. The probability to fail for each device is $p_f = 0.01$

estimation of the obtained reliability, but it is realistic enough to point out the reasons why erasure codes are more efficient than replication.

Let p_f be the probability that a given storage component fails. In case of replication, it is easy to see that data are lost if and only if the N replicas fail at the same time. If we assume, to a first approximation, that the N replicas have the same probability to fail p_f, and that these failures are independent then we can assert that the probability to lose data p_{loss} is:

$$P_{loss,replication} = \prod_{i=1}^{N} p_f = (p_f)^N$$

As erasure codes can tolerate up to $n - k$ failures, a data loss thus requires the simultaneous failures of at least, *any* set of $n-k+1$ storage components. Assuming, as previously, a homogeneous and independent failure probability p_f, then the probability to lose data p_{loss} can be expressed as (binomial law):

$$P_{loss,erasurecodes} = \sum_{i=n-k+1}^{n} \binom{n}{i} p_f^i (1-p_f)^{n-i}$$

In Figure 1.2 we plot the probability to lose data depending on the storage overhead introduced when using replication or erasure codes. The data loss probability is dramatically reduced when using erasures codes compared to replication (the y-axis is in logarithmic scale). In addition, for the same storage overhead, the larger k, the smaller the probability to lose data. Intuitively, erasure codes spread the redundancy on many more devices than with replication. This combinatorial effect enhances the reliability as the loss of data thus involves multiple simultaneous failures. Note

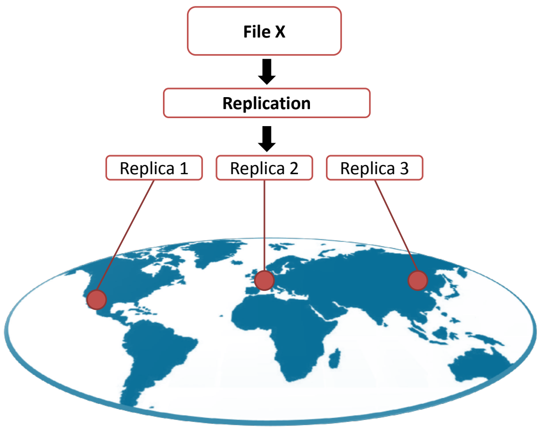


Figure 1.3: Geographically dispersed replicas enhances the reliability

that the computation of the data loss probability is much more complex in reality. For example, it is well known that failures in real systems are not independent and usually exhibit a high degree of correlation [34]. A thorough comparison between the reliability of replication and erasure codes can be found in [56, 84].

Summary

To summarize, for the same storage overhead, erasure codes provide better durability compared to replication. In other words, to ensure the same level of durability, erasure codes dramatically reduce the required storage space compared to replication. Finally, the choice of the type of redundancy is a designer decision. On the one hand, replication suffers from a high storage overhead but can easily be implemented while providing fast data accesses. On the other hand, erasures codes are more efficient but are complex to deploy due to coding and decoding operations [14, 56]. Some systems [49] propose to leverage the best of both. For instance, they combine erasure codes for durability and replication for latency reduction.

1.2.3 Data placement

Redundancy is not sufficient on its own to ensure fault tolerance. In fact, it is not worth making several copies of a file stored on the same disk as it will not enhance the reliability. The failure of this disk would automatically involve the failure of all the copies, and consequently the loss of the file. This underlines the interplay between the redundancy and the way it is spread in the system, namely the placement strategy.

Placement strategies usually aim to maximize a given metric, such as the durability of data. In this case, the challenge is to place data on storage devices which failures are as independent as possible. Typically, storage devices can be spread over geographically dispersed sites to ward off natural disasters [49]. Durability is not the only metric to be maximized, as placement strategy can also aim to enhance the data access latency, or data availability for example [26, 28].

Regardless of the metrics, the major concern shared by all placement strategies is to balance the storage load between all the available storage devices. A classical way to ensure this load balancing is to randomly assign data to storage devices. This random solution is straightforward to implement while providing close to evenly loaded devices and has thus been adopted by numerous systems [13, 14, 20].

1.2.4 Maintenance

Redundancy and its placement enable the system to carry on working even if the system experiences failures. However, this must be complemented with a maintenance mechanism capable of recovering from the loss of data. In fact, failures have to be repaired to sustain the redundancy policy. Otherwise, redundancy would only delay the time for data to be lost without preventing it. Maintenance has already been the focus of numerous storage system designs [13, 31, 39, 79] and rests upon two key functions. First it requires a monitoring layer able to accurately detect the failures. Second, the actual repair of data must be performed according to a given repair policy.

Detection

When designing a maintenance mechanism, the crucial challenge is to decide *when* to trigger a repair. In other words, how to distinguish permanent failures, requiring a repair, from temporary errors for which a repair may turn out to be useless. A clever detection mechanism minimizes the amount of network traffic sent in response to transient failures while maintaining reliability.

A practical and commonly implemented solution to decide on the nature of a failure is to use a *timeout* [13, 20]. Typically, once a node has not replied to a solicitation after a timeout has expired, it is considered as permanently failed. However, deciding on a timeout value is a tedious task. On the one hand, short timeouts accurately detect all the failures but generate a high number of false positives. On the other hand, longer timeouts decrease durability because the time to recognize permanent failures increases, thus augmenting the "window of vulnerability".

Authors in [20] show that *reintegration* of replicas is a key concept to lower the impact of false positives. The interest of reintegration is to be able to leverage the fact that nodes which have been wrongfully timedout are reintegrated in the system. For example, let assume that a given storage device has been incorrectly declared as faulty. A repair has thus been performed to sustain the redundancy factor while it turned out not to be necessary in the end. If reintegration is implemented, this means that the system is now one repair process ahead. Consequently, it can leverage this unnecessary repair to avoid triggering a new instance of the repair protocol when the next failure occurs.

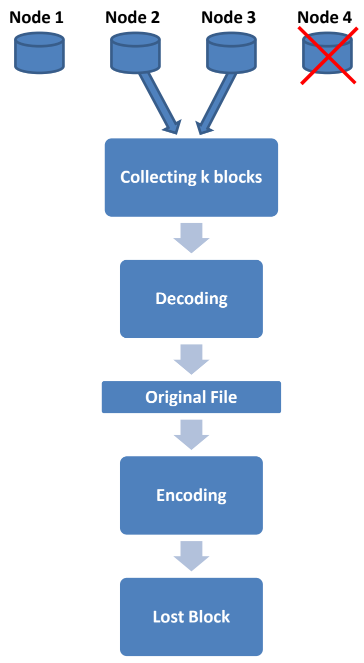


Figure 1.4: Classical repair process of a single block. (k=2, n=4)

Repair

The goal of repairs is to restore the level of fault tolerance by refreshing lost redundancy before data is lost due to permanent failures. When redundancy is implemented with plain replication, the repair of a lost replica only consists in the transfer of a new replica to be stored on a non-faulty device. The amount of consumed bandwidth is thus equals to the amount of lost data. However, repairing erasure coded data is much more complex. In fact, as pointed out in [72], one of the major concern of erasure codes lies in the maintenance process. It incurs an important overhead in terms of bandwidth utilization as well as in decoding operations as explained below.

Maintenance of Erasure Codes The repair process works as follows (see Figure 1.4): to repair one block of a given file, the new node first needs to download k blocks of this file (*i.e.*, corresponding to the size of the file) to be able to decode it. Once decoded, the new node can re-encode the file and regenerate the lost block. This must be iterated for all the lost blocks. Three issues arise:

1. Repairing one block (typically a small part of a file) requires the new node to download enough blocks (*i.e.* k) to reconstruct the entire file. This is required for all the blocks previously stored on the faulty node.

2. The new node must then decode the file, though it does not intend to access it. Decoding operations are known to be time consuming in particular for large files.

3. Reintegrating a node which has been wrongfully declared as faulty is almost useless. This is due to the fact that the new blocks created during the repair operation have to be strictly identical to the lost ones for this is necessary to sustain the coding strategy [2]. Therefore reintegrating a node results in having two identical copies of the involved blocks (the reintegrated ones and the new ones). Such blocks can only be useful if either the reintegrated node or the new node fails but not in the event of any other node failure.

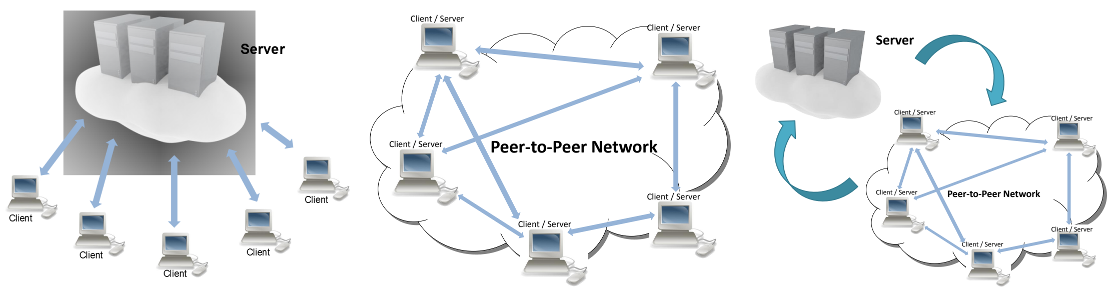
In order to minimize these drawbacks, various solutions have been suggested. Lazy repairs for instance as described in [13] consist in deliberately delaying the repairs, waiting for a successive amount of defects before repairing all the failures together. This enables to repair multiple failures with the bandwidth (*i.e.* data transferred) and decoding overhead needed for repairing one failure. However delaying repairs leaves the system more vulnerable in case of a burst of failures. Architectural solutions have also been proposed, as for example the *Hybrid strategy* [72]. This consists in maintaining one full replica stored on a single node in addition to multiple encoded blocks. This extra replica is used for repair avoiding the decoding operation. However maintaining an extra replica on a single node significantly complicates the design, while incurring scalability issues. Finally, new classes of codes have been designed [30, 45] which trade optimality in order to offer a better tradeoff between storage, reliability and maintenance efficiency.

1.2.5 The bandwidth bottleneck

The way of implementing redundancy, placement and maintenance policies has a strong impact on the system bandwidth consumption. For instance, while geographically-spread replication enhances the reliability, it also increases the inter-site bandwidth consumption. Alternately, the increased reliability provided by erasure codes comes at the price of a high bandwidth consumption during the maintenance process.

Bandwidth is a crucial element since data durability can only be ensured if there is sufficient bandwidth to repair lost data. In fact, an efficient maintenance mechanism may as well smooth the bandwidth consumption which is known to be one key aspect in storage systems [14, 79]. In addition, the scarcer the available bandwidth of the system, the more challenging becomes its management. Peer-to-peer storage systems are a typical example of such a case, as described in the next section.

[2]This can be achieved either by a tracker maintaining the global information about all blocks or by the new node inferring the exact structure of the lost blocks from all existing ones.



(a) Centralized (b) Peer-to-Peer (c) Hybrid

Figure 1.5: Architectures of storage systems.

1.3 Architecture

The architecture of a storage system describes, on the one hand, how the clients access to the storage service and on the other hand, how the underlying components of this storage service are organized. In other words, how these storage devices are interconnected and how they can communicate with each other. Centralized architecture are opposed to fully decentralized architecture, typically peer-to-peer networks (See Figure 1.5). We describe these two opposite paradigms in the the following sections.

1.3.1 Centralized architectures

The classical architecture of an online storage system follows the client-server paradigm, which consists of two distinct entities:

- The server, which provides the service and all the resources required for the service.
- The client, which makes use of the service and exploits the resources supplied by the server.

From the client viewpoint, the storage service is thus assigned to an infrastructure managed by an external entity. Majors actors such as Google [4], Amazon [1] or Microsoft [5] already propose storage services for free, or charged depending of the client storage needs. Note that, while the server appears as a single interlocutor to the clients, its underlying architecture is necessarily distributed. Typically storage devices are gathered into large *data-centers*. Network topologies of these data-centers are an active research area [7, 11, 62] and are outside the scope of this thesis.

While providing efficiency and simplicity from the user viewpoint, centralized architectures are known to suffer from two major issues, namely scalability and fault tolerance. Firstly, the service needs to be dimensioned to sustain all the possible clients and must be upgraded whenever the service demand increases. In other words, the more users to serve, the more resources have to be added. In addition, a centralized architecture represents a single point of failure. As *putting all eggs in one basket* is well known to be unsafe, the centralization of storage devices may involve reliability issues.

1.3.2 Peer-to-Peer

The peer-to-peer architecture of the system implies that the resources (storage, bandwidth, and computation) are provided by the users themselves, which collaborate to provide a reliable service. In a peer-to-peer system, all users, usually referred to as *peers*, play the role of both the client and the server. While being a customer of the service, every peer has to contribute, sharing part of its resources.

Because the resources are contributed by participant peers, a peer-to-peer system can scale almost arbitrarily, without requiring a "fork-lift upgrade" of existing infrastructure, for example, the replacement of a server with a more powerful one. In addition, as all resources are decentralized, the burden of communication is also spread between all peers, thus avoiding the bandwidth bottleneck of centralized servers.

Decentralization also provides resilience to faults and attacks, as peer-to-peer architectures are usually independent of dedicated infrastructure and centralized control, contrary to classical client-server systems. The decentralized organization of peer-to-peer systems, instead, exploits the independence of the different peers to provide a good level of reliability. This organization may be either *unstructured* or *structured* depending on the way peers are interconnected. A classical example of a structured peer-to-peer topology is to implement a *distributed hash table (DHT)* [70, 74, 78], the description of which is outside the scope of this thesis. A thorough description of peer-to-peer systems as well as their associated topologies can be found in [59, 71].

Peer-to-Peer storage system

Several studies [9] show that end user hard drives are on average half empty and represent a considerable amount of unused storage space available at the edge of the network. Indeed the authors in [9] reveal, after a five-year study, that the ratio of free space over the total file system capacity remained constant over the years. Aggregating this free space, represents therefore a real and scalable *Eldorado* of available space. peer-to-peer storage is a typical example of applications where this idle storage space could be leveraged [21, 49, 50, 52, 55, 75]. In such a peer-to-peer storage system, each node is in charge of storing the data of other nodes in exchange for having its own data stored and made available in a durable way.

While peer-to-peer alternatives can potentially offer *virtually unlimited storage* [15, 46], they are still not appealing enough performance-wise. It is acknowledged [14, 79] that those pure peer-to-peer architectures may fail to deliver reliable storage by only exploiting the resources of peers. Indeed, peer-to-peer storage systems are limited by the low to medium availabilities of participating peers and by the slow up-links of peers' network connections. This limits the amount of data that peers can transfer and places peer-to-peer systems way behind datacenter-based systems [52]. Not only this may impact the reliability of the stored content but also this does not provide a convenient system for users. For example, retrieval times to store or retrieve data can be an order of magnitude higher that the time required for direct download [66].

1.3.3 Hybrid

In order to move towards practical system deployment while still leveraging users' resources, hybrid architectures, where both servers and peers coexist, have been very recently proposed in various contexts [36]. They add some centralization for reliability or efficiency while still leveraging the users' resources at the edge of the network. For example, a server assisted peer-to-peer storage system is described in [52]. In their system, which can be referred to as *CDN-assisted*, the CDN enables to reduce the time needed to backup data, while the use of peers guarantees that the burden of storage and communication on the data center remains low. In this last approach, a peer uploads data to a set of other peers if they are available, and falls back on the datacenter otherwise, thus using the datacenter as a stable storage provider. However, a centralized component remains in the system thus incurring scalability issues. The first contribution of this thesis is precisely to decentralize the stable storage provider of the hybrid scheme. To this end, we propose an architectural solution described in the following chapter.

Chapter 2

Increasing the available bandwidth

In this chapter, we propose a gateway-assisted architecture for peer-to-peer storage systems, where residential gateways are turned into a stable buffering layer in between the peers and the Internet. Due to their high availability, gateways provide a stable rendez-vous point between users' uptime, thus masking the asynchrony between them. As a result, the amount of data they can exchange is greatly increased thus reducing the time for backup and restore operations. This work has been published in the proceedings of the IEEE International Conference on Peer-to-Peer Computing in 2011 (P2P 2011).

2.1 Introduction

Existing peer-to-peer approaches, either hybrid [36,52] of fully decentralized [21,49,50,55,75] usually do not take into account the low-level structure of the network. Indeed, most peer-to-peer applications ignore the presence of a gateway in between each peer and the Internet. As a result they do not leverage the presence of the gateway while it can greatly improve the overall performance of the system.

We believe that leveraging the gateway storage space may render peer-to-peer systems viable alternatives for storage. This should provide a reasonable solution even when peers experience a low availability as long as they connect frequently enough to the system. The residential gateways are ideal to act as stable buffers: they lay at the edge of the network between the home network and the Internet, and are highly available since they remain powered-on most of the time [82].

The remainder of this chapter is structured as follows. In Section 2.2, we remind the basics about gateways and then detail our architecture. Section 2.3 introduces a framework for comparison of our proposal to that of competitors, and Section 2.4

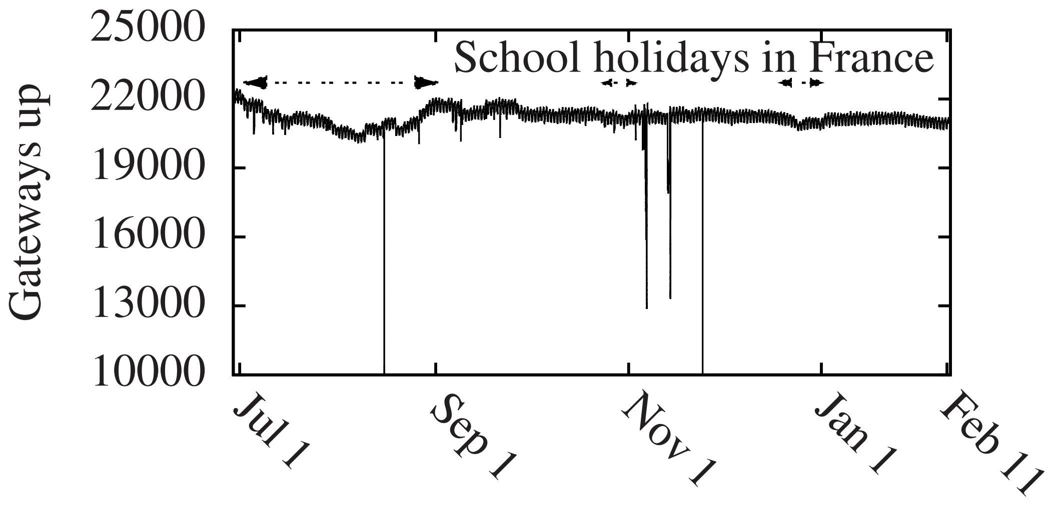


Figure 2.1: Availability of residential gateways mesured on a French ISP. The dataset has been acquired sending pings to a random sample of gateway IPs.

presents our evaluation study. We discuss respectively some specific points and related work in Section 2.5 and Section 2.6. Finally, we conclude this chapter in Section 2.6.

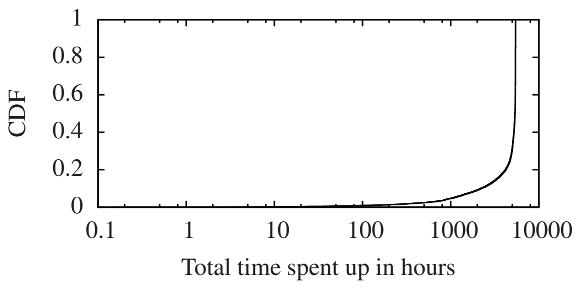
2.2 A gateway assisted system

Residential gateways connect home local area networks (LAN) to the Internet. They act as routers between their WAN interface (Cable or DSL) and their LAN interfaces (Ethernet and WiFi). They started to be deployed in homes to share Internet access among different devices and got additional functions as services (VoIP, IPTV) were delivered over the Internet. It is now fairly common to have home gateways embedding a hard drive, acting as Network Attached Storage to provide storage services to other home devices and offering some other ones to the outside world [33, 82].

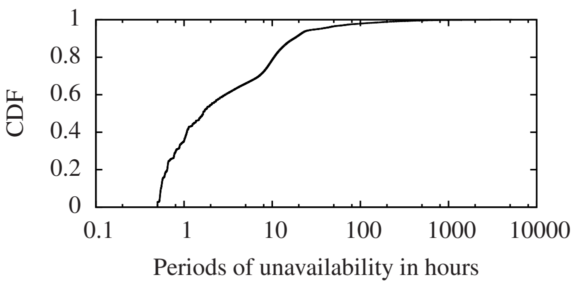
2.2.1 Stability of residential gateways

As residential gateways provide not only Internet connectivity, but also often VoIP, IPTV and other services to the home, the intuition tells us that they remain permanently powered on. To confirm this assumption, we extracted a trace of residential gateways of the French ISP Free, using active measurements[1]. We periodically ping-ed a set of IP addresses randomly chosen in the address range of this ISP, which has a static IP addressing scheme. We obtained the uptime patterns of 25, 000 gateways

[1]This trace and additional information can be found at the following URL `http://www.thlab.net/~lemerrere/trace_gateways/` . To the best of our knowledge, this is the first trace tracking availability of gateways.



(a) CDF of Total uptime



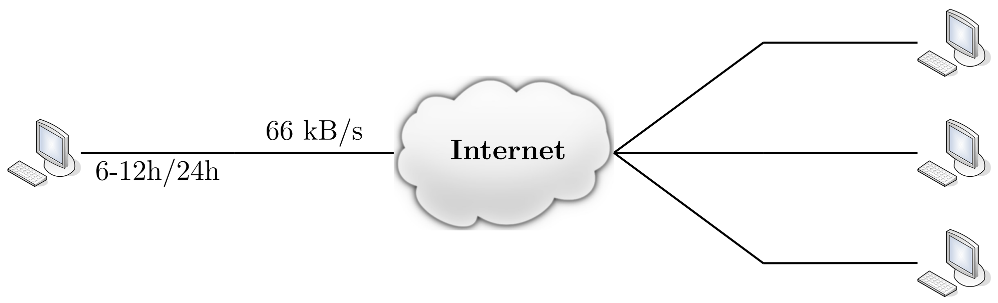
(b) CDF of Dowtime duration

for 7.5 months, covering week-patterns [12, 27], and holidays. We plot the availability of those devices against time, in the classical representation of availability, on Figure 2.2.1. Some clear acquisition artifacts appear due to both the unreliability of the ICMP monitoring and temporary failures on the path between our platform and the targeted network. Yet, the trace confirms the common intuition about the stability of those devices, in spite of a few users having power-off habits (on a daily or a holiday basis, see Figure 2.2b), thus slightly reducing the average availability. The average availability of gateways in this trace is 86%, which confirms the results observed in [82], where the authors used traces from a biased sample (only BitTorrent users) [25]. This has to be contrasted with the low to medium availabilities of peers generally recorded in the literature, as *e.g.* 27% in [52], or 50% in [41].

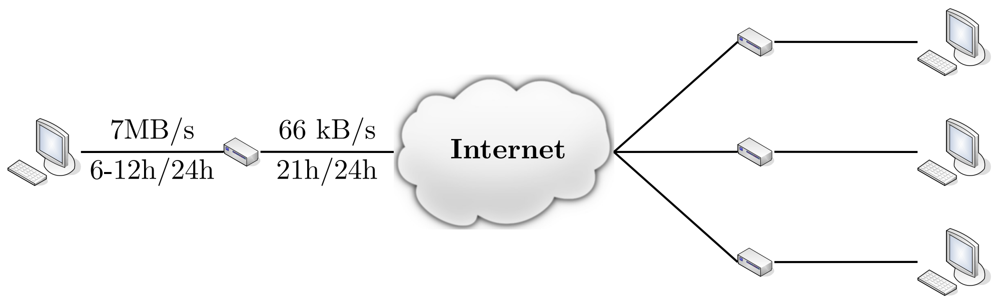
This increased stability makes gateways appealing candidates for backing peer-to-peer services. The intuition is that data is temporarily stored on gateways to compensate for peers transient availability. Note that we advocate the use of gateways as buffers and not storage. This choice is motivated by the increasing number of devices embedding storage, within the home and attached to a gateway. Dimensioning the storage of the gateway accordingly would be costly and would break the peer-to-peer paradigm by creating a central point in charge of hosting resources of attached devices durably: the contributed resources would no longer scale with the number of clients. In our buffer model, each device is required to provide a portion of its available space [15, 46], to participate to the global backup system.

2.2.2 System rationale

Our architecture is specifically tailored for the current architecture of residential Internet access. Indeed, most previous works assume that peers are directly connected to the network (see Figure 2.2a) while, in most deployments, a residential gateway is inserted in between the peers in the home network and the Internet. Hence, a realistic low-level network structure is composed of *(i)* peers, connected to the gateway through Ethernet or Wifi, *(ii)* residential gateways, providing the connection to the Internet, and *(iii)* the Internet, which we assume to be over provisioned (architecture depicted on Figure 2.2b). In our approach, we propose to use storage resources of residential



(a) With passive gateways



(b) With active gateways

Figure 2.2: A global picture of the network connecting the peers to the service. Those end-devices are available $6 - 12h$*/day. If we allow the gateway, which is available* $21h$*/day, to perform buffering, we can benefit from the speed difference between local links (7MB/s) and ADSL links (*$66KB$*/s).*

gateways, thus creating a highly available and distributed buffer to be coupled with peers.

Such an architecture is appealing as it takes into account *(i)* the availability that differs between peers and gateways, and *(ii)* the bandwidth that differs between the LAN and the Internet connection. Firstly, the peers tend to have a low to medium availability (*i.e.* from 25% or 6 hours/day on average on a Jabber trace, to 50% on a Skype trace we introduce later on) while gateways have a high availability (i.e., 86% or 21 hours/day on average). Secondly, peers are connected to the gateways through a fast network (at least 7MB/s) while the Internet connection (between gateways and the Internet) is fairly slow (*i.e.* 66 kB/s on average for ADSL or Cable). Our architecture exploits the major difference of throughput between the LAN and the Internet connection (WAN) by offloading tasks from the peer to the corresponding gateway quickly, thus using the Internet connection more efficiently (*i.e.* 21h/day instead of only $6 - 12$h/day on average).

This enables the large-scale deployment of online storage applications by fixing the issues provoked by the combination of slow up-links and short connection periods (as in the case of pure peer-to-peer). These issues are becoming increasingly important as the size of the content to backup increases while ADSL bandwidth has not evolved significantly over the past years. For example, uploading 1GB (a 300 photo album) to online storage requires at least 4h30 of continuous uptime. Hence, these applications require users to change their behavior (*e.g* let their computers powered for the whole night to be able to upload large archives); this limits their deployment and makes automated and seamless backup close to impossible. Our approach precisely aims at combining peers' fast but transient connections with gateways' slow but permanent connections. Following this logic, if peers upload directly to the Internet, they can upload on average 1.4-2.8GB/day (Fig. 2.2a); if we consider that the gateway is an active equipment that can perform buffering, a peer can upload 148-296GB/day to the gateway and the gateway can upload on average 4.8GB/day (Fig. 2.2b). We then advocate that turning the gateway into an active device can significantly enhance online storage services, be they peer-to-peer or cloud systems.

In the last part of this section, we propose the design of a gateway-assisted peer-to-peer storage system (GWA) based on these observations, and relying on two entities: *(i)* users' gateways, present in homes and providing Internet connectivity, and *(ii)* peers, being users' devices connected to the Internet (through a gateway) and having some spare resources to contribute to the storage system.

2.2.3 Gateway-assisted storage system

We consider a general setting to backup data to third parties on the Internet, generic enough for us to compare approaches from related work in the same framework.

The content to be backed up is assumed to be ciphered prior to its introduction in the system, for privacy concerns. The way on how this content can be located in the distributed storage system is outside the scope of this work. We consider that users upload data from one peer, under the form of archives. In order to achieve a sufficient reliability, the system adds redundancy to the content stored. To this end, it splits the archive into k blocks, and adds redundancy by expanding this set of k blocks into a bigger set of n blocks using erasure correcting codes [56] so that any subset of k out of n blocks allows recovering the original archive. This enables to increase the file availability as the resulting system-wide availability is

$$A = \sum_{i=k}^{n} \binom{n}{i} \bar{p}^i (1 - \bar{p})^{n-i} \tag{2.1}$$

where $\bar{p}$ is the average availability of peers, which is smaller than A. In the rest of this chapter, we set a target A_t for the system-wide availability so that n must be the smallest n ensuring that $A > A_t$. Intuitively, the availability targeted by the application is the portion of time a backed up data is online for restore. High availability rates have been shown cumbersome to reach in dynamic systems [14], so a reasonable trade-off should be considered [66].

For a backup operation, the client peer uploads the file and the redundancy blocks to other peers as follows:

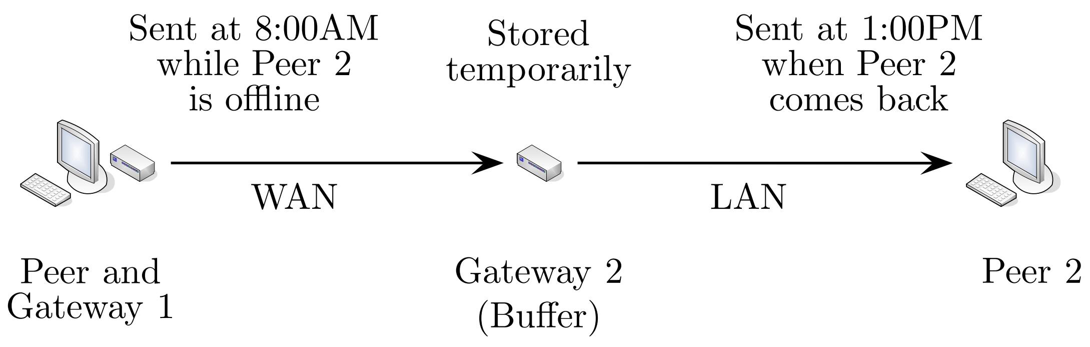

Sent at 8:00AM while Peer 2 is offline
Stored temporarily
Sent at 1:00PM when Peer 2 comes back
WAN
LAN
Peer and Gateway 1
Gateway 2 (Buffer)
Peer 2

Figure 2.3: Backup operation: buffering a block at a random gateway

- **1. Prepare.** As soon as it gets connected, the client peer starts pushing the archive at LAN speed to its gateway, which buffers the data. At this point, the data has been partially backed up but the final level of reliability is not yet guaranteed.

- **2. Backup.** In our system, the gateway is in charge of adding the redundancy; this allows faster transfer from the peer to the gateway as a lower volume of data is concerned. Once done, it starts uploading data to other gateways, at WAN speed (left-hand side of Figure 2.3). Gateways are active devices that can serve peer requests thus ensuring data availability and durability even if data is not fully pushed to remote peers. Therefore, data can be considered totally backed up when all blocks have reached the selected set of independent remote gateways.

- **3. Offload.** Finally, remote gateways offload, at LAN speed, the content to their attached peers (right-hand side on Figure 2.3) as soon as the attached peer becomes available.

A user can request access to its data at anytime; the success of immediate data transfer from the storage system to the requesting peer depends on the targeted availability of the backup, that has been set by the system administrator. To reclaim backed up data, the role of all elements in the systems are reversed and the restore is performed as follows:

- **1. Fetch.** To access a data, the requesting client peer informs its gateway of the blocks it is interested in. The client gateway carries on the download on behalf of the client peer by contacting the remote gateways handling peers where the data was uploaded. If the data was offloaded to some peer, it is fetched as soon as possible by the corresponding remote gateway.

- **2. Restore.** Then the remote gateway sends the data to the requesting client gateway.

- **3. Retrieve** When the client gateway has succeeded in getting the whole content (the data has been restored), it informs the client peer that its retrieval request has been completed, as soon as it connects back.

2.3 A comparison framework for backup schemes

We extensively evaluate our approach through simulations, taking as inputs execution traces of large-scale deployed systems. This section first presents the experimental setup, the competing approaches, namely pure peer-to-peer (noted *P2P* hereafter) and CDN-assisted (noted *CDNA*), and the performance metrics.

2.3.1 Parameters and data sets

The setting we described in previous section comes with the following set of parameters:

Network Bandwidth: On the WAN side (Internet connection), the bandwidth is heterogeneous and set according to the traces from a study of residential broadband networks [25]; it exhibits an average of 66 kB/s[2]. On the LAN side, we assume a constant bandwidth of 7 MB/s, capturing both wired and WiFi connections in home environments. The LAN bandwidth only impacts our gateway-assisted approach, as other approaches suffer from the bottleneck at the WAN interface since they upload data directly to devices in the WAN (peers or CDN).

Backup and Restore requests: The backup and restore user requests are modeled with Poisson processes, which are known to match user requests [41], of parameters λ_{backup} and λ_{restore} that represent the rate of backup and restore requests in the system. Each request is related to one archive. We assume that all archives have the same fixed size for all peers, typically representative of some large multimedia content (we make this size vary). As a result, the behavior of peers is assumed to be homogeneous: we are not aware of any trace giving hints about the real behavior of users within peer-to-peer storage/backup systems.

Peer availability: In order to model the up-time of personal computing devices (*i.e.* peers), we rely on two instant messaging traces. We only remove from the traces the peers with an availability lower than 1%, since these have a behavior not compatible with a backup application running in the background (*e.g.* people that have tried Skype only once but never use it again).

- **Skype** In this trace [41] of about 1269 peers over 28 days, the average availability of peers is 50%, which represents a medium availability when considering peer-to-peer systems.

- **Jabber** In this trace, provided by the authors of [52], the average availability is 27%. We used a slice of 28 days containing 465 peers.

[2]Note that all amounts of data and bandwidth are given in bytes.

All peers strictly follow the behavior of the trace. In particular, since our backup application runs in the background, *client* peers are not assumed to remain connected during the whole backup (their presence follows the IM/Skype trace), as opposed to the assumptions made in [52]. Note that these traces may however over-estimate uptime period since people using Skype are likely to be online more often than the average user.

Gateway availability: To model the up-time of residential gateways, we rely on our gateway trace presented in Section 2.2. Since the gateway and peer traces have been obtained independently, they do not capture the correlation between the behavior of a peer and of the associated gateway. Hence, we randomly assign a gateway to each peer. In order to avoid unrealistic scenarios where the peer is up while the gateway is down, we assume a gateway to be available when one of its attached peers is up, to allow communication between them: we rely on the gateway trace only for gateway to gateway communication.

Redundancy Policy: As explained previously, the redundancy policy is based on erasure correcting codes and is entirely determined by the number of blocks k each archive is split into and by the targeted availability A_t, which is set by the administrator. The backup is thus considered as complete when there are enough redundancy blocks n backed up in the network to guarantee a system-wide availability of at least A_t. The relation between the availability of peers and the values k and A_t has been studied in papers studying in details redundancy policies [14, 56, 84].

2.3.2 Competitors

We compare the performance of our *GWA* scheme against the two main classes of related backup systems, namely *P2P* and *CDN Assisted*, within the same simulation framework.

P2P The vast majority of peer-to-peer storage protocols historically presents a purely decentralized system with one-to-one uploads/downloads, without servers [49, 75]. They assume that gateways are passive devices that cannot store and forward but only route packets. This protocol is similar to the protocol we described in the previous section but does not have active gateways acting as buffers.

CDNA A possible enhancement consists in introducing a CDN to mitigate the low availability of peers [52]. The CDN is a central service in the core network, having unbounded capacity. We consider the most peer-to-peer variant of the protocol in [52] (*i.e.* the opportunistic one). In this protocol, the peers upload content to other peers in priority and upload to the CDN only when the whole bandwidth is not used (*i.e.*, not enough remote peers are available). This enables to lower time to backup by avoiding waiting times. However, the CDN does not upload the content to remote peers, but client peers eventually upload again to remote peers thus uploading twice in some cases. Indeed, pricing schemes at CDN implies that uploading from the CDN should be avoided so as to reduce costs. A schematical view is given in Figure 2.4. The CDN never fails, hence, a single copy of the content on CDN is enough to ensure

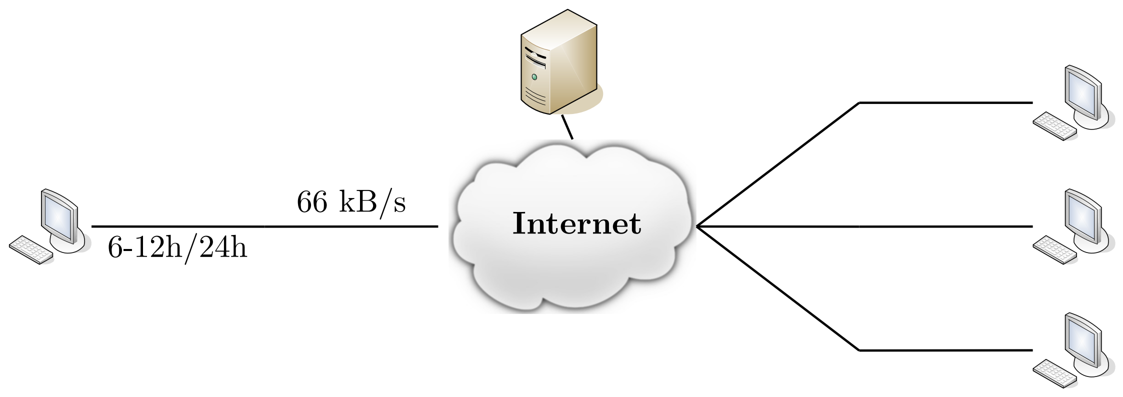


Figure 2.4: A global picture of the network connecting peers and CDN, as used in [52]. Note that the CDN (Server) has an infinite capacity and 100% availability. However, since the bandwidth used at the CDN is billed, the CDN is not used as a relay *but as another kind of end-storage (i.e. it does not upload content to other peers but only stores content temporarily until the backing up peers have uploaded content to enough peers.)*

an availability of 100%. As a result, a data backup is successful as soon as s fragments have been uploaded to the storage server and t fragments have been uploaded to the peers so that the targeted availability is guaranteed, as stated in (2.2).

$$\sum_{i=k-s}^{t} \binom{t}{i} \bar{p}^i (1-\bar{p})^{t-i} > A_t \tag{2.2}$$

$\bar{p}$ is the average availability of a peer and A_t is the targeted availability.

2.3.3 Simulation setup & Performance Metrics

We implement a *cycle-based* simulator to compare our *GWA* architecture with the two aforementioned alternatives. Depending on the simulated architecture namely *P2P*, *GWA* or *CDNA*, the system is respectively composed of only peers, peers and gateways, or peers and a CDN. Devices transfer their data depending on their upload bandwidth and on their behavior, which are *trace-driven* for peers and gateways while CDN is always up. We evaluate the time for each backup or restore request to be completed, while measuring the storage needs of the system.

In our simulations, the redundancy policy relies on the following values $k_{skype} = 16$, $k_{jabber} = 8$, $A_t = 0.7$, $\bar{p}_{skype} = 0.5$ and $\bar{p}_{jabber} = 0.27$ leading to $n_{\text{skype}} = 35$ and $n_{\text{jabber}} = 33$. We assign to each peer a set of n distinct remote peers in a uniform random way, for example using the hashing scheme of a distributed hash table. When a given peer requests a data backup operation, the n encoded blocks are placed on its set of n remote peers.

At each cycle, backup and restore requests are generated among peers according to the previously defined Poisson processes. Note that a backup request is uniformly distributed among the set of online peers, while a restore request is uniformly distributed among the set of peers whose data has already been fully backed-up (*i.e.* all blocks have been offloaded on remote peers). The rate of backup requests is set so that each peer requests to store about three archives of 1GB per month on average (*i.e.* $\lambda_{\text{backup}} = 0.4$ and $\lambda_{\text{restore}} = 0.05$ for the Skype trace).

Once all backup and restore requests have been distributed among the peers, each device (*i.e.* peer or gateway) transfers as much data as its upload bandwidth allows for the duration of the cycle (*i.e.* since residential connections are asymmetric, we assume that the system is bounded by the upload bandwidth of devices). If the remote device is offline, the client device has to wait until its reconnection.

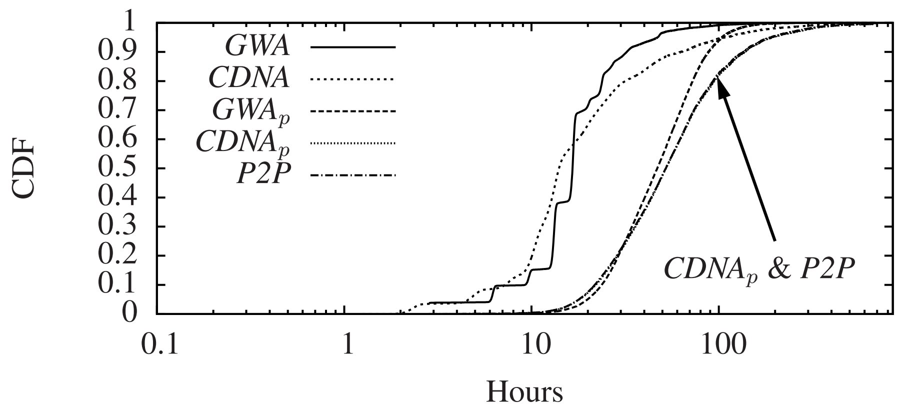
Our simulations were conducted with a cycle time-step set to 5 minutes. Figures report the average behavior over 50 simulations. We evaluate the three systems according to the following four metrics:

Time to backup, noted TTB. A backup request is considered successful when the data is stored safely. Safe storage is achieved when n blocks have been uploaded to the CDN and the remote peers for *CDNA*, to the remote gateways for *GWA*, and to the remote peers for the *P2P* thus satisfying the targeted availability described earlier. The time to backup is evaluated as the difference between the time the n^{th} block has been downloaded and the time of the backup request.

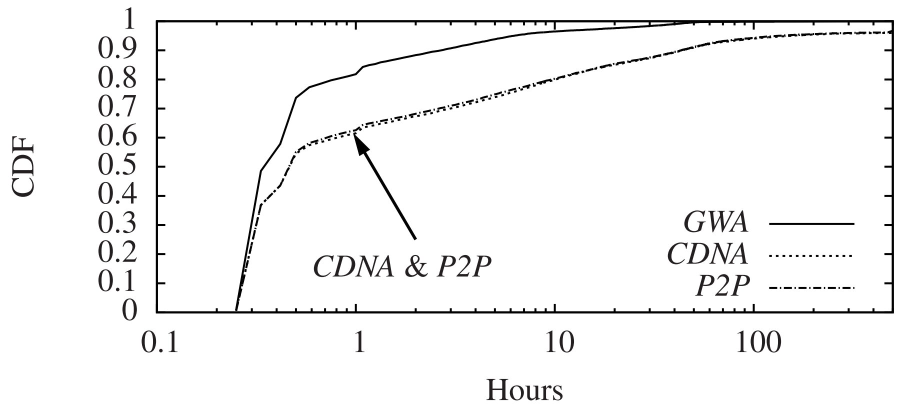
Time to offload, shown on the same plots as TTB. A variant measure for both *CDNA* and *GWA* is the time to fully offload an archive to the remote peers. This means that no data is left on the CDN or on gateways, accordingly. We note this variant $CDNA_p$ and GWA_p respectively.

Time to restore, noted TTR. A restore request is considered successful as soon as the data is restored safely at the user's place, that is to say when at least k blocks have been retrieved on the client peer, or on the gateway of the client peer for *GWA*. The time to restore is evaluated as the difference between the time the k^{th} block has been downloaded and the time of the restore request. We measure this time for random files after a long enough period, so that the selected files have been offloaded to peers. This represents the worst case for TTR, and this assumption reflects the fact that restore requests are more likely to happen long after backups.

Data buffered. It describes the size of the buffer that is required on gateways, for dimensioning purposes. We measured the maximum and the average amount of data stored on those buffers.



(a) TTB (Skype trace)



(b) TTR (Skype trace)

Figure 2.5: CDF of Time to Backup and Time to Restore for the Skype trace

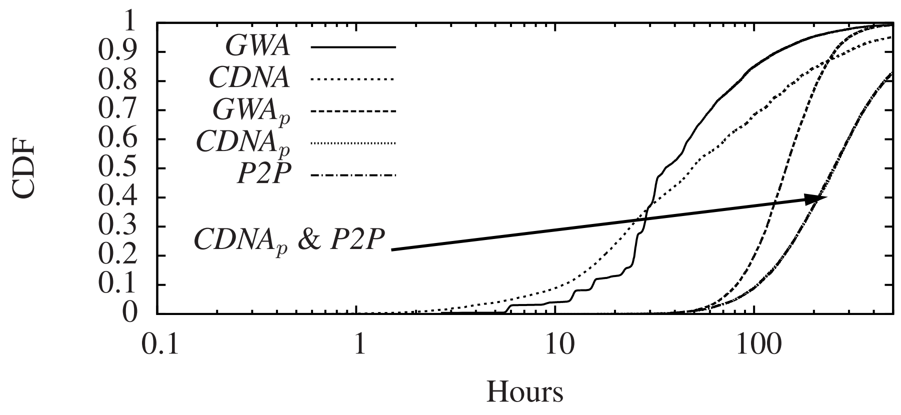
2.4 Results

2.4.1 Time to backup & restore results

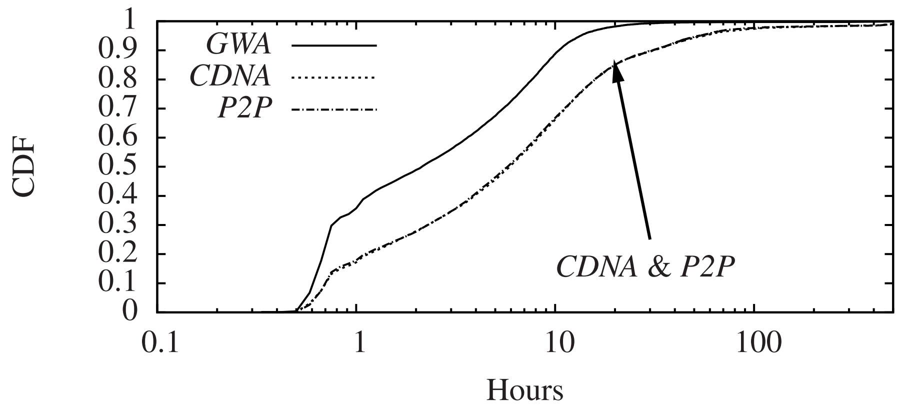
We first evaluate the performance of our approach to backup and restore data with regard to time. We plot the experimental cumulative distribution function (CDF) for TTB and TTR arguably the most critical metrics from the users' standpoint. CDF(t) represents the cumulative number of requests that have been completed after t hours.

Figure 4.6a depicts the TTB for the Skype trace. Results show that our gateway assisted approach improves the TTB over the *CDNA* approach[3]. Both considerably improve the performance over the *P2P* approach. Our *GWA* completes 90% of backup in 30 hours, the *CDNA* completes 90% of backups in 60 hours and the *P2P* completes 90% of backups in 140 hours. A consequence of this improvement is that data is backed up faster, reducing the window of potential losses.

[3]Note that results for *CDNA* are consistent with simulations made in [52], with an improvement factor between 2 and 3 over *P2P* storage



(a) TTB (Jabber trace)



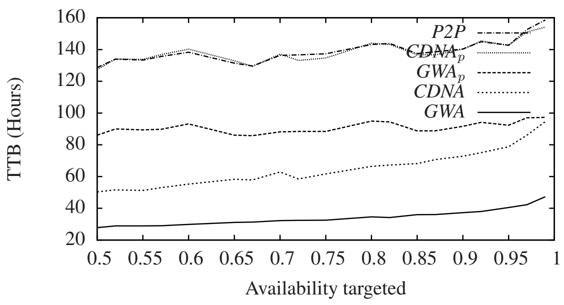
(b) TTR (Jabber trace)

Figure 2.6: CDF of Time to Backup and Time to Restore for the Jabber trace

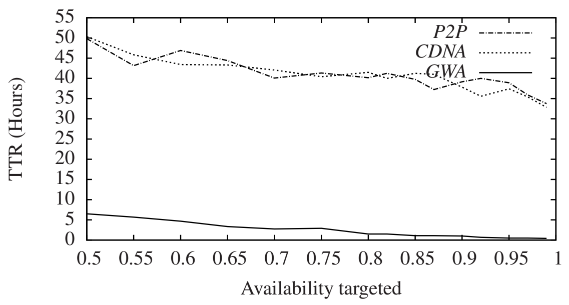
Along the performances of *P2P*, *CDNA* and *GWA*, we also indicate those of $CDNA_p$ and GWA_p. In *GWA* and *CDNA*, once the data is backed up (*i.e.* it is stored safely at some place being the remote gateways or the CDN), the backup process continues by offloading the data hosted on the remote gateways or on the CDN to the remote peers (*i.e.* until no data is left on the remote gateways or on the CDN). This process takes some time and its total duration is shown as GWA_p and $CDNA_p$. It is interesting to observe that while the *CDNA* does not enhance this time when compared to *P2P* (*P2P* curves and $CDNA_p$ curves are superimposed on all plots), our *GWA* approach improves this time significantly, reducing it from 140 hours to 90 hours (for 90% of backups to be offloaded).

We plot the time to restore for the Skype trace on Figure 4.6b. Results show that the TTR of our *GWA* approach is significantly reduced when compared to *GWA* and *P2P* approaches. Our *GWA* allows 90% of restores to be completed in less than 3 hours while both *CDNA* and *P2P* require 40 hours to complete 90% of restores. This makes the system much more user-friendly, when it comes to retrieving lost files.

In order to show that these results are not trace-dependent, we run the same set



(a)



(b)

Figure 2.7: Results (Skype trace), as a function of availability target, 90th percentile. Labels follow the order of the curves.

of experiments on a trace exhibiting a lower peer availability, namely the Jabber trace. We observe similar results. Previous observations hold as the time to backup (Figure 2.6a) for our *GWA* is reduced when compared to both *CDNA* and *P2P*. Similarly, the time to restore is also reduced (Figure 2.6b). Overall, the absolute times are increased when compared to results observed over the Skype trace because of the lower average availability of peers in this trace (25% instead of 50%).

Note that the absence of full convergence in some cases is due to some peers leaving the network for good before the backup is complete and because the trace duration is limited.

In the remaining part of this section, we evaluate the impact of other parameters, namely the targeted availability A_t, the number of blocks k, the number of peers involved in the system and the amount of data to backup and restore (*i.e.* the archive sizes) on both the TTB and TTR. We vary parameters one by one, keeping the other parameters to the values previously defined. Previous results have shown that some small part of backups may take significantly more time to complete than average; this somehow introduces a bias in observation of the system. In the following results, we will then consider the 90^{th} percentile for all approaches, in order to gain clear insights from systems' trends (*i.e.* the times displayed are the time so that 90% of the operations have been completed). The behavior, for one set of parameters, of the remaining 10% can be seen on the previously described CDF.

2.4.2 Influence of targeted availability

On Figure 2.7a, we plot the opposed variations of TTB and TTR when the targeted availability changes. We observe that the TTB increases when the targeted availability increases. This is due to the fact that achieving a higher availability for the file requires uploading more data, to introduce more redundancy. However, as shown on Figure 2.7b, increasing the targeted availability (*i.e.*, the amount of redundancy introduced) helps to reduce the TTR. As a result the targeted availability is a tradeoff

Table 2.1: Additional storage costs induced by various target availabilities, when compared to our reference target of 0.7

Target availability	0.5	0.7	0.9	0.95	0.99	0.995
Jabber (k=8)	−12%	-	27%	39%	67%	79%
Skype (k=16)	−11%	-	14%	20%	34%	40%

between having a low TTR and keeping the TTB reasonable. We observe that the improvement of our *GWA* approach is very significant over *CDNA* or *P2P* approaches. This is due to the fact that our approach represents a paradigm shift: the data is now considered as restored when stored in the home of the requester (on its gateway). Indeed, this prevents exterior factors such as block losses in the system from having any impact. When the requesting peer comes back online, it can seamlessly fetch data in order of minutes from its gateway, making this step nearly transparent when compared to operations at WAN speed.

Another aspect is impacted by the availability setting of the storage system: increasing the targeted availability requires to store more redundancy blocks thus increasing the storage costs per peer, as shown on Table 2.1. Hence, keeping the targeted availability reasonably low helps to keep storage costs per peer low thus increasing the total amount of data that can be backed up. In the remaining simulations, we choose to use a medium value of 0.7 for that targeted availability.

2.4.3 Influence of code length k

Increasing the code length k (*i.e.* the number of blocks required to decode) slightly improves both TTB and TTR, as shown on Figures 2.8a and 2.8b. However, increasing k also increases some side costs such as coding and decoding costs. As a consequence, we choose to set $k = 16$ to leverage the code properties while retaining low coding and decoding costs.

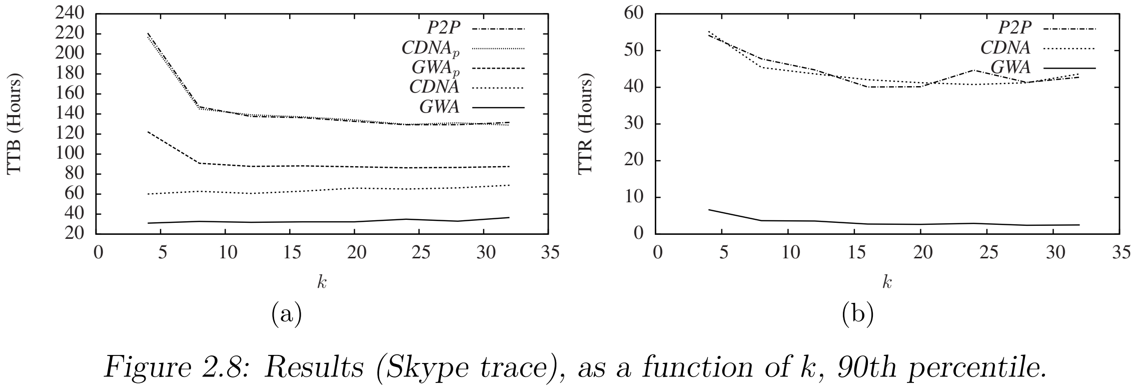


Figure 2.8: Results (Skype trace), as a function of k, 90th percentile.

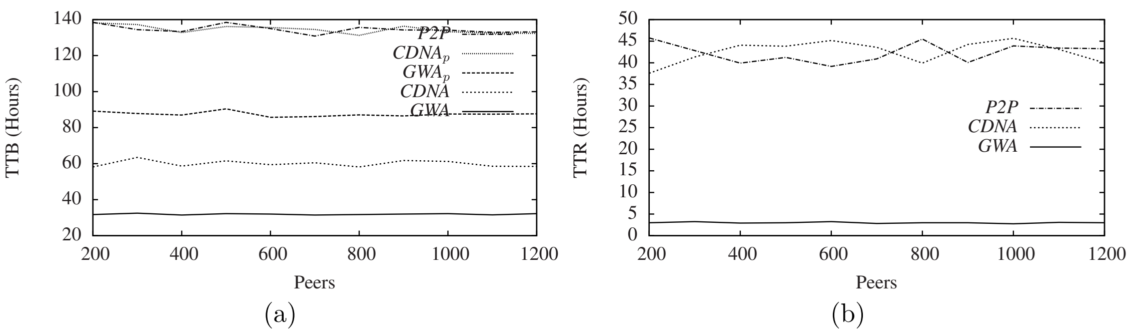


Figure 2.9: Results (Skype trace), with a varying network size, 90th percentile

2.4.4 Influence of the network size

Next, we evaluate the influence of the number of peers participating in the system by varying the number of peers from 200 to 1269 (all the peers of our trace). We also set the parameters for the restore and backup events so that on average each peer performs around 3 backup requests per month and 3 restore requests per year, similarly to what was done for all other simulations. Our results are shown on Figures 2.9a and 2.9b. Both TTB and TTR remain constant. Our simulations are limited by the total number of peers present in the trace; yet, we observe on this moderate scale that our system is likely to be scalable in the number of participating peers, as no bottleneck occurs at some gateways, which could have increased the TTR. Note that these constant TTB and TTR are consistent with the fact that the load (backup and restore requests) per peer is constant.

2.4.5 Influence of the data size

In order to assess the gains with respect to the usability of *GWA*, we plot the evolution of the TTB (Figure 2.10a) and the TTR (Figure 2.10b) as the amount of data to backup increases. Our architecture enables to leverage the difference of bandwidth between the local network and the Internet connection in order to use the uplink more efficiently (21h/24h instead of only 12h/24h); clearly, in this case, it enables users to backup larger amounts of data (here by a factor of 2). Note that our simulations are limited by the trace length (28 days).

2.4.6 Buffer dimensioning

For dimensioning purposes, we evaluate how much disk space should be provisioned on the gateways in order to implement our backup service. Figure 2.11 shows the CDF of maximal storage at gateways resulting from 1GB archive backups, at any time, on the Skype trace (*i.e.* the worst case space occupied on any gateway at any simulation step). Storage needs remain within reasonable bounds: 99% of gateways

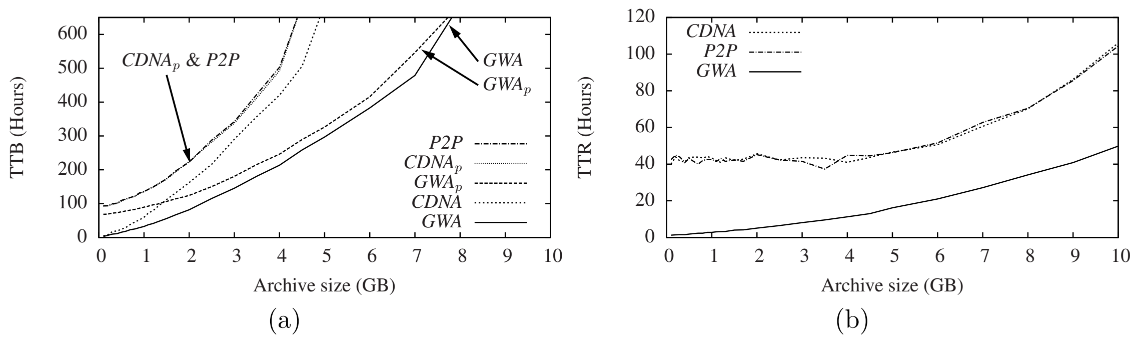


Figure 2.10: Results (Skype trace), with a varying amount of data to backup, 90th percentile

have stored at most 2.5GB of data at anytime. In a deployment scenario, if we limit the provisioned storage to 2.5GB at each gateway, the remaining 1% of gateways could ask peers pushing blocks to delay their query until their buffers have emptied, without impacting performance since it would impact less than 1% of peers and would occur only exceptionally.

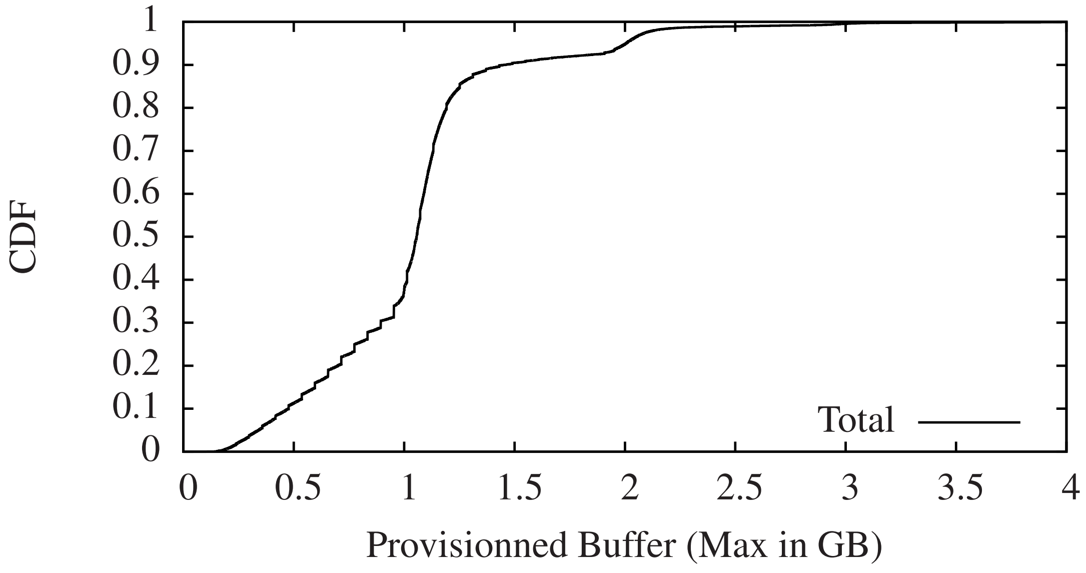


Figure 2.11: Maximal storage measured on gateways at any time (Skype trace)

Buffers are used in burst mode for relatively short periods of time, as shown on Figure 2.12, which plots the average of the same storage functionalities for the whole trace period: the average is negligible compared to the maximum presented on the previous figure. Furthermore, we observe that data is effectively offloaded since nothing remains on buffers when no new backups are requested (on the simulations, we performed backups only for the 13 first days) thus confirming results shown on previous curves for GWA_p.

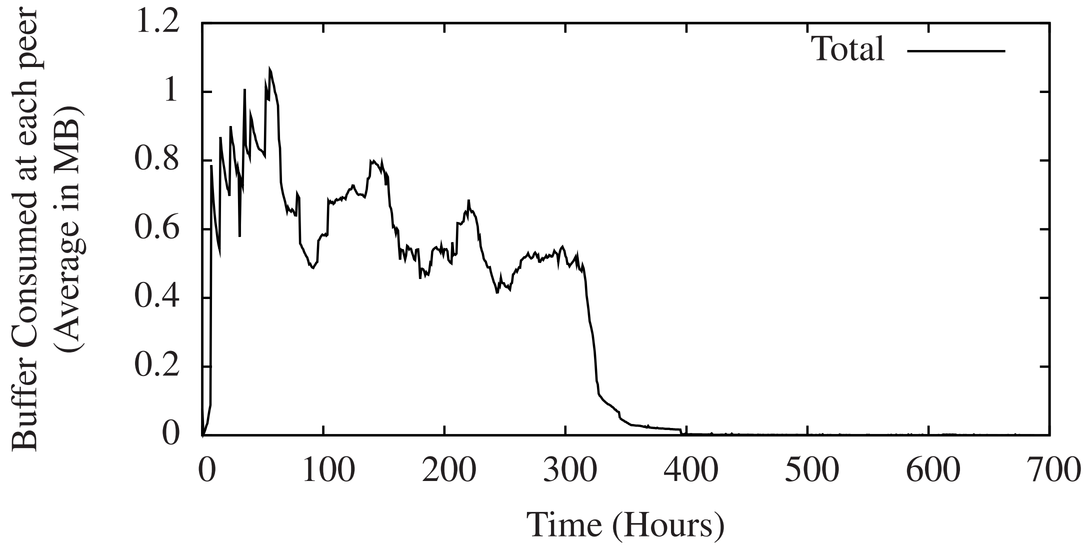


Figure 2.12: Average storage measured on gateways (Skype trace)

These results are related to our model where each gateway hosts one peer. The storage requirements would increase if multiple peers are behind each gateway. However, we study residential gateways, which are shared among a very small number of users. The storage requirements increase at most linearly with the number of users and hence would remain rather low as only a few peers are behind each gateway.

2.5 Discussion

Our results clearly indicate that our proposal efficiently distributes the centralized buffer scheme of [52], while increasing general backup performances and represents a significant improvement over previous approaches.

The Web architecture, in particular when considering CDN, relies on cache servers close to clients [54]. However, these servers are located within the Internet and cannot benefit from the difference of throughput between the local home network and the Internet: our system relies on the specific place of the gateway in between the home network and the Internet to leverage this difference. Moreover, cache servers are generally passive (*e.g.*, HTTP proxy) while in our system, the gateway is not only a cache but also an active participant that can serve directly other peers when data is requested.

Additionally, from the user's standpoint, our storage system could enable the bandwidth usage to be smoothed to provide users with a more transparent service (*i.e.* using the upload for backup when users are not using their computer/Internet connection). Indeed, using an important part of the upload bandwidth to quickly complete the backup operation, may severely affect the user's experience of Internet browsing or activity. A user's gateway is able to upload, as long as there is some available bandwidth and even if the user's computer is turned off (typically nightly). A similar advantage, appealing for Internet and service providers [51], is that such an architecture enables the implementation of scheduling policies for delaying transfers from gateways to the Internet so as to smooth the usage of the core network.

Lastly, this method also solves another issue that might appear when the distributed application operates worldwide. It has been shown that peers' availability patterns can vary according to local time (depending on geographical location) [12, 27]; in systems where some resources are insufficiently replicated, this could lead to an asynchrony of presence between a requesting peer and another peer hosting the resource. At best, the overlap of presence of both peers is sufficient to download the file, while it may also require a few sessions to complete, due to insufficient time overlap. As our *GWA* proposal relies on the hosting peer to upload the requested file to a more stable component (its gateway), asynchrony is no longer an issue as gateways provide stable rendezvous point between requesters and providers. This is of interest for delay tolerant applications such as backup, allowing the service to be operated with much lower costs on storage. Beyond the practical problem of using gateways in home environments, our solution then makes the case for leveraging clouds of stable components inserted in the network, to make them act as buffers in order to mask availability issues introduced in dynamic systems.

2.6 Related Work

We compared our approach to the peer-assisted one presented in [52] that leverages a central server and offloads backed up data to peers when they become available. Such a server-centric approach is also to be found in [36], where authors propose an hybrid architecture coupling low I/O bandwidth but high durability storage units (being an Automated Tape Library or a storage utility such as Amazon S3 [1]) with a P2P storage system offering high I/O throughput by aggregating the I/O channels of the participating nodes but low durability due to the device churn. This study provides a dimensional and system provisioning analysis via an analytical model that captures the main system characteristics and a simulator for detailed performance predictions. The simulator does not use real traces but synthetic ones, mainly to be able to increase the failure density and to reveal the key system trends. This work explores the trade-offs of this hybrid approach arguing it is providing real benefits compared to pure P2P systems [21, 49, 50, 55, 75]. Durability of the low I/O bandwidth unit is considered as perfect, but it always comes at a certain cost. In our approach, we do not assume we have such nodes and show that our approach is sustainable under real availability traces. Finally, FS2You [58] proposes a BitTorrent-like file hosting, aiming to mitigate server bandwidth costs; this protocol is not designed to provide persistent data storage.

The increasing power of residential gateways has enabled numerous applications to be deployed on them. This may allow savings in terms of power. One widely deployed system is the implementation of BitTorrent clients in those boxes (see *e.g.* FON [3]), which avoids the user to let her computer powered on [33]. Another example is the concept of Nano Data Centers [82], where gateways are used to form a P2P system to offload data centers. Similarly, some approaches were proposed to move tasks from computers to static devices as *set-top boxes*, for VoD [47] and IPTV [19]. Yet, those applications fully run on gateways while, in our approach, the gateway only acts as buffering stage.

Summary

This chapter addresses the problem of reducing the impact of churn on the available bandwidth. It has been widely acknowledged that availability of transient peers is a key parameter, that can by itself forbid a realistic storage service deployment if too low. We propose to address this inherently transient behavior of end peers by masking it through the use of more stable hardware, already present in home environments, namely gateways. Consequently, the available bandwidth of the system can be used more efficiently, thus augmenting the amount of data users can exchange. Our experiments, based on real availability traces, show that this architectural paradigm shift, significantly improves the user experience of storage systems over previous approaches, while remaining scalable.

We also provide a new trace of gateway availabilities, which is of interest for trace-based simulation of systems built upon gateways. As future works, it would be interesting to acquire a trace of user behaviors with respect to storage demand in peer-to-peer systems.

CHAPTER 3

Reducing the maintenance bandwidth

This chapter aims to reduce the maintenance bandwidth and presents two distinct contributions. In the first one, we propose to act at the upstream level of the maintenance, during the failure detection process. In order to avoid unnecessary bandwidth consumption due to incorrect failure detections, an adaptive and node-level timeout mechanism is described. The second part of this chapter presents a repair protocol especially designed for erasure-coded stored data. The proposed repair mechanism enables to repair multiple files simultaneously, thus factorizing the bandwidth costs. The first part of this work has been published in the proceedings of the IEEE International Symposium on Reliable Distributed Systems (SRDS 2012). The last part is under submission to date.

While the work presented in the last chapter aimed at increasing the available bandwidth to the user, the contributions described in this chapter concentrates on the reduction of the maintenance bandwidth. As already explained in section 1.2.4, a storage system has to provide a reliable maintenance mechanism over time so as to ensure data durability. This maintenance incurs a high bandwidth consumption and is of particular interest to minimize in order to avoid system congestion.

3.1 During detection: Adaptive node-level timeout

When designing a repair mechanism, the first crucial challenge is to decide when to trigger a repair. In other words, how to distinguish permanent failures requiring a repair from temporary disconnections. In the latter case, a repair may turn out to be useless, consuming bandwidth unnecessarily. A common and practical solution to decide on the nature of a failure is to use a timeout. Typically, once a node has not replied to a solicitation after a timeout has expired, it is considered as permanently failed. However deciding on a timeout value is difficult, especially at the scale of the entire system. More specifically, nodes may exhibit various availability patterns: defining a systemwide timeout value might not be optimal. Deciding on a value for an administrator is also a tedious and difficult task.

3.1.1 Introduction

In this chapter we take up the challenge of tackling the question *When should the system conclude on a transient or a permanent failure thus triggering a repair?* We argue that a one-size-fits-all approach is not sufficient and that the statistical knowledge of *every single node* availability, as opposed to system-wide parameters, provides a means to tackle those questions efficiently. In other words, while most previous works have either assumed that nodes are homogeneous or that simple averages on behaviors are representative, we account for the heterogeneity of nodes availability, in order to decide when repairs should be triggered.

Our repair mechanism trades traditional system-level timeout based on network-wide statistics against a per node timeout, based on the node availability patterns. Nodes self-organize to compute their adapted timeout in a probabilistic way, according to their own behavior and to the current redundancy factor.

The rest of this chapter is organized as follows: first we present our main motivations for this work in Section 2. Section 3 introduces the assumptions we consider. Our timeout strategie is then detailed in Sections 5 respectively. Then we present related work and conclude in Sections 7 and 8.

3.1.2 Related Work

Pioneer failure detection mechanisms in storage systems [22] were using *eager repair*, *i.e.* the system immediately repairs the loss of data as soon as a host failure is detected. However, this repair process is extremely bandwidth consuming as it does not take into account that failures can be transient. In [13], repairs are deliberately delayed in an attempt to mask transient failures. In [42], the authors suggest using a high redundancy level in order not to generate a repair in case of failure, providing data durability even if correlated failures occur; nevertheless, the counterpart is that the price in terms of generated storage overhead is very high. In [31], availability knowledge is used to adjust the repair rate of the system as a whole, not at the node level. The designers of Carbonite [20] were the first to place reintegration (ie. reintegrate replicas which have been wrongfully considered as failed) as a key of conceiving a functional distributed storage system. With regard to the distinction between transient and permanent failures some authors have recently proposed advanced timeout design. The authors in [69] analyze the use of timeout in a stochastic model and compare solutions with reintegration against standard ones. The approach of [81] is the closest to ours. However their proposed algorithm mainly focuses on the accuracy of instantaneous detection but does not consider the impact of the false positive or false negative errors on the average data availability. The authors of [85] are the first to model node behavior with a continuous semi-Markov chain, which does not require the use of exponential distribution. However, they always assume a scalar value and a homogeneous availability.

3.1.3 Leveraging predictability and heterogeneity

In this section, we review the two motivations of our availability-aware timeout, namely: *(i)* leveraging node predictability with respect to availability, and *(ii)* accounting for heterogeneous node availability patterns.

Availability and Predictability

Many originally proposed distributed storage systems do not explicitly deal with specific availabilities of nodes [22]. In quest for performance, some works have identified trends in the availability of the hosting resources [12, 15, 27, 53]. The important question of leveraging predictability has then been addressed [53, 60] to handle transient failures efficiently.

In this chapter, we propose to leverage this observed predictability in the behavior of nodes to provide an adapted timeout design that overcome availability-agnostic ones.

Heterogeneous availability patterns

Most of the systems that account for availability of resources rely on a single system-wide parameter, typically the mean or the distribution of availabilities of all nodes of the system [13, 81, 85]. While it is convenient to apply theoretical models such as Markov models or basic probabilities for the amount of redundancy to create, recent studies highlight why such models have limited applicability in practice [32, 40].

While some storage systems use platforms such as home gateways [82] that have a homogeneous and high availability, the majority of peer-to-peer deployment platforms exhibit a non-negligible heterogeneity in practice. To illustrate our claim, we plot on Figure 3.1 the mean and the standard deviation of availabilities of nodes composing systems such as Microsoft desktop computers, Skype or PlanetLab. Those availability traces, from scientific publications, are made available in a repository [6]. The figure shows a significant variance in node behavior. This is confirmed in a recent storage system analysis [52]. This trend is even more striking for the two rightmost systems. This demonstrates that availability cannot be accurately expressed by a basic scalar mean trying to represent the overall trend of the fraction of time nodes are up. Furthermore, reducing availability to a mean or a distribution [63] ignores information

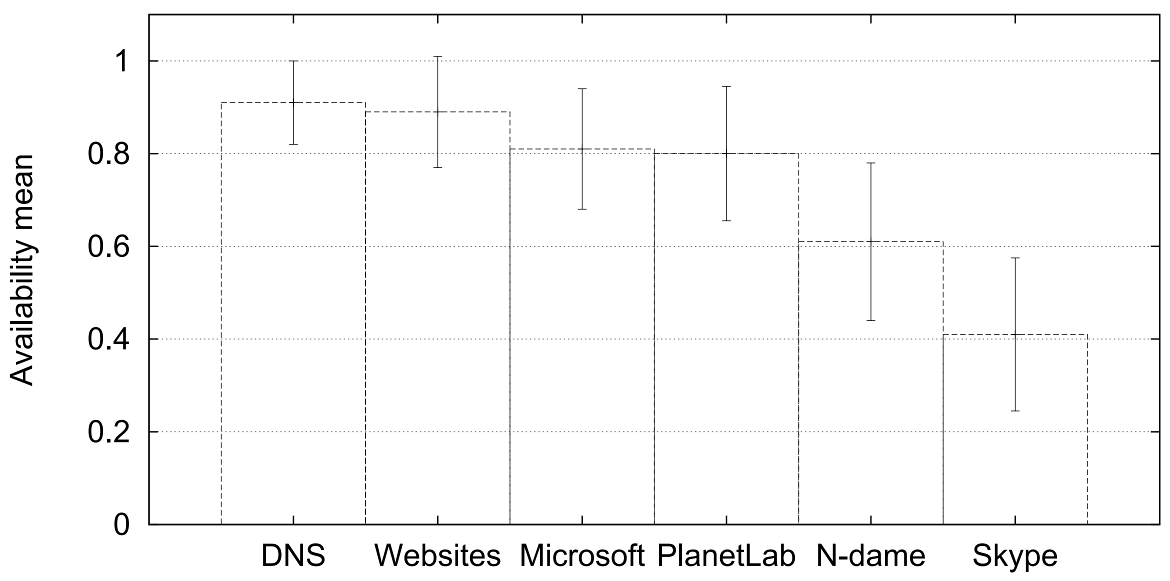


Figure 3.1: Availability mean and deviation for various existing systems

about node availability patterns while such information could be leveraged to increase the reliability of distributed storage systems [53, 60, 76]. Finally a recent study [65] confirms that disregarding heterogeneities in nodes availabilities affects negatively the performance of peer-to-peer storage systems.

3.1.4 Assumptions

Our availability-aware solution is specifically designed to be applied to any type of network logic where nodes may exhibit temporary and possibly recurrent, periods of unavailability. Systems with a near-perfect availability of their components obviously have no need for such a study. Furthermore, even systems exhibiting unpredictability of some significant part of nodes of the network can leverage our approach. Note that there is a direct relation between a repair and the bandwidth consumed to perform it. In the following, we evaluate the bandwidth consumption as the number of repairs.

In order to exploit information on node availability, our system must keep track of a limited history of those availability and unavailability periods. In practice, such availability history can be maintained by one or a few servers, the node itself providing it on demand, or by a distributed monitoring system if nodes cannot be trusted [61]. This availability history is represented as a vector of a predefined size (acting as a sliding window of time). For each time unit, the corresponding vector entry is set to 1 if the particular node was online at that time and -1 otherwise. We assume one-week history vectors and a one-hour time unit in the following. This length has been shown to capture most user behaviors accurately (*e.g.* diurnal and week end presence patterns) [12, 27, 53, 60].

We assume that the set of all storage nodes is partitioned into disjoint *clusters*, such that any node belongs to one, and only one cluster. Each file to store is encoded using an erasure code (k,n), and is randomly assigned to a single cluster. All the n encoded blocks of a given file are then stored on the n nodes of the same cluster (each cluster is thus composed of n distinct nodes). All nodes are monitored, either by a monitoring server, or through gossip for example [53, 76]: each node in a cluster then monitors other cluster nodes to detect failures.

Finally, as our aim is a pragmatic study of what can be achieved beside purely theoretical models for placement or timeouts, we use as a basis publicly available traces, that have been deeply studied in their respective original papers [6]. Those traces come from a wide range of systems; when applying techniques on them, the goal is to underline tendencies for the associated kind of availability they exhibit, more than just proposing a specific improvement for a narrow range of systems.

3.1.5 Per-node & adaptive timeout

This section describes how the timeout value is made both adaptive and per-node. Remember that in a cluster, all nodes are monitored; therefore for each node disconnection, a timer is started and runs until reconnection (either on a monitoring server, or by cluster nodes themselves, as explained in Section 3.1.4). The cluster is then aware of the unavailability duration of each disconnected node. As in the classical timeout model, if the unavailability period of a given node exceeds its timeout value, the node is considered as permanently down and the system provides this cluster with a new node to be used to *repair* the lost encoded block. As opposed to most approaches that set a system-wide timeout, we aim at determining an accurate timeout value reflecting each node's behavior with respect to availability. This tends to suggest that one might analyze node failure detection in a standard model of decision making under uncertainty. In fact the *a priori* probability for a given system is too coarse to provide usable information for each given node.

Contrarily, adding extrinsic information, such as the availability patterns of a given node, might help when observing a given unavailability time to evaluate the probability that this node will return, therefore that the disconnection (the failure) is transient. To illustrate our purpose, let us consider the following example: if a node, usually always available, is detected unavailable for a few hours, the probability that it will reconnect is low, and it gets lower as time passes. On the contrary, let us consider a node subject to a diurnal availability pattern. If such a node is detected unavailable for a few hours, the probability that it will reconnect is high. Hence, considering statistical knowledge at a node granularity is useful to determine the probability that a node will reconnect (i.e. the failure is transient). This motivates the need for *per-node* timeouts in heterogeneous systems.

In systems implementing *reintegration*, it may be the case that a node, wrongfully declared as permanently failed, is reintegrated after the repair has taken place. This yields the presence in the cluster of more encoded blocks than the n required. In our approach we propose to *adapt* each node's timeout dynamically to account for such situations. In case of an excess of encoded blocks, this is translated in our adaptive method by setting a less aggressive timeout for each cluster node.

Timeout model

The novelty of our timeout model is to be constructed at the node granularity. The model described below is then defined for each node, depending on its **own** attributes.

After the failure of a node, we call H_0, the event representing the fact that the node will return into the system and H_1, the event representing the fact the node will not return into the system. H_1 and H_0 are two disjoint events then $\Pr(H_1) = 1 - \Pr(H_0)$. $\Pr(H_0)$ is the *a priori* probability that the node will return into the system. $\Pr(H_0)$ is evaluated as the ratio between transient failures and all failures (either transient or permanent) in the system directly computed on a server. Note that this *a priori* probability could also, for example, be estimated from an aggregation protocol in a decentralized way. Let $t_{downtime}$ be the duration of the current downtime of the node and $t_{timeout}$ the timeout value associated to this node. Let its downtime distribution be f_d; this distribution verifies $\int_{(t=0)}^{\infty} f_d dt = 1$; then

$$\Pr(t_{downtime} > t_{timeout} \mid H_0) = \int_{(t=timeout)}^{\infty} f_d dt$$

By definition $\Pr(t_{downtime} > t_{timeout} \mid H_0) \in [0,1]$ and $\Pr(H_0) \in [0,1]$. $\Pr(t_{downtime} > t_{timeout} \mid H_0)$ then corresponds to the statistical knowledge on each node behavior. In fact as the availability history of each node is stored, its downtime probability distribution can be computed, and then also $\Pr(t_{downtime} > t_{timeout} \mid H_0)$ depending on the timeout value. By definition, $\Pr(t_{downtime} > t_{timeout} \mid H_1) = 1$ and $\Pr(t_{downtime} < t_{timeout} \mid H_1) = 0$. If the node does not come back into the system, on the one hand its downtime will be superior to any timeout value, on the other hand its downtime cannot be inferior to any timeout value. We also note:

$$P_{FA} = \Pr(H_0 \mid t_{downtime} > t_{timeout})$$

P_{FA} is called the False Alarm probability. P_{FA} represents the probability that a node comes back in the system while the system has decided that it would not (i.e. the node has been timed-out). The higher the P_{FA}, the more (probably) useless repairs are tolerated. According to Bayes' Theorem:

$$P_{FA} = \frac{\Pr(H_0 \cap (t_{downtime} > t_{timeout}))}{\Pr(t_{downtime} > t_{timeout})}$$

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{\Pr(t_{downtime} > t_{timeout})}$$

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{[\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)] + 1 - \Pr(H_0)}$$

leading to :

$$P_{FA} = \frac{\Pr(H_0) \times \Pr(t_{downtime} > t_{timeout} \mid H_0)}{1 + \Pr(H_0) \times [\Pr(t_{downtime} > t_{timeout} \mid H_0) - 1]} \tag{3.1}$$

The definition interval of P_{FA} is then $[0, \Pr(H_0)]$.

To sum up, Equation (3.1) expresses the false alarm probability as a function of the *a priori* probability that a node will eventually come back into the system and of its downtime distribution. This expression is necessary to compute per-node timeouts, as explained in the next section.

Computing an adaptive per-node timeout

The scheme presented hereafter is performed in each cluster at each time unit (order of hours or days, depending on the stability of the system). Each cluster has to deal with its departed and returning nodes. Returning nodes reintegrate their cluster. On the contrary, if a node is timed-out and if the cluster size falls under n, the system provides this cluster with a new node in order to create a new encoded block. Those steps are performed periodically within each cluster:

- Update cluster so as to reintegrate wrongfully timed-out nodes & insert new nodes to replace timed-out ones
- Compute false alarm probability (at the cluster level)
- Compute timeout of each node in the cluster

The false alarm probability varies depending on how critical the situation is, with regard to the number of alive nodes in the cluster. We propose hereafter a method used to evaluate this situation (note that this method is not proved as being an optimal one, but just a practical example that we use for evaluation). At each time unit, we compare the current number of available encoded blocks against the one on the previous week. This difference, noted Δ, is positive if there are more blocks currently than in the past, negative otherwise. Note that the only parameter measured is the number of available blocks, regardless of which node stores them. If there is no difference, the false alarm probability is defined as the half of its interval of definition so $P_{FA} = \Pr(H_0)/2$. We define the step of variation as $\Pr(H_0)/n$, with n the initial number of nodes in a cluster. P_{FA} is computed as: $P_{FA} = \Pr(H_0)/2 - (\Delta \cdot \Pr(H_0)/n)$. At the end of this step, each cluster has a false alarm probability that is a function of the measured criticality of the situation.

Once the false alarm probability has been determined in each cluster, each node is able to specify its own timeout value. According to Equation (3.1) we have:

$$\Pr(t_{downtime} > t_{timeout} \mid H_0) = \frac{P_{FA} \cdot (1 - \Pr(H_0))}{\Pr(H_0) \cdot (1 - P_{FA})}$$

	(k=8, n=16)	(k=8, n=24)	(k=16, n=32)	(k=16, n=48)	mean
Savings	20.4%	32.9%	26.0%	41.6%	30.2%

Table 3.1: Reduction of the number of repairs (Skype)

$$\int_{(t=timeout)}^{\infty} f_d dt = \frac{P_{FA} \cdot (1 - \Pr(H_0))}{\Pr(H_0) \cdot (1 - P_{FA})} \tag{3.2}$$

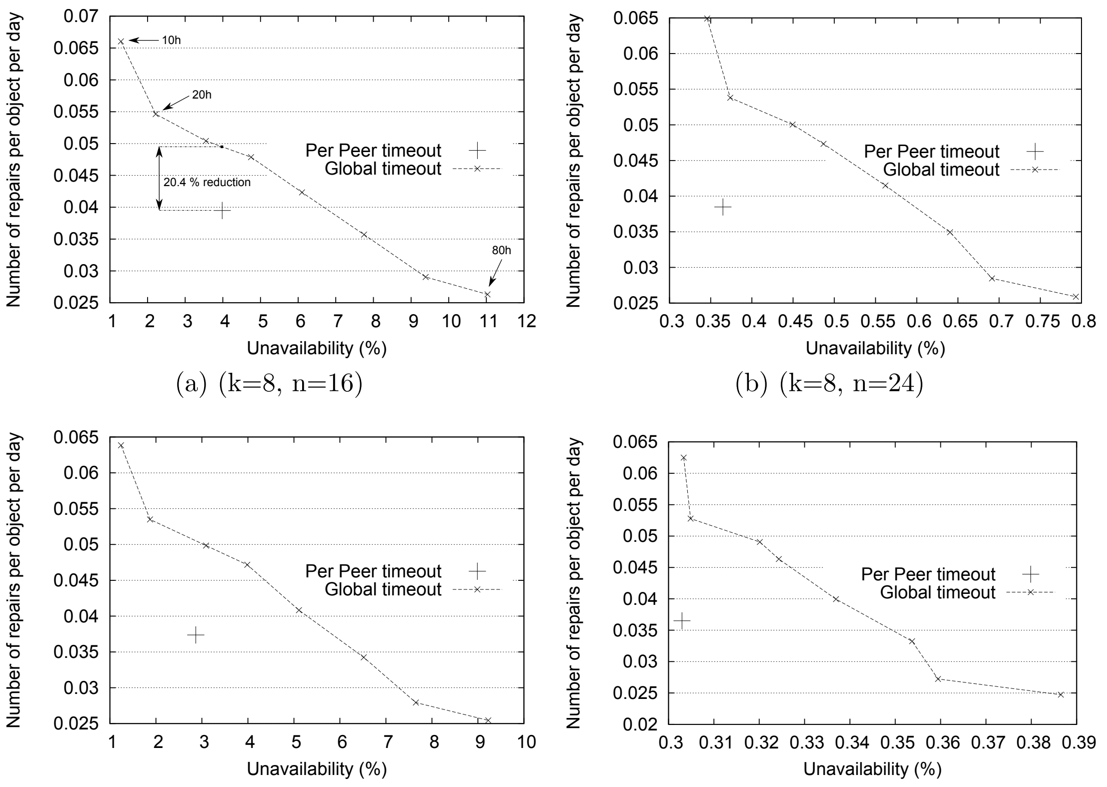
The timeout value is thus set so as to solve Equation (3.2). In practice, the timeout value is incremented until the integral is greater than the right term.

3.1.6 Evaluation

We evaluate our per-node timeout on public traces. We compare the cost/availability trade-off of our approach against systems using global timeout from aggressive (low) to relaxed (high) values. Encoded blocks of false positive nodes are also reintegrated in the global timeout simulation. We evaluate the trade-off on three different system traces, namely PlanetLab, Microsoft and Skype. As in Section 2.3 nodes having an uptime below 1% have been removed from the trace, resulting in a number of alive nodes respectively equal to 308, 51,313 and 1,901. The simulation is performed on a three-week period. The *learning period* is the first week, after which the history window simply slides. In fact an initial availability history must be available so that each node is able to compute its initial downtime distribution. Data availability mean and number of repairs are then evaluated during the next two weeks. During these two weeks downtime distributions of each node are updated following their behavior, after each new end of a downtime session.

Each of the *alive* nodes stores one data item. We evaluate our approach and the global timeout one along the following metrics: mean data availability and number of repairs generated by timed-out nodes. Results for the Skype, Planet Lab and Microsoft traces are respectively plotted on Figure 4.7, 3.3 and 3.4 for various code parameters. The X-axis represents the mean of data unavailability in percent (*i.e.* $1 - availability$). The Y-axis is the number of repairs per data item and per day, it is equivalent to the number of timed-out nodes, as each of them triggers a repair. Then unavailability versus the number of repairs is plotted for various values of global timeout (10 to 80 hours in Figure 4.7a) and for the per-node timeout, in the classic way of representing timeouts versus repairs, as done in recent papers [81, 85].

Table 3.1, 3.2 and 3.3 respectively summarizes the savings on the number of repairs depending on the code parameters, for the Skype, Planet Lab and Microsoft trace when using adaptive timeout, compared to global ones. Savings are given in percent and evaluated for equivalent availability which is linearly interpolated for global timeouts and for each code parameters. For instance, for ($k = 8, n = 16$) (Figure 4.7a) the interpolated point has coordinates (4 , 0.0495). For an equivalent unavailability of 4%



(a) (k=8, n=16) (b) (k=8, n=24)

(c) (k=16, n=32) (d) (k=16, n=48)

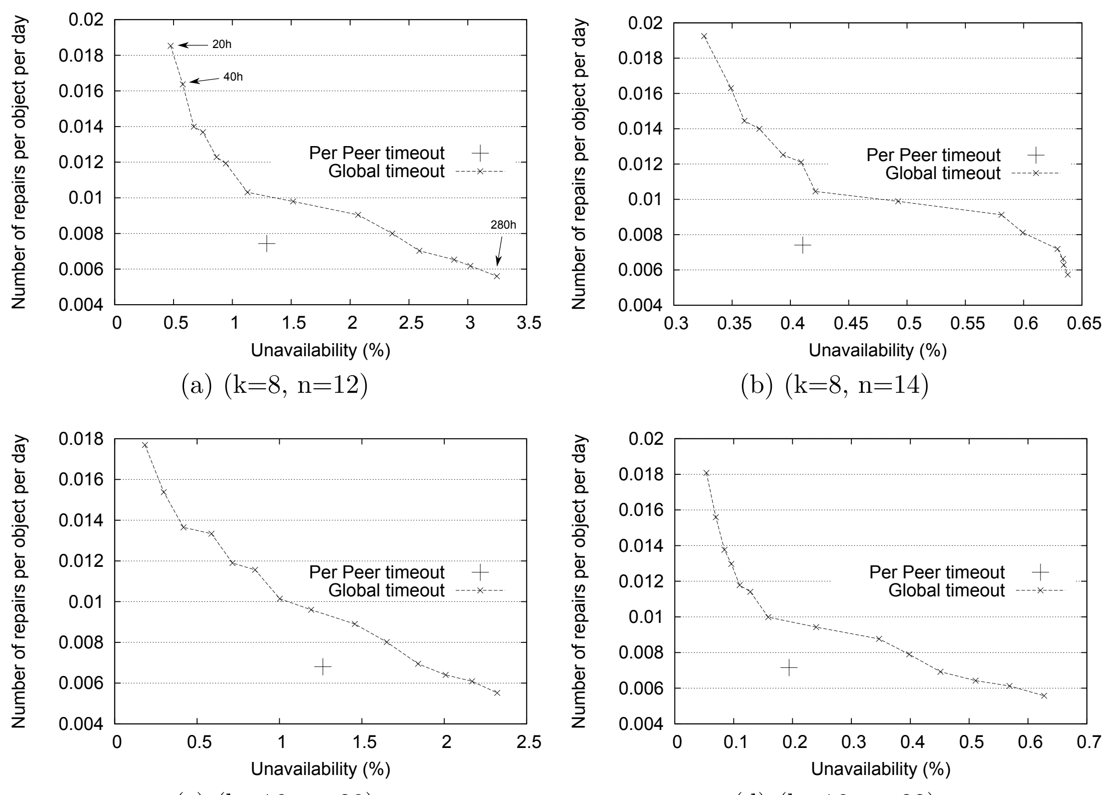
Figure 3.2: Skype results for various code parameters.

	(k=8, n=12)	(k=8, n=14)	(k=16, n=20)	(k=16, n=22)	mean
Savings	26.4%	37.8%	27.7%	26.6%	29.6%

Table 3.2: Reduction of the number of repairs (Planet Lab)

the global timeout would lead to 0.0495 repairs by object and by day whereas our per-node timeout only need 0.0394, thus resulting in a 20.4% saving.

Unsurprisingly, a first observation applicable to all plots is that an aggressive global timeout value (10H) leads to a low unavailability but produces a high repair rate, triggering an important bandwidth consumption in practice. On the other hand, a relaxed timeout value (80H) enables to reduce the number of repairs at the expense of decreased data availability. A second observation is that regardless of the global timeout value, our per-node timeout always provides a better trade-off between availability and the number of repairs. In other words, in all cases for an equivalent availability (and obviously for the same code parameters), the number of repairs will always be higher using global timeout than our per-node timeout. Note that both traces show various levels of node predictability (especially when using a one-week prediction period) [60]. Our approach thus saves significant bandwidth, resulting from



(a) (k=8, n=12) (b) (k=8, n=14)

(c) (k=16, n=20) (d) (k=16, n=22)

Figure 3.3: Planet Lab results for various code parameters.

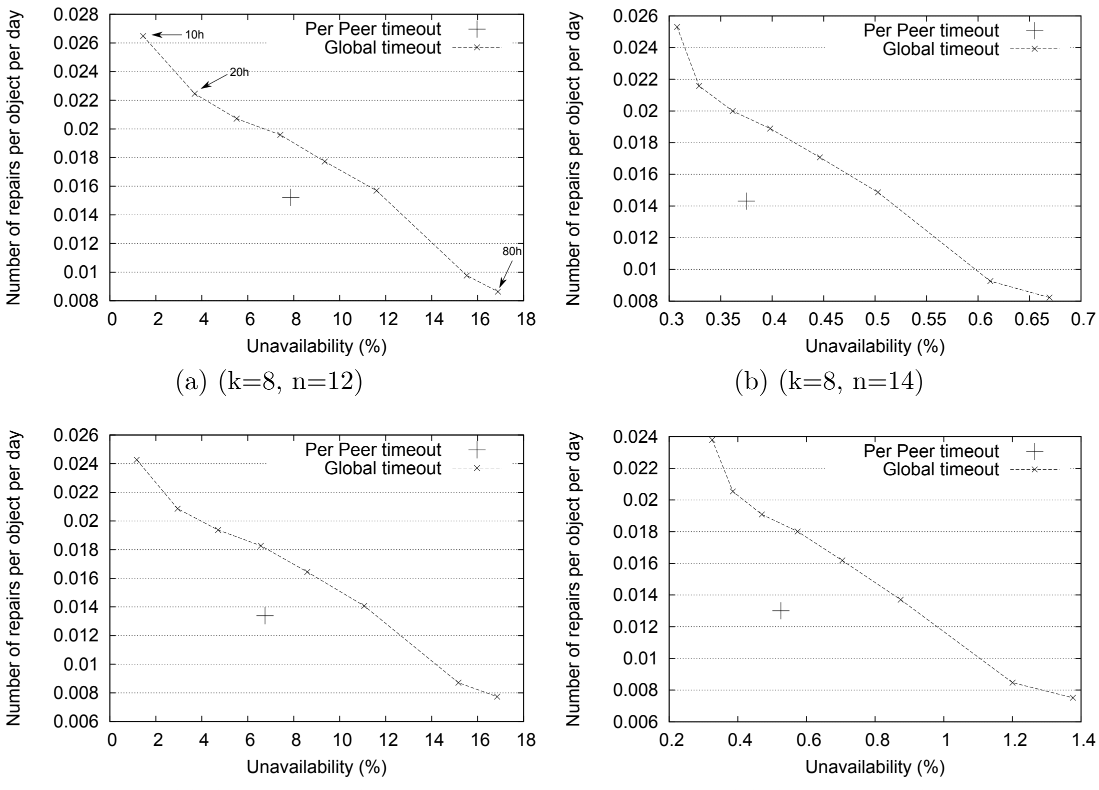
	(k=8, n=12)	(k=8, n=14)	(k=16, n=20)	(k=16, n=22)	mean
Savings	20.5%	26.9%	26.1%	29.7%	25.8%

Table 3.3: Reduction of the number of repairs (Microsoft)

the decreased number of repairs. We also emphasize that not only raw performance on those metrics is greatly improved, but another important benefit of the adaptive timeout method is that a system administrator does not have to arbitrarily set a static value for timeout. The system self-organizes for adapted value computation, suppressing the need for this decision before runtime.

Summary

In this chapter, we aimed to reduce the consumed bandwidth due to wrongfully detected failures. To this end, we demonstrated the interest of leveraging the knowledge of node availability patterns, in particular when node exhibit heterogeneous behavior with regard to uptime. We have advocated an implementation at a fine granularity, namely on a per node basis. Evaluations conducted on real traces have shown to



(a) (k=8, n=12)

(b) (k=8, n=14)

(c) (k=16, n=20)

(d) (k=16, n=22)

Figure 3.4: Microsoft results for various code parameters.

decrease the number of unnecessary repairs. This directly translates into important savings in terms of the maintenance bandwidth consumption.

3.2 During reparation: Clustered Network Coding

Once a failure has been declared as permanent by any failure detection mechanism, a repair has to be performed to maintain the initial redundancy level of the system. While repairing a replica is straightforward, as it only requires its single transfer, repairing erasure coded data is much more challenging. In fact, it incurs a high overhead in terms of bandwidth utilization as well as decoding operations as reminded in section 1.2.4.

3.2.1 Introduction

Several major storage systems as those of Microsoft [17], Google [34] or even Facebook [80] have recently adopted erasure codes, more specifically *Reed-Solomon codes*. There is thus a tangible move from replication to erasure coding, allowing a more efficient use of scalability-critical resources.

Reed-Solomon codes are the de facto standard of code-based redundancy in practice. However, those codes have been designed and optimized to deal with lossy communication channels but do not specifically target networked storage systems. Indeed, Reed-Solomon codes are precisely known to suffer from important overhead in terms of bandwidth utilization and decoding operations when maintenance has to be triggered.

In order to address these two drawbacks, architectural solutions have been proposed [72], as well as new code designs [30, 45, 48], paving the way for better tradeoffs between storage, reliability and maintenance efficiency. The optimal tradeoff has been very recently provided by Dimakis & al [23] with the use of *network coding*. However open issues regarding the feasibility of deploying those new codes in practical distributed storage systems remain. More specifically, most studies focuses on theoretical results and very few evaluate how hard it is to implement theses codes in a production system [29]. Moreover those new codes are examined under the simplifying assumption that only one file is stored per failed machine, thus ignoring practical issues when dealing with the maintenance of multiple files.

In this chapter, we propose a repair mechanism which makes use of network coding to repair multiple files simultaneously, without any decoding operations. We call this approach *CNC*, for Clustered Network Coding. This mechanism is made as simple as possible, both in terms of design and implementation with the purpose of leveraging the power of erasure codes, while reducing its known drawbacks.

The rest of this chapter is organized as follows. Our approach is presented in Section 3.2.3 and analyzed in Section 3.2.4. We then evaluate our implementation and compare it against state of the art approaches in Section 3.2.5. Finally, we present related work in Section 3.2.2 and conclude in Section 3.2.5.

3.2.2 Related Work

The problem of efficiently maintaining erasure-coded content has triggered a novel research area both in theoretical and practical communities. Design of novel codes tailored for networked storage system has emerged, with different purposes. For instance, in a context where partial recovering may be tolerated, priority random linear codes have been proposed in [57] to offer the property that critical data has a higher opportunity to survive node failures than data of less importance. Another point in the code design space is provided by self-repairing codes [64] which have been especially designed to minimize the number of nodes contacted during a repair thus enabling faster and parallel replenishment of lost redundancy.

In a context where bandwidth is a scarce resource, network coding has been shown to be a promising technique which can serve the maintenance process. Network coding was initially proposed to improve the throughput utilization of a given network topology [10, 38]. Introduced in distributed storage systems in [23], it has been shown that the use of network coding techniques can dramatically reduce the maintenance bandwidth. The authors of [23] derived a class of codes, namely *regenerating codes* which achieve the optimal tradeoffs between storage efficiency and repair bandwidth. In spite of their attractive properties, *regenerating codes* are mainly studied in an information theory context and lack of practical insights. Indeed, this seminal paper provides theoretical bounds on the quantity of data to be transferred during a repair, without supplying any explicit code constructions. The computational cost of a random linear implementation of these codes can be found in [29]. A broad overview of the recent advances in this research area are surveyed in [24].

Very recently, authors in [48] and [68] have designed new code tailored for cloud systems. In [48], the authors proposed a new class of Reed-Solomon codes, namely *rotated Reed-Solomon* codes with the purpose of minimizing I/O for recovery and degraded read. *Simple Regenerating Codes*, introduced in [68] reduce the maintenance bandwidth while providing exact repairs, and simple XOR implementation. Yet this reduction comes at the price of loosing the optimal storage efficiency and, to the best of our knowledge, those new codes have not been implemented to date.

Some other recent works [43, 44] aim to bring network coding into practical systems. However they rely on code designs which are not MDS, thus consuming more storage space, or are only able to handle a single failure hence limiting their application context.

3.2.3 Clustered Network Coding

The CNC repair mechanism is designed to sustain a predefined level of reliability, *i.e.* of data redundancy, by recovering from failures with a limited impact on performance. We assume that the failure detection is performed by a monitoring layer and that the system triggers the repair process, assigning new nodes to replace the faulty ones, in charge of recovering the lost data and store it. The predefined reliability level is set by the storage system operator. This reliability level then directly translates into the redundancy factor to be applied to files to be stored, with parameters k (number of blocks sufficient to retrieve a file) and n (total number of redundant blocks for a file). A typical scenario for using CNC is a storage cluster like in the Google File System [37], where files are streamed into extents of the same size, for example 1GB as in Windows Azure Storage [17]. These extents are erasure coded in order to save storage space.

Random Codes

We propose to use *random linear codes* (random codes for short) as an alternative to classical Reed-Solomon codes. Interestingly, randomness can provide a very simple and flexible way to construct MDS codes[1] (as are Reed-Solomon codes, see section 1.2.2) with very good properties. We argue that random codes may offer an appealing alternative to classical erasure codes in terms of storage efficiency and reliability, while considerably simplifying the maintenance process. Random codes have been initially evaluated in the context of distributed storage systems in [8]. Authors showed that random codes can provide an efficient fault tolerant mechanism with the property that no synchronization between nodes is required. Instead, the way blocks are generated on each node is achieved independently in such a way that it fits the coding strategy with high probability. Avoiding such synchronization is crucial in distributed settings, as also demonstrated in [38]. We remind the basics about random codes in the following sections.

Encoding

Encoding a file using random codes is simple: each file is divided into k chunks and the blocks stored for reliability are created as random linear combinations of these k blocks (see Figure 3.5). More formally, each file of size $\mathcal{M}$ is chunked into k blocks k_i of equal size $\frac{\mathcal{M}}{k}$ with $i \in [1, k]$. Each block k_i belongs to $\mathbb{F}_q^l$ ie. vectors of size l in a finite field of size q (usually $GF(q)$ with $q = 2^p$). For all practical implementations we usually have $p = 8$ or $p = 16$.

Each peer j stores a block X_j which is a random linear combination of these k_i blocks:

$$X_j = \sum_{d=1}^{k} \alpha_d k_d \tag{3.3}$$

The α_d coefficients are chosen uniformly at random in the field $\mathbb{F}_q$, ie. $Pr(\alpha_d = \alpha) = \frac{1}{q}, \forall \alpha \in \mathbb{F}_q$. Note that the arithmetic is performed over the finite field $\mathbb{F}_q$.

[1]With a probability close to one

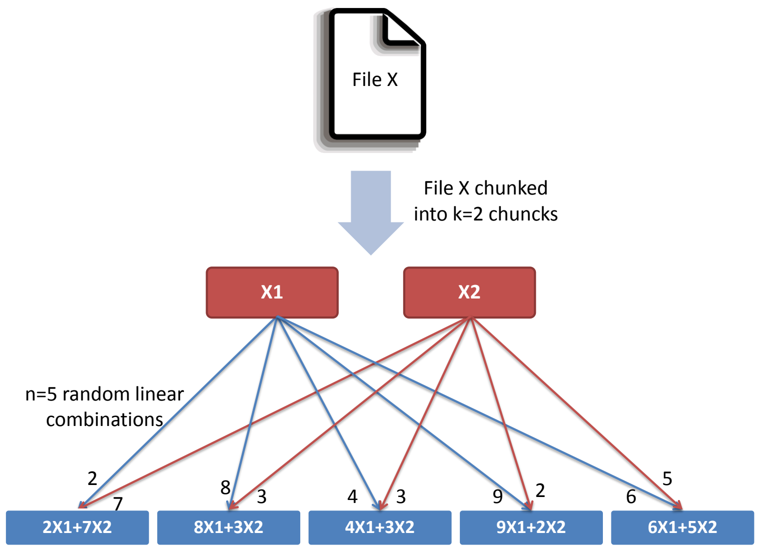

File X
File X chunked into k=2 chuncks
X1
X2
n=5 random linear combinations
2
7
8
3
4
3
9
2
6
5
2X1+7X2
8X1+3X2
4X1+3X2
9X1+2X2
6X1+5X2

Figure 3.5: Creation process of encoded blocks using a random code. All the coefficients are chosen randomly. Any $k = 2$ blocks is enough to reconstruct the file X.

In addition to store the block X_j, we also need to store the associated random coefficients $(\alpha_1, \alpha_2, ..., \alpha_k)$. This is typically a negligible overhead compared to the size of each block X_j.

Note that this is a different encoding paradigm compared to classical erasure codes, with a fixed encoding matrix and thus a fixed rate $\frac{k}{n}$. In fact when using random codes, the notion of *rate* disappears as with the k blocks one can generate as many *redundant* blocks as it wants. In fact one just needs to make a new random combinations of the k blocks. This property makes random codes a *rateless* code (fountain code).

Decoding

In mathematical terms, each redundant block which has been created as a random linear combination of the blocks $\{X_1, X_2, ..., X_k\}$ can be seen as a random vector of the subspace spanned by $\{X_1, X_2, ..., X_k\}$. Then the reconstruction of a file boils down to get k linearly independent vectors in this subspace. In fact every family of k vectors which is linearly independent forms a non-singular matrix which can be inverted and thus the file can be recovered. Theory on random matrix over finite field tells us that if one takes k random vectors of the same subspace, these k vectors are linearly independent with a probability close to one. More formally, let D be the random variable denoting the dimension of the subspace spanned by k random vectors which belong to $\mathbb{F}_q^k$.

Then it can be shown that:

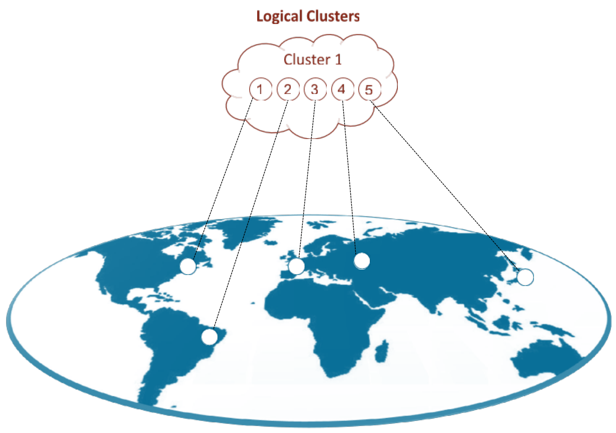


Figure 3.6: Clusters are constructed at a logical level : nodes participating in a given cluster may span geo-dispersed sites for reliability.

$$Pr(D = k) = \frac{(q^k - 1)\prod_{i=1}^{k-1}(q^k - q^i - 1)}{q^{(k^2)}} \tag{3.4}$$

This equation gives the probability that the dimension of the subspace spanned by k random vectors is *exactly* k, and so that the family of these k vectors is linearly independent. This probability is shown to be very close to one for every k when using practical field sizes (typically 2^8 or 2^{16}). For example for a field size of 2^{16} and for $k = 16$ which are classical and practical values, when contacting exactly $k = 16$ nodes the probability to be able to reconstruct the file is 0.999985.

In other words, the property of randomness tells us that with a probability close to one, an encoded file can be reconstructed with exactly any k redundant blocks (as mentioned above this is optimal). Note that each file is thus encoded with a different random generator matrix , and that this matrix is never constructed explicitly and does not even exist in one place in the network. This is not a problem as the only requirement is that each block is stored with its associated coefficients.

To summarize, random codes provide a very simple and flexible way to encode data optimally (MDS codes) with high probability. In addition their rateless property is very suitable in the context of distributed storage system as it leaves reintegration possible (see next section).

A Cluster-based Approach

To provide an efficient maintenance, CNC relies on *(i)* hosting all blocks related to a set of files on a single cluster of nodes, and *(ii)* repairing multiple files simultaneously .

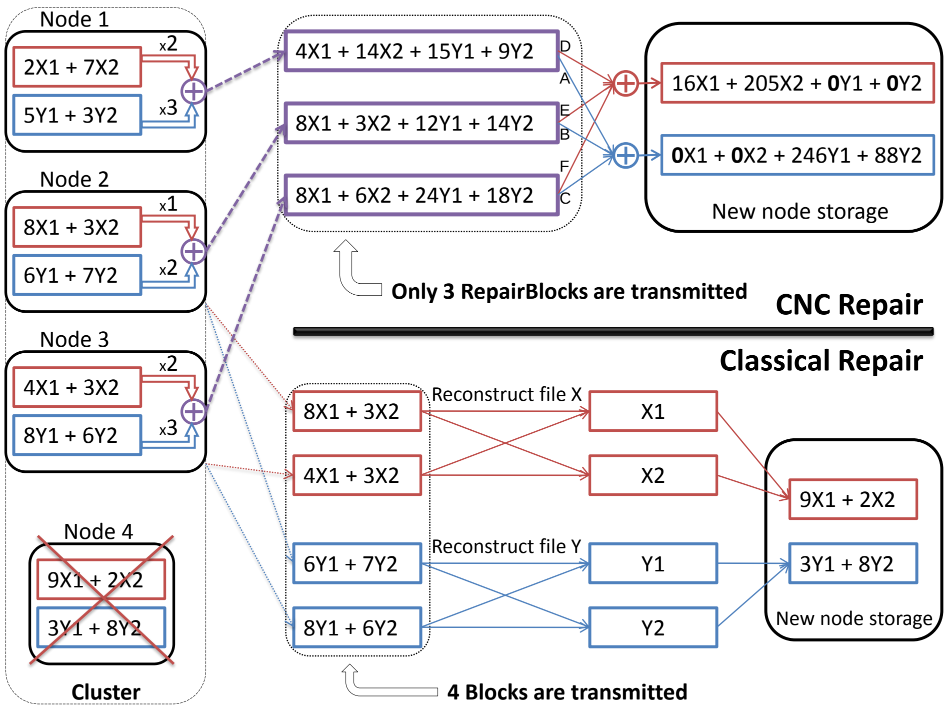


Figure 3.7: Comparison between CNC (top) and classical maintenance process (bottom), for the repair of a failed node which was storing two blocks of two different files (X & Y) in a cluster of 4 (with $k = 2, n = 4$). All stored blocks as well as transferred blocks and RepairBlocks in the example have exactly the same size.

To this end, the system is partitioned into disjoint (logical) clusters of n nodes, so that each node of the storage system belongs to one and only one cluster. Each file to be stored is encoded using random codes and is randomly associated to a single cluster, so as to balance the storage load on each cluster evenly. All blocks of a given file are then stored on the n nodes of the same cluster. In other words, CNC placement strategy consists in storing blocks of two different files belonging to the same cluster on the same set of nodes. An analytical evaluation of the mean time to data loss for this clustering placement can be found in [18]. Note that these clusters are constructed at a logical level. In practice, nodes of a given cluster may span geo-dispersed sites to provide an enhanced reliability, as illustrated on Figure 3.6. Obviously, there is a tradeoff between minimizing inter-site traffic and high reliability, this is outside the scope of this thesis. In such a setup, the storage system manager (*e.g.* the master node in the Google File System [37]) only needs to maintain two data structures: an index which maps each file to one cluster and an index by cluster which contains the set of the identifier of nodes in this cluster. This simple data placement scheme leads to significant data transfer gains and better load balancing, by clustering operations on encoded blocks, as explained in the remaining part of this section.

Maintenance of CNC

When a node failure is detected, the maintenance operation must ensure that all the blocks hosted on the faulty node are repaired in order to preserve the redundancy factor and hence the predefined reliability level of the system. Repair is usually performed at the granularity of a file. Yet, a node failure typically leads to the loss of several blocks, involving several files. This is precisely this characteristic that CNC leverages. Typically, when a node fails, multiple repairs are triggered, one for each particular block of one file that the faulty node was storing. Traditional approaches using erasure codes actually consider a failed node as the failure of all of its blocks. By contrast, the novelty of CNC is to leverage network coding at the node level, *i.e.* between multiple blocks of different files on a particular cluster. This is possible since CNC placement strategy clusters files so that all nodes of a cluster store the same files. This technical shift enables to significantly reduce the data to be transferred during the maintenance process.

An Illustrating Example

Before generalizing in the next section, we first describe a simple example (see Figure 3.7). This provides the intuition behind CNC while comparing it to a classical maintenance process. We consider two files X and Y of size $\mathcal{M} = 1024$ MB, encoded with random codes ($k = 2$, $n = 4$), stored on the 4 nodes of the same cluster (*i.e.* Nodes 1 to 4). File X is chunked into $k = 2$ chunks X_1, X_2 and file Y into chunks Y_1 and Y_2. Each node stores two encoded blocks, one related to X and the other Y, which are respectively a random linear combination of $\{X_1, X_2\}$ and $\{Y_1, Y_2\}$. Each block is of size $\frac{\mathcal{M}}{k} = 512$MB. Each node thus stores a total of $2 \times 512 = 1024$MB. We now consider the failure of Node 4.

In a classical repair process, the new node asks $k = 2$ nodes their blocks corresponding to file X and Y and downloads 4 blocks, for a total of $4 \times 512 = 2048$MB. This enables the new node to decode the two files independently, and then re-encode each file to regenerate the lost blocks of X and Y and store them.

Instead, CNC leverages the fact that the encoded blocks related to X and Y are stored on the same node and restored on the same new node to encode the files together rather than independently during the repair process. More precisely, if the nodes are able to compute a linear combination of their encoded blocks, we can prove that if $k = 2$, only 3 blocks are sufficient to perform the repair of the two files X and Y. Thus the transfer of only 3 blocks incurs the download of $3 \times 512 = 1536$MB, instead of the 2048MB needed with the classical repair process. In addition, this repair can be processed without decoding any of the two files. In practice, the new node has to contact the three remaining nodes to perform the repair. Each of the three nodes sends the new node a random linear combination of its two blocks with the associated coefficients. Note that the two files are now intermingled, *i.e.* encoded together. However, we want to be able to access each file independently after the repair. The challenge is thus to create two new random blocks, with the restrictions that one is only a random linear combination of the X blocks, and the other of the Y blocks. In this example, finding the appropriate coefficients in order to cancel the X_i or Y_i, comes down to solve for each file X and Y a system of two equations with three unknowns, respectively (A, B, C) (D, E, F) in Figure 3.7. The new node then makes two different linear combinations of the three received blocks according to the previously computed coefficients, (A=-6, B=-22, C=25) and (D=20, E=9, G=-17) in the example. Thereby it creates two new independent random blocks, related to file X and Y respectively. The repair is then performed, saving the bandwidth consumed by the transfer of one block *i.e.*, 512MB in this example. Note that the example is given over the integers for simplicity, though arithmetic operations would be computed over a finite field in a real implementation.

CNC: The General Case

We now generalize the previous example for any k. We first define a *RepairBlock* object: a RepairBlock is a random linear combination of two encoded blocks of two different files stored on a given node. RepairBlocks are transient objects which only exist during the maintenance process *i.e.*, RepairBlocks only transit on the network and are never stored permanently.

We are now able to formulate the core technical result of this section; the following proposition applies in a context where different files are encoded using random codes with the same k, and the encoded blocks are placed according to the cluster placement described in the previous section.

Proposition 1. *In order to repair two different files, downloading $k+1$ RepairBlocks from $k+1$ different nodes is a sufficient condition.*

Proof. The purpose of the proof is to show that a repair is correct, *i.e*, new encoded blocks are random blocks which belong to the initial vector space. Else, if blocks were random only in a subspace of the initial space, the system will progressively degenerate with the successive repairs. We start the proof with the following lemma.

Lemma 1. *A linear combination of independent random variables chosen uniformly in a finite field $\mathbb{F}_q$ also follows a uniform distribution over $\mathbb{F}_q$.*

Proof. Let $\mathbb{S}_N$ be the random variable defined by the linear combination of N random variables $\{X_1, X_2, ..., X_N\}$. These N random variables are independent and take their values uniformly in the finite field $\mathbb{F}_q$.

$$\mathbb{S}_N = \sum_{i=1}^{N} \alpha_i X_i$$

with $\forall i, X_i \in \mathbb{F}_q$, and $\alpha_i \in \mathbb{F}_q^*$
We show by recurrence that if $\forall i, Pr(X_i = x_i) = \frac{1}{q}$ then $Pr(\mathbb{S}_N = s_N) = \frac{1}{q}$
The case $N = 1$ is trivial. Let first show that for $N = 2$ the proposition is true.

$$\begin{aligned}
\mathbb{S}_2 &= \alpha_1 X_1 + \alpha_2 X_2 \\
Pr(\mathbb{S}_2 = s_2) &= Pr(\alpha_1 X_1 + \alpha_2 X_2 = s_2) \\
&= \sum_{x_1=0}^{q-1} Pr(X_1 = x_1) Pr(X_2 = \frac{s_2 - \alpha_1 x_1}{\alpha_2}) \\
&= \sum_{x_1=0}^{q-1} \frac{1}{q}\frac{1}{q} = q\frac{1}{q}\frac{1}{q} \\
&= \frac{1}{q}
\end{aligned}$$

The proposition is thus true for $N = 2$. We suppose that it is true for all N, and prove that it is true for $N + 1$.

$$\begin{aligned}
\mathbb{S}_{N+1} &= \mathbb{S}_N + \alpha_{N+1} X_{N+1} \\
Pr(\mathbb{S}_{N+1} = s_{N+1}) &= Pr(\mathbb{S}_N + \alpha_{N+1} X_{N+1} = s_{N+1}) \\
&= \sum_{x_{N+1}=0}^{q-1} [Pr(X_{N+1} = x_{N+1}) \\
&\times Pr(\mathbb{S}_N = s_{N+1} - \alpha_{N+1} x_{N+1})] \\
&= \sum_{x_{N+1}=0}^{q-1} \frac{1}{q}\frac{1}{q} = q\frac{1}{q}\frac{1}{q} \\
&= \frac{1}{q}
\end{aligned}$$

□

Definition 1. *A random vector $\vec{V}$ in a vector space $X = \text{span}\{X_1, X_2, ..., X_k\}$ where $X_i \in \mathbb{F}_q^l$ is defined as :*

$$V = \sum_{i=1}^{k} \alpha_i X_i$$

where the α_i coefficients are chosen uniformly at random in the field $\mathbb{F}_q$, ie. $Pr(\alpha_i = \alpha) = \frac{1}{q}, \forall \alpha \in \mathbb{F}_q$.

Let X be the vector space defined as $\text{span}\{X_1, X_2, ..., X_k\}$. Let Y be the vector space defined as $\text{span}\{Y_1, Y_2, ..., Y_k\}$. No assumptions are made on X_i and Y_i except that they are all in $\mathbb{F}_q^l$. In fact as X_i and Y_i are file blocks, it is not possible to ensure linear independence for example.

Let B_x^i be an encoded block of the file F_x stored on node i. B_x^i is a random linear combination of the $\{X_1, X_2, ..., X_k\}$, thus $B_x^i \in \text{span}\{X_1, X_2, ..., X_k\} = X$ which is a subspace of $\mathbb{F}_q^l$ of dimension $\text{Dim}(X) \leq k$.

Lemma 2. *$\forall \, \text{Dim}(X)$, B_x^i is a random vector in X.*

Proof. Let $\mathcal{B}$ be the largest family of linearly independent vectors of $\{X_1, X_2, ..., X_k\}$
$\forall l \mid X_l \notin \mathcal{B}, \exists! \{b_1^l, ..., b_j^l\}$ such that $X_l = \sum_{j|X_j \in \mathcal{B}} b_j^l X_j$

$$\begin{aligned} B_x^i &= \sum_{j=1}^{k} a_j^i X_j \\ &= \sum_{j|X_j \in \mathcal{B}} a_j^i X_j + \sum_{l|X_l \notin \mathcal{B}} a_l^i X_l \\ &= \sum_{j|X_j \in \mathcal{B}} a_j^i X_j + \sum_{l|X_l \notin \mathcal{B}} a_l^i \sum_{j|X_j \in \mathcal{B}} b_j^l X_j \\ &= \sum_{j|X_j \in \mathcal{B}} (a_j^i + \sum_{l|X_l \notin \mathcal{B}} a_l^i b_j^l) X_j \end{aligned}$$

From Lemma 1, all the coefficients of the linear combination are random over $\mathbb{F}_q$ thus B_x^i is a random vector in $\text{span}(\mathcal{B})$.
As $\text{span}(\mathcal{B}) = \text{span}\{X_1, X_2, ..., X_k\} = X$
Then B_x^i is a random vector in X. □

Let D^i be the random linear combination of two stored blocks by the node i with $i \in [1, k+1]$.

$$\begin{aligned} D^i &= \delta_x^i B_x^i + \delta_y^i B_y^i \\ &= \delta_x^i (\sum_{j=1}^{k} a_j^i X_j) + \delta_y^i (\sum_{l=1}^{k} a_l^i Y_l) \\ &= D_x^i + D_y^i \end{aligned}$$

By definition, $D^i \in \text{span}\{X_1, X_2, ..., X_k, Y_1, ..., Y_k\}$
$D_x^i = \sum_{j=1}^{k} \delta_x^i a_j^i X_j$
As δ_x^i are chosen randomly in $\mathbb{F}_q$, then from Lemma 1, D_x^i is a random vector in X.
Let's take a family $\{D_x^1, ..., D_x^{k+1}\}$.
As $\text{Dim}(X) \leq k$ it exists $\{\alpha_1, ..., \alpha_{k+1}\} \neq 0$ such that $\sum_{i=1}^{k+1} \alpha_i D_x^i = 0$
Thus :

$$\begin{aligned} \sum_{i=1}^{k+1} \alpha_i D^i &= \sum_{i=1}^{k+1} \alpha_i D_x^i + \sum_{i=1}^{k+1} \alpha_i D_y^i \\ &= \sum_{i=1}^{k+1} \alpha_i D_y^i \end{aligned}$$

As α_i are chosen independently with D_y^i then new vector is a random vector in Y. The reasoning is identical to get the new vector in X, thus completing the proof. □

This proposition implies that instead of having to download $2k$ blocks as with Reed-Solomon codes when repairing, CNC decreases that need to only $k+1$. Other implications and analysis are detailed in the next section. Note that the encoded blocks of the two files do not need to have the same size. In case of different sizes, the smallest is simply zero-padded during the network coding operations as usually done in this context; padding is then removed at the end of the repair process. In a real system, nodes usually store far more than two blocks, implying multiple iterations of the process previously described. More formally, to restore a failed node which was storing x blocks, the repair process must be iterated $\frac{x}{2}$ times. In fact, as two new blocks are repaired during each iteration, the number of iteration is halved compared to the classical repair process. Note that in case of an odd number of blocks stored, the repair process is iterated until only one block remains. The last block is repaired downloading k blocks of the corresponding file which are then randomly combined to conclude the repair. The overhead related to the repair of the last block in case of an odd block number vanishes with a growing number of blocks stored.

The fact that the repair process must be iterated several times can also be leveraged to balance the bandwidth load over all the nodes in the cluster. Only $k+1$ nodes over the n of the cluster are selected at each iteration of the repair process. As all nodes of the cluster have a symmetrical role, a different set of $k+1$ nodes can be selected at each iteration. In order to leverage the whole available bandwidth of the cluster, CNC makes use of a random selection of these $k+1$ nodes at each iteration. In other words, for each round of the repair process, the new node selects $k+1$ nodes uniformly at random over the n cluster nodes. Doing so, we show that every node is evenly loaded *i.e.*, each node sends the same number of RepairBlocks in expectation.

More formally, let N be the number of RepairBlocks sent by a given node. In a cluster where n nodes participate in the maintenance operation, for T iterations of the repair process, the average number of RepairBlocks sent by each node is :

$$E(N) = T\frac{k+1}{n} \tag{3.5}$$

Proof. During the repair process, the load on each node can be evaluated using a Balls-in-Bins model. Balls correspond to a block to be downloaded while bins represents the nodes which are storing the blocks. For each iteration of the repair protocol, k different nodes are selected to send a repair block. This corresponds to throwing k identical balls into n bins, with the constraints that once a bin has received a ball, it can not receive another ball at this round. In other words exactly k different bins are chosen at each round.

Lemma 3. *At each round i, the probability that a given bin has received one ball is* $\frac{k}{n}$

Proof. Let A be the event "the bin contains one ball at round i". Thus $\overline{A}$ corresponds to the event "the bin is empty at round i". $Pr(\overline{A})$ is computed as the number of ways to place the k balls inside the $n-1$ remaining bins, over all the possibilities to place the k balls into the n bins.

$$\begin{aligned} Pr(A) &= 1 - Pr(\overline{A}) \\ &= 1 - \frac{C_k^{n-1}}{C_k^n} \\ &= 1 - \frac{\frac{(n-1)!}{k!(n-1-k)!}}{\frac{n!}{k!(n-k)!}} \\ &= 1 - \frac{(n-k)!}{n(n-1-k)!} \\ &= 1 - \frac{(n-k)}{n} \\ &= \frac{k}{n} \end{aligned}$$

□

Let X be the number of balls into a given bin after t rounds. As the selection at each round are independent, the number of balls into a given bin follows a binomial law :
$X \sim \mathrm{B}(t,p)$ with $p = \frac{k}{n}$ (See Lemma 3) The expected value, denoted E(X), of the Binomial random variable X with parameters t and p is : $E(X) = tp = t\frac{k}{n}$ □

An example illustrating this load balancing is provided in the next section.

3.2.4 Analysis

The novel maintenance protocol proposed in the previous section enables *(i)* to significantly reduce the amount of data transferred during the repair process; *(ii)* to balance the load between the nodes of a cluster; *(iii)* to avoid computationally intensive decoding operations and finally *(iv)* to provide useful node reintegration. The benefits are detailed below.

Transfer Savings

A direct implication of Proposition 1 is that for large enough values of k, the data to transfer required to perform a repair is halved; this directly results in a better usage of available bandwidth. To repair two files in a classical repair process, the new node needs to download at least $2k$ blocks to be able to decode each of the two files. Then the ratio $\frac{k+1}{2k}$ (CNC over Reed-Solomon) tends to 1/2 as larger values of k are used.

The exact necessary amount of data $\sigma(x,k,s)$ to repair x blocks of size s encoded with the same k is given as follows:

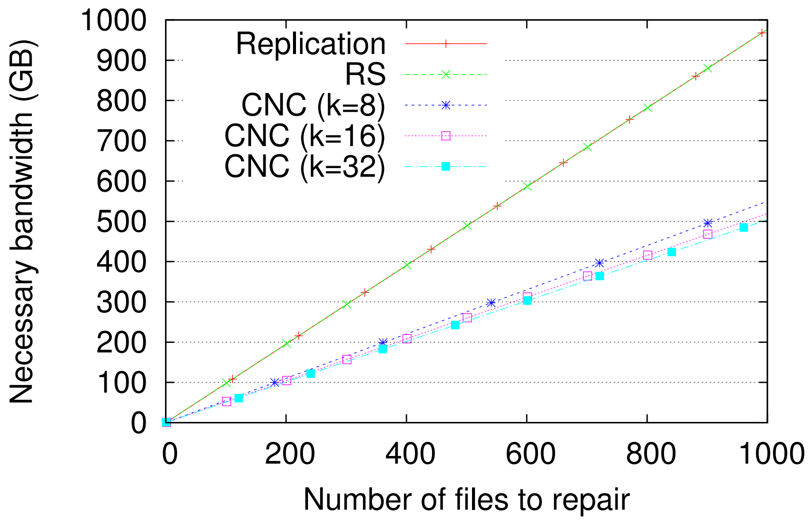


Figure 3.8: Necessary amount of data to transfer to repair a failed node, according to the selected redundancy scheme (files of 1 *GB each).*

$$\sigma(x,k,s) = \begin{cases} \frac{x}{2}s(k+1) & \text{if x is even} \\ \frac{x}{2}s(k+1+\frac{k-1}{x}) & \text{if x is odd} \end{cases}$$

An example of the transfer savings is given in Figure 3.8, for $k = 16$ and a file size of $1GB$.

From Proposition 1, CNC requires to repair lost files by groups of two. One can wonder whether there is a benefit in grouping more than two files during the repair. In fact a simple extension of Proposition 1 reveals that to group G files together, a sufficient condition is that the new node downloads $(G-1)k+1$ RepairBlocks from $(G-1)k+1$ distinct nodes over the n nodes of the cluster. Firstly, this implies that the new node must be able to contact many more nodes than $k+1$. Secondly, we can easily see that the gains made possible by CNC are maximal when grouping the file by two: savings in data transfer when repairing are expressed by the ratio $\frac{(G-1)k+1}{Gk}$. The minimal value of this ratio ($\frac{1}{2}$, which is equivalent to the maximal gain) is obtained for $G = 2$ and large value of k.

A second natural question is whether or not downloading fewer than $(G-1)k+1$ RepairBlocks to group G files together is possible. We can positively answer this question, as the value $(G-1)k+1$ is only a *sufficient* condition. In fact, if nodes do not send random combinations, but carefully choose the coefficients of the combination, it is theoretically possible to download less RepairBlocks. However, as G grows, finding such coefficients becomes computationally intractable, especially for large values of k. These coefficients can be found in some cases using *interference alignment* techniques (see for example [24]). Details of these techniques are outside the scope of this section

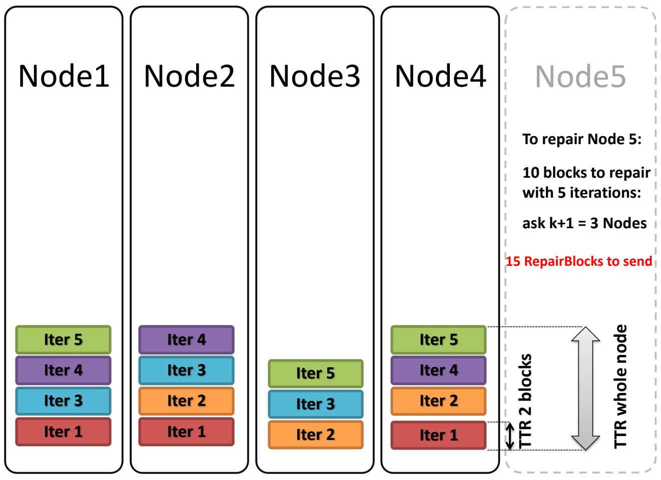


Figure 3.9: Natural load balancing for blocks queried when repairing a failed node (node 5*), for* 10 *blocks to restore.*

as no efficient algorithm is known to solve this problem to date. This then calls for the use of the simpler operation *i.e.*, $G = 2$ as we have presented in this section.

Load Balancing

As previously mentioned, when a node fails, the repair process is iterated as many times as needed to repair all lost blocks. CNC ensures that the load over remaining nodes is balanced during maintenance; Figure 3.9 illustrates this. This example involves a 5 node cluster, storing 10 different files encoded with random codes ($k = 2$). Node 5 has failed, involving the loss of 10 blocks of the 10 files stored on that cluster. Nodes 1 to 4 are available for the repair process.

CNC provides a load-balanced approach, inherent to the random selection of the $k+1 = 3$ nodes at each round. In addition, only $T = 5$ iterations of the repair process are necessary to recreate the 10 new blocks, as each iteration enables to repair 2 blocks at the same time. The total number of RepairBlocks sent during the whole maintenance is $T \times (k+1) = 15$, whereas the classical repair process needs to download 20 encoded blocks. The random selection ensures in addition that the load is evenly balanced between the available nodes of the cluster. Here, nodes 1,2 and 4 are selected during the first repair round, then nodes 2, 3 and 4 during the second round and so forth. The total number of RepairBlocks is balanced between all available nodes, each sending $\frac{T\times(k+1)}{n} = \frac{15}{4} = 3.75$ RepairBlocks on average. As a consequence of using the whole available bandwidth in parallel, and as opposed to sequentially fetching blocks for only a subset of nodes, the Time To Repair (TTR) a failed node is also greatly reduced. This is confirmed experimentally in Section 3.2.5.

No Decoding Operations

Decoding operations are known to be time consuming and therefore should be done only in case of file accesses. While the use of classical erasure codes requires such decoding to take place upon repair, CNC avoids those cost-intensive operations. In fact, no file needs to be decoded at any time in CNC: repairing two blocks only requires to compute two linear combinations instead of decoding the two files. However the output of our repair process is strictly equivalent to the situation where the files are decoded. This greatly simplifies the repair process over classical approaches. As a consequence, the time to perform a repair is reduced by orders of magnitude compared to the classical reparation process, especially when dealing with large files as confirmed by our experiments (Section 3.2.5).

Reintegration

The decision to declare a node as faulty is usually performed using timeouts; this is typically an error prone decision [20]. In fact, nodes can be wrongfully timed-out and can reconnect once the repair is done. While the longer the timeouts, the fewer errors are made, adopting large timeouts may jeopardize the reliability guarantees, typically in the event of burst of failures. The interest of reintegration is to be able to leverage the fact that nodes which have been wrongfully timed-out are reintegrated in the system. Authors in [20] showed that reintegration of replicas was a key concept to save maintenance bandwidth. However, reintegration has not been addressed when using erasure codes.

As previously mentioned, when using classical erasure codes, the repaired blocks have to be strictly identical to the lost ones. Therefore reintegrating a node which was suspected as faulty in the system is almost useless since this results in two identical copies of the lost and the repaired blocks. Such blocks can only be useful in the event of the failure of two specific nodes, the incorrectly timed-out node and the new one. Instead, reintegration is always useful when deploying CNC. More precisely, every single new block can be leveraged to compensate for the loss of any other block and therefore is useful in the event of the failure of any node. Indeed, new created blocks are simply new random blocks, thus different from the lost ones while being functionally equivalent. Therefore each new block contributes to the redundancy factor of the cluster. Assume that a node which has been incorrectly declared as faulty returns into the system. A repair has been performed to sustain the redundancy factor while it turned out not to be necessary. This only means that the system is now one repair process ahead and can leverage this unnecessary repair to avoid triggering a new instance of the repair protocol when the next failure occurs.

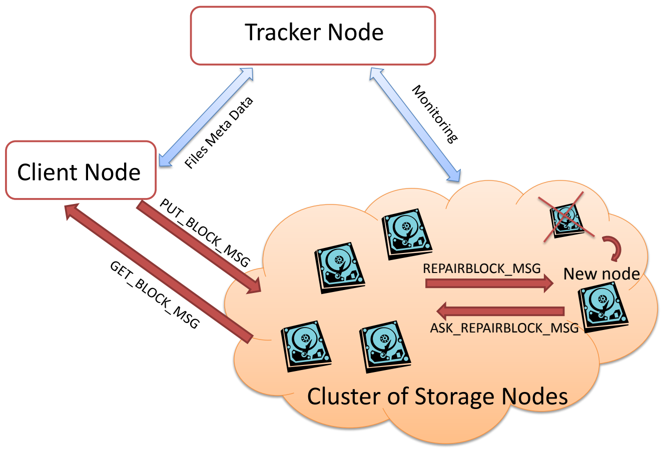


Figure 3.10: System overview

3.2.5 Evaluation

In order to confirm the theoretical savings provided by the CNC repair protocol, in terms of bandwidth utilization and decoding operations, we deployed CNC over a public experimental platform. We describe the implementation of the system and CNC experimental results in this section.

System Overview

We implemented a simple storage cluster with an architecture similar to Hadoop [77] or the Google File System [37]. This architecture is composed of one *tracker node* that manages the metadata of files, and several *storage nodes* that store the data. This set of storage nodes forms a cluster as defined in Section 3.2.3. The overview of the system architecture is depicted in Figure 3.10. Client nodes can PUT/GET the data directly to the storage nodes, after having obtained their IP addresses from the tracker. In case of a storage node failure, the tracker initiates the repair process and schedules the repair jobs. All files to be stored in the system are encoded using random codes with the same k. Let n be the number of storage nodes in the cluster, then n encoded blocks are created for each file, one for each storage node. Remind that the system can thus tolerate $n - k$ storage node failures before files are lost for good.

PUT/GET and Maintenance Operations In the case of a PUT operation, the client first encodes blocks. The coefficients of the linear combination associated with each encoded block are appended at the beginning of the block. Those n encoded blocks are sent to the n storage nodes of the cluster using a PUT_BLOCK_MSG. A PUT_BLOCK_MSG contains the encoded information, as well as the hash of the corresponding file. Upon receipt of a PUT_BLOCK_MSG, the storage node stores the encoded block using the hash as filename. To retrieve the file, the client sends a GET_BLOCK_MSG to at least k nodes, out of the n nodes of the cluster. A GET_BLOCK_MSG only contains the hash of the file to be retrieved. Upon receipt of a GET_BLOCK_MSG the storage node sends the block corresponding to the given hash. As soon as the client has received k blocks, the file can be recovered.

In case of a storage node failure, a new node is selected by the tracker to replace the failed one. This new node sends a ASK_REPAIRBLOCK_MSG to $k + 1$ storage nodes. An ASK_REPAIRBLOCK_MSG contains the two hashes of the two blocks which have to be combined following the repair protocol described in Section 3.2.3. Upon receipt of an ASK_REPAIRBLOCK_MSG, the storage node combines the two encoded blocks corresponding to the two hashes, and sends the resulting block back to the new node. As soon as $k + 1$ blocks are received, the new node can regenerate two lost blocks. This process is iterated until all lost blocks are repaired.

Deployment and Results

We deployed the system previously described on a public testbed, namely GRID'5000. The GRID'5000 project aims at building an experimental Grid platform gathering less than a dozen of geographically distributed sites in France combining up to 5000 processors with a certain level of heterogeneity both in terms of processor and network types. Our experiments ran on 33 nodes connected with a 1GB network. Each node has 2Intel Xeon L5420 CPUs 2.5 GHz, 32GB RAM and a 320GB hard drive. We randomly chose 32 storage nodes to form a cluster, as defined in Section 3.2.3. The last remaining node was elected as the tracker. All files were encoded with $k = 16$, and we assumed that the size of each inserted file is 1GB. This size is used in Windows Azure Storage for sealed extents which are erasure coded [17].

Scenario

In order to evaluate our maintenance protocol, we implemented a first phase of i files insertion in the cluster, and artificially triggered a repair during the second phase. According to the protocol previously described, the tracker selects a new node to replace the faulty one, to which it sends the list of IP addresses of the storage nodes. The new node then directly asks RepairBlocks to storage nodes, without any intervention of the tracker, until it recovers as many encoded blocks as the faulty node was storing. We measured the time to repair a faulty node depending on the number of blocks it was hosting. The time to repair is defined as the time between the reception of the list of IPs, and the time all new encoded blocks are effectively stored on the new node. We compared CNC against a classical maintenance mechanism (called RS), which would be used with Reed-Solomon codes as described in Section 1.2.4 and with standard replication. All the presented results are averaged on three independent experiments. This small number of experiments can be explained by the fact that

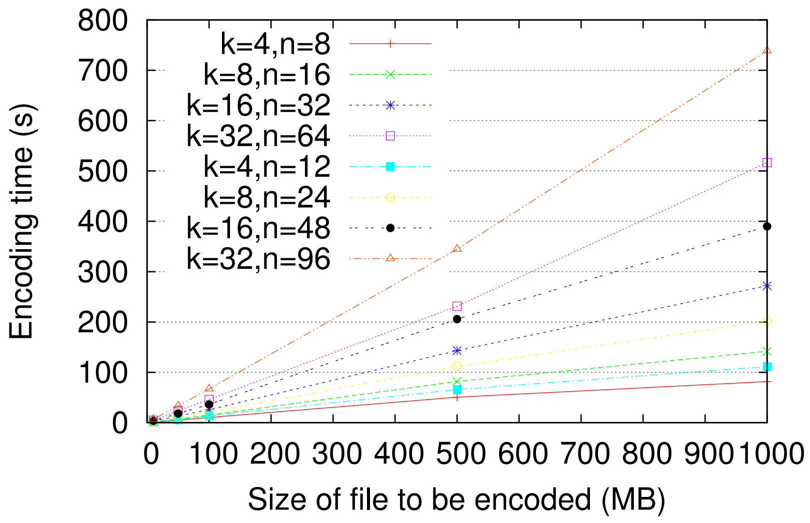


Figure 3.11: Encoding time depending on file size when using random codes.

our experimental testbed enables to make a reservation on a whole cluster of nodes in isolation ensuring that experiments are highly reproducible and we observed a standard deviation under 2 seconds for all values.

Coding

We developed a Java library to deal with arithmetic operations over a finite field. In this experiment, arithmetic operations are performed over a finite field with 2^{16} elements as it enables to treat data as a stream of unsigned short integers (16 bits). Additions and subtractions correspond to XOR operations between two elements. Multiplications and divisions are performed in the logspace using lookup tables which are computed offline. This library enabled us to implement classical matrix operations over finite fields, such as linear combinations, encoding and decoding of files.

We measure the encoding time when using random codes for various code rates, depending on the size of the file to be encoded. Results are depicted on Figure 3.11. We show that for a given (k, n) the encoding time is clearly linear with the file size. For example with ($k = 16$,$n = 32$) the encoding time for a file of size 512MB and 1GB are respectively 143 and 272 seconds. In addition, the encoding time increases with k and with the code rate, as more encoded blocks have to be created. For instance, a file of 1GB with $k = 16$ is encoded in 272 seconds for a code rate $1/2$ ($n = 32$), whereas 390 seconds are necessary for a code rate $1/3$ ($n = 48$).

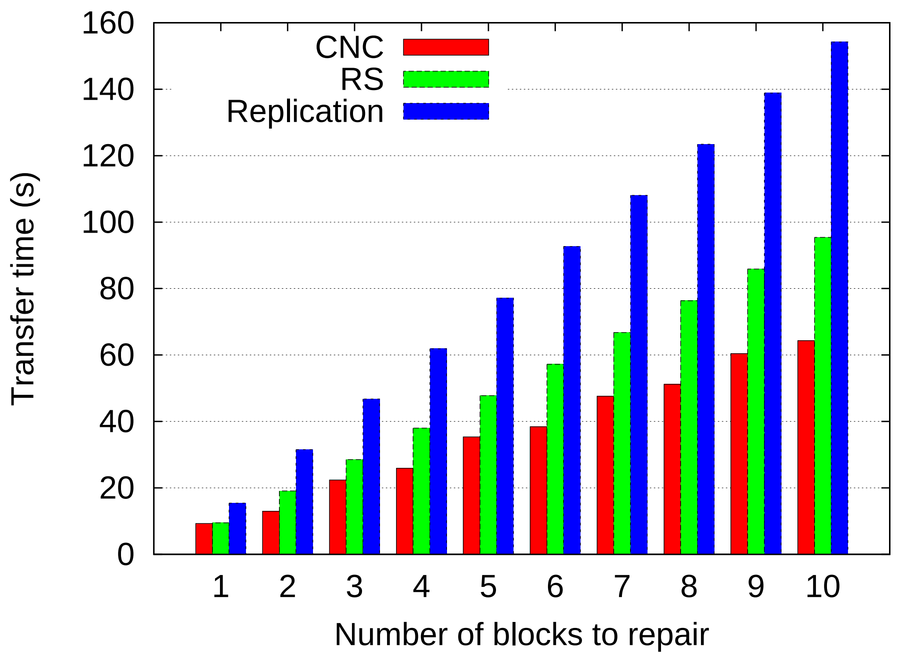


Figure 3.12: Time to transfer the necessary quantity of data to perform a complete repair

Transfer Time

In this experiment, we evaluated the time to transfer the whole quantity of data needed to perform a complete repair for CNC, RS and replication, depending on the number of blocks to be repaired. In order to quantify the gains provided by CNC in isolation, we disabled the load balancing part of the protocol in this experiment. In other words, the same set of nodes is selected for all iterations of the repair process. The results are depicted on Figure 4.8.

Firstly, we observe that CNC consistently outperforms the two alternative mechanisms. As CNC incurs the transfer of a much smaller amount of data, the time to transfer the blocks during the repair process is greatly reduced compared to both RS and replication. For instance, to download the necessary quantity of data to repair a node which was hosting 10 blocks related to 10 different files, CNC only requires 64 seconds whereas RS and replication requires respectively 95 and 154 seconds on average. It should also be noted that no coding operations are done in this experiment, except for CNC as nodes have to compute a random linear combination of their encoded blocks to create a RepairBlock before sending it. This time is taken into account, thus explaining why the transfer time for CNC is not exactly halved compared to RS.

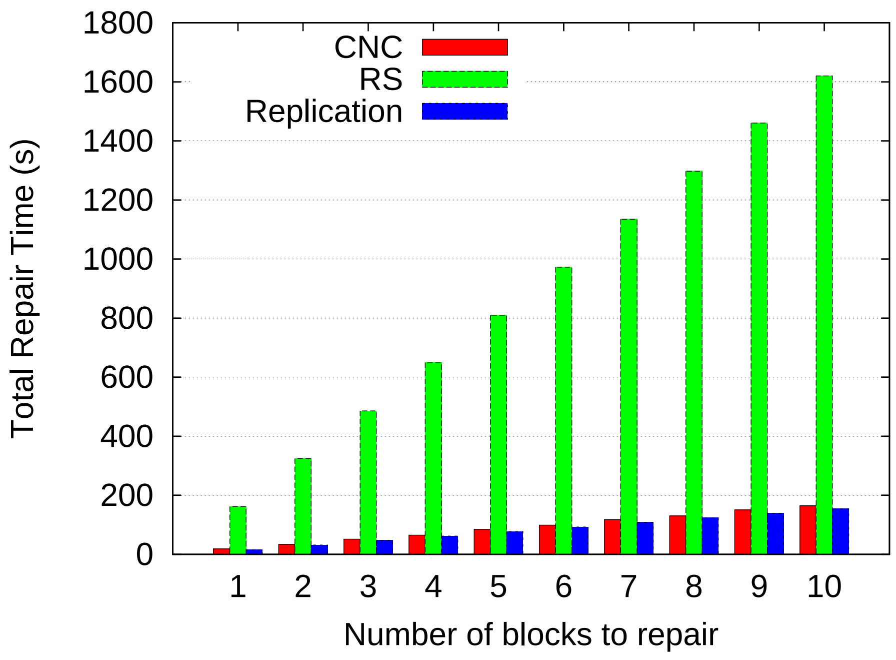


Figure 3.13: Total repair time

A second observation is that CNC also scales better with the number of files to be repaired. As opposed to CNC, both RS and replication involve transfer times for multiple files which are strictly proportional to the time to transfer a single file. For example RS and CNC requires 9 seconds to download a single file, but RS requires 95 seconds to download 10 files, while CNC only requires 64 seconds for the same operation.

Finally, replication leads to the highest time to transfer. This is mainly due to the fact that replication does not leverage parallel downloads from several nodes as opposed to CNC and RS. Yet replication does not suffer from computational costs, which can dramatically increase the whole repair time of a failure as shown in the next section.

Repair Time

In this experiment, we measured the total repair time of a failure, depending on the number of blocks (related to different files) the faulty node was storing. The results, depicted on Figure 4.9, include both the transfer times, evaluated in the previous section, as well as coding times. Thereby it represents the effective time between a failure is detected and the time it has been fully recovered. As replication does not incur any coding operations, the time to repair is simply the time to transfer the files.

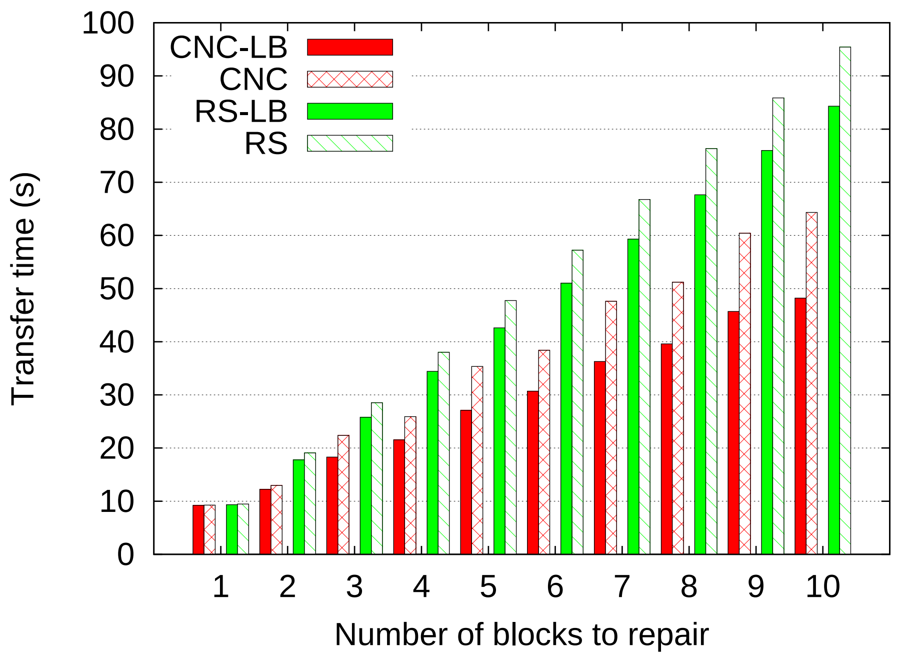


Figure 3.14: Impact of load balancing on Time To Transfer

Note that for the sake of fairness, we enable the load balancing mechanism both for CNC and RS.

Figure 4.9 shows that the repair time is dramatically reduced when using CNC compared to RS, especially with an increasing number of files to be repaired. For instance to repair a node hosting 10 blocks related to 10 different files, CNC and replication require respectively 165 and 154 seconds while RS needs 1620 seconds on average. These time savings are mainly due to the fact that decoding operations are avoided in CNC. In fact, the transfer time is almost negligible compared to the computational time for RS. The transfer time only represents 6% of the time to repair a node hosting 10 blocks related to 10 different files with RS. This clearly emphasizes the interest of avoiding computationally intensive tasks such as decoding during the maintenance process.

We also observe that time to repair a failure with CNC is nearly equivalent to the one needed when using replication. As shown in Figure 4.8, replication transfer times are much higher than CNC ones, but this is counter-balanced by the fact that some coding operations are necessary in CNC. In other words, CNC saves time compared to replication during the data transfer, but these savings are cancelled out due to linear combination computations. Finally our experiments show that, as opposed to RS, CNC scales as well as replication with the number of files to be repaired.

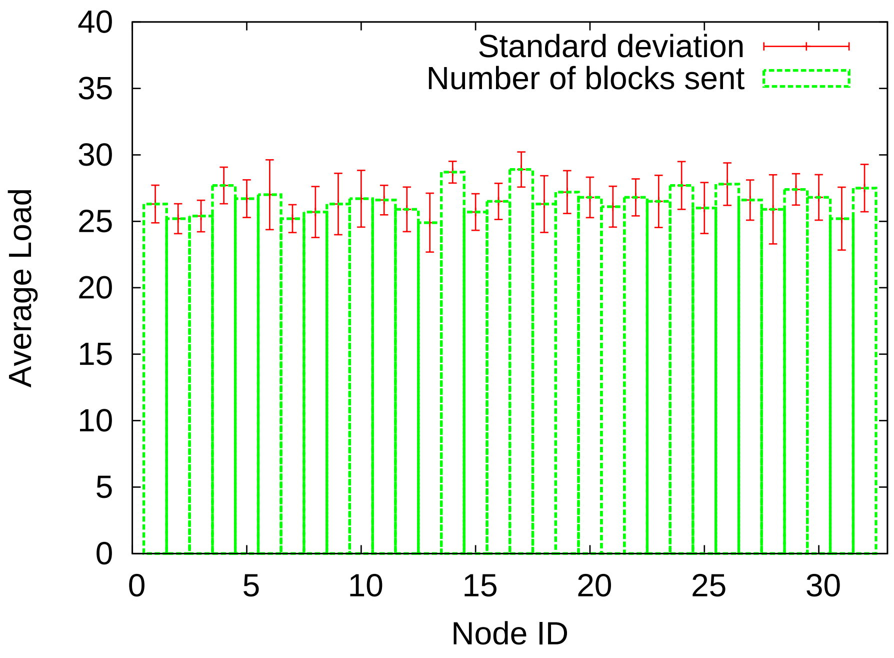


Figure 3.15: Average load on each node in the cluster

Load Balancing

As shown in Section 3.2.3, CNC provides a natural load balancing feature. The random selection of nodes from which to download blocks during the maintenance process ensures that the load is evenly balanced between nodes. In this section, we evaluate the impact of this load balancing on the transfer time for both CNC and RS.

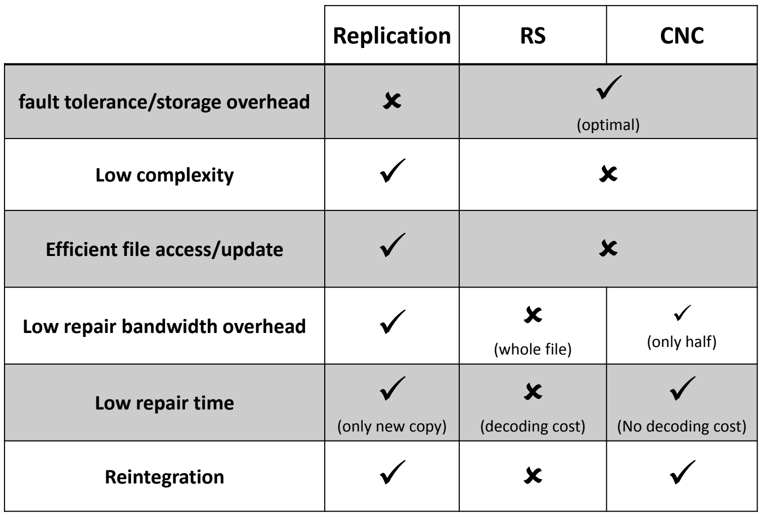
Figure 3.14 depicts the transfer time for both RS and CNC depending on the number of files to be repaired. We compare the transfer time between the load balanced approach (CNC-LB and RS-LB), and its counterpart which involves a fixed set of nodes, as done in Section 3.2.5. Results show that transfer times are reduced when load balancing is enabled, as the whole available bandwidth can be leveraged. In addition, time savings due to the load balance increases as more files have to repaired.

Figure 3.15 shows the number of blocks sent by each of the 32 nodes of the cluster for a repair of a node which was storing 100 blocks when using CNC. This involves 50 iterations of the protocol, where at each iteration, $k + 1 = 17$ distinct nodes send a RepairBlock. We observe that all nodes send a similar number of blocks *i.e.*, nearly 26, in expectation. This is consistent with the expected value analytically computed, according to Equation 3.5 as $\frac{25\times 17}{32} = 26.5625$.

Summary

While erasure codes, typically Reed-Solomon, have been acknowledged as a sound alternative to plain replication in the context of reliable distributed storage systems, they suffer from high costs, both bandwidth and computationally-wise, upon node repair. This is due to the fact that for each lost block, it is necessary to download enough blocks of the corresponding file and decode the entire file before repairing.

In this section, we address this issue and provide a novel code-based system providing high reliability and efficient maintenance for practical distributed storage systems. The originality of our approach, CNC, stems from a clever cluster-based placement strategy, assigning a set of files to a specific cluster of nodes combined with the use of random codes and network coding at the granularity of several files. CNC leverages network coding and the co-location of blocks of several files to encode files together during the repair. This provides a significant decrease of the bandwidth required during repair, avoids file decoding and provides useful node reintegration. We provide a theoretical analysis of CNC. We also implemented CNC and deployed it on a public testbed. Our evaluation shows dramatic improvement of CNC with respect to bandwidth consumption and repair time over both plain replication and Reed-Solomon-based approaches. Table 3.16 summarizes the properties of CNC and existing redundancy mechanisms, Replication and Reed-Solomon Codes (RS), clearly conveying the benefits of CNC.



	Replication	RS	CNC
fault tolerance/storage overhead	✗	✓ (optimal)	
Low complexity	✓	✗	
Efficient file access/update	✓	✗	
Low repair bandwidth overhead	✓	✗ (whole file)	✓ (only half)
Low repair time	✓ (only new copy)	✗ (decoding cost)	✓ (No decoding cost)
Reintegration	✓	✗	✓

Figure 3.16: Comparison of CNC with most implemented redundancy mechanisms, i.e. *Replication, and Reed-Solomon codes (RS).*

Chapter 4

Conclusions & Perspectives

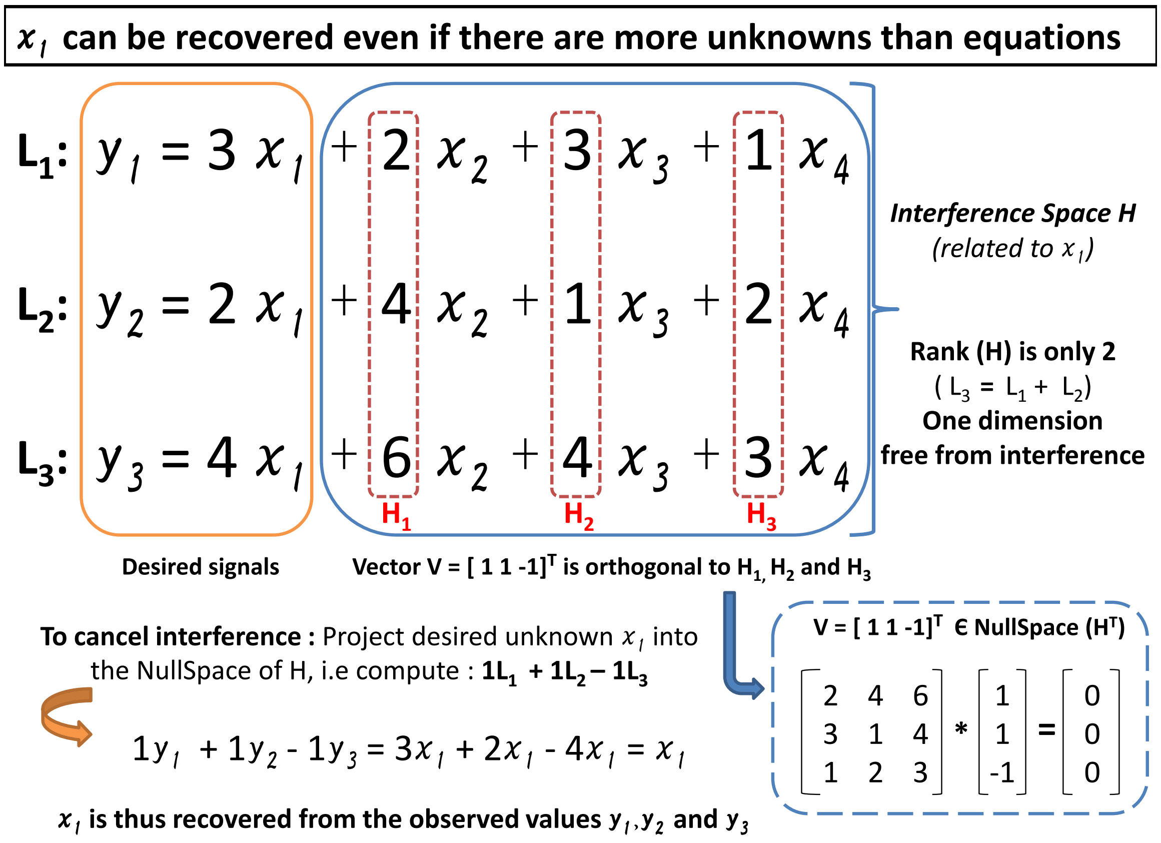
In the past two chapters, we have explored how to optimize the bandwidth of a distributed storage system. In the first one, we showed how the available bandwidth of peer-to-peer storage systems can be increased, by leveraging the presence of gateways inside the network topology. In the second one, we presented two mechanisms aiming at reducing the bandwidth consumption during the maintenance process. First, we proposed an adaptive and per-node timeout in order to decrease the number of repairs wrongfully triggered. Second, we presented a maintenance protocol which enables to repair multiple files jointly, thus factorizing the bandwidth costs. However, the design of this protocol has raised interesting questions in terms of theoretical as well as practical research directions. We briefly discuss these two aspects in the following sections.

4.1 Going further with interference alignment

In Section 3.2.3, we showed that, instead of contacting k nodes during the repair process, contacting $k+1$ nodes enables to repair **two** blocks jointly. This directly translates into a halved bandwidth consumption compared to classical reparation.

A natural question would be: is it possible to achieve an higher bandwidth reduction if contacting more than $k+1$ is allowed. For example, is it possible to repair **three** blocks jointly when contacting $k+2$ nodes. Note that, instead of halving the necessary bandwidth as proposed in Section 3.2.3 , repairing three blocks jointly would result in a bandwidth consumption divided by three.

In this section, we show that it is actually possible and provide some examples where nodes do not send random combinations of their encoded blocks, but carefully choose the coefficients of the combination. We experimentally show that for small k



x_1 can be recovered even if there are more unknowns than equations
L1: $y_1 = 3x_1 + 2x_2 + 3x_3 + 1x_4$
L2: $y_2 = 2x_1 + 4x_2 + 1x_3 + 2x_4$
L3: $y_3 = 4x_1 + 6x_2 + 4x_3 + 3x_4$
H1
H2
H3
Interference Space H (related to x_1)
Rank (H) is only 2
($L_3 = L_1 + L_2$)
One dimension free from interference
Desired signals
Vector V = [1 1 -1]^T is orthogonal to H_1, H_2 and H_3
V = [1 1 -1]^T ∈ NullSpace (H^T)
$\begin{bmatrix} 2 & 4 & 6 \\ 3 & 1 & 4 \\ 1 & 2 & 3 \end{bmatrix} * \begin{bmatrix} 1 \\ 1 \\ -1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$
To cancel interference : Project desired unknown x_1 into the NullSpace of H, i.e compute : $1L_1 + 1L_2 - 1L_3$
$1y_1 + 1y_2 - 1y_3 = 3x_1 + 2x_1 - 4x_1 = x_1$
x_1 is thus recovered from the observed values y_1, y_2 and y_3

Figure 4.1: Example of an underdetermined linear system i.e., *there are more unknowns than equations. However some unknown can be recovered since interference are aligned into a smaller subspace.*

values, it is possible to find such coefficients using *interference alignment* techniques (see for example [24]).

Interference alignment

Interference alignment is a radical idea that has recently emerged out of the study of the capacity of interference networks [16]. The origins of interference alignment lie in elementary linear algebra, we briefly provide the intuition of this promising technique with the following example. Assume a simple system of three linear equations. There are three observed values, y_1, y_2 and y_3, and four unknowns x_1, x_2, x_3 and x_4.

$$\begin{cases} y_1 = 3x_1 + 2x_2 + 3x_3 + 1x_4 \\ y_2 = 2x_1 + 4x_2 + 1x_3 + 2x_4 \\ y_3 = 4x_1 + 6x_2 + 4x_3 + 3x_4 \end{cases}$$

As there are more unknowns than equations, it is usually impossible to recover the values of all the unknowns x_1, x_2, x_3 and x_4. However in this example, it is actually possible to recover the value of x_1. In fact, all interference terms related to x_1, *i.e.*, x_2, x_3 and x_4, are aligned in a smaller subspace. In other words, it exists a linear dependence between the interference vectors H_1, H_2 and H_3 (see Figure 4.1). If the interfering space does not span the whole space, but only a two-dimensional, over the three available (the number of equations), this leaves one dimension free from interference. x_1 can thus be projected into this interference-free space to be recovered. For example, note that the vector $V = [1, 1, -1]^T$ is orthogonal to every interference vector H_1, H_2 and H_3. Applying the row operations $1L_1 + 1L_2 - 1L_3$ leads to a single equation with a single unknown x_1 and can thus be solved.

This brings to light why this technique is called *interference alignment*, as interference vectors have to be aligned in a subspace of the whole space in order to let some dimension interference free. We next show how it can be used for the repair problem in distributed storage system.

Exhaustive search of solutions

Applying interference alignment in CNC involves that, instead of sending a random linear combination of the stored blocks, nodes have to carefully choose the coefficients of the combination. We show a simple example in order to illustrate this technique.

Assume that three files X, Y and Z are stored on a five-node cluster, and each file is encoded with $k = 2$ (see Figure 4.2). One node fails, thus leaving four nodes available for the repair process. Each node sends a (non-random) linear combination of its three encoded blocks. A proper choice of the coefficients of this linear combination would let possible to recover three new encoded blocks, one for each file X, Y and Z. For example, the coefficients of the linear combination sent by node 1 is the *alignment vector* $(251, 26, 134)$. The challenge of finding *good* coefficients relies on the fact that :

- To repair the block related to file X, all the terms related to file Y **and** file Z must be cancelled.
- To repair the block related to file Y, all the terms related to file X **and** file Z must be cancelled.
- To repair the block related to file Z, all the terms related to file X **and** file Y must be cancelled.

More formally, this means that each node has to send a linear combination such that each interference matrix H_X, H_Y and H_Z must **not** be full rank[1] (see Figure 4.3). Thus, multiple alignment constraints have to be satisfied to be able to cancel all the interference terms. Note that these multiple alignment constraints make this problem

[1]It exists a linear dependence between the columns (or rows) vectors of the matrix.

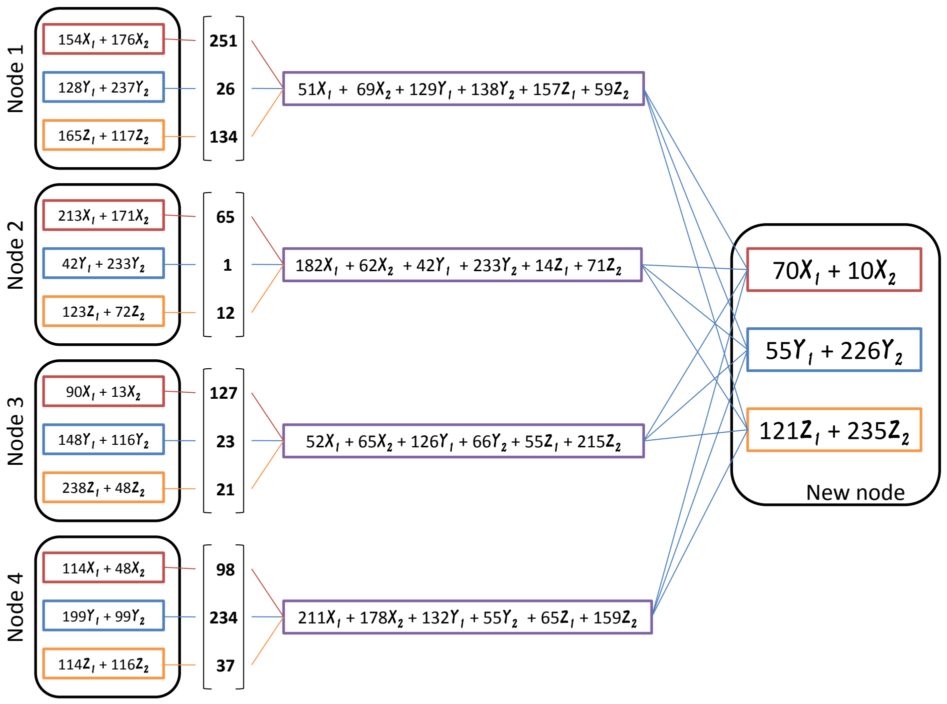


Figure 4.2: Example of a repair process combining three files X, Y and Z using interference alignment. Each node carefully chooses the coefficients of the linear combination in order to align interference. One new block for each file can thus be recovered without interference.

much harder than when grouping files by two. Indeed, the fact that theses conditions must be satisfied **at the same time** is challenging and boils down to solve a set of multivariate polynomial equations over a finite field. This problem is not linear any more, and is known to be, in general, an NP-hard one [35].

In order to check if it was possible to find such coefficients in the previous example, we performed an exhaustive search among all alignment vectors in $\mathbb{F}_{256}$. In other words, for each alignment vector, and for all possible values of these vectors, we checked if the multiple alignment constraints could be satisfied[2]. We found solutions for $k = 2$, $k = 3$ and $k = 4$. We provide an example of the coefficients for a complete repair for $k = 2$ in Appendix. However, for higher k, performing an exhaustive search becomes computationally intractable, especially in large finite fields.

To summarize, from a theoretical point of view, interference alignment is a promising technique to extend the CNC repair mechanism. As explained above, it would enable to further reduce the bandwidth consumption during the repair process. However,

[2]Note that the NullSpace of each interference matrix must also be distinct to perform a valid reparation

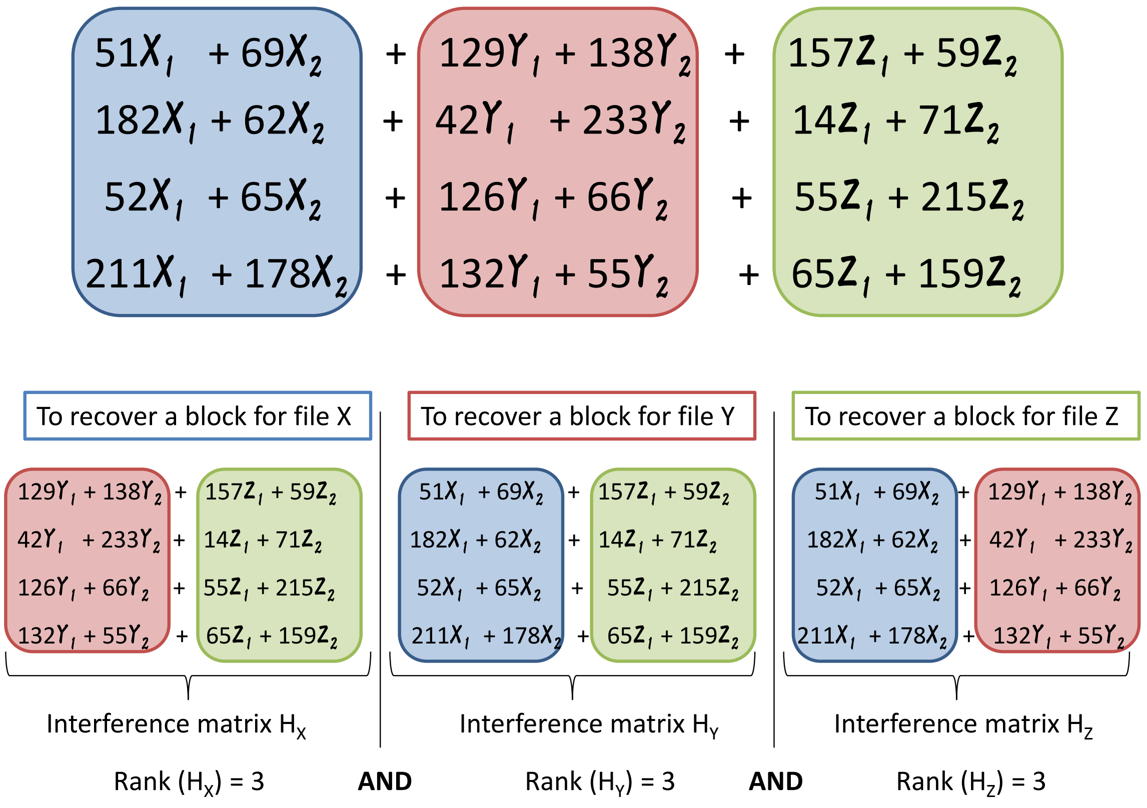


Figure 4.3: Each interference matrix must ***not*** *be full rank in order to be able to recover one block for each file without interference.*

from a practical point of view, finding *good* coefficients to align interference terms is a challenging problem (see for example [24]). In addition, the consequences of contacting a larger number of nodes during the repair process remain unclear in practical systems. We discuss this point in the next section.

4.2 Practical issues

While various repair mechanisms has been proposed in the literature, it remains a huge gap between their design, and the performances of their implementation in real systems. First of all, most repair models rely on a very strong assumption: they assume that nodes never fail or disconnect during a repair. This is obviously hard to ensure in a real system. Studying the impact of these failures on the repair bandwidth overhead has rarely been tackled yet. To the best of our knowledge, the only technical report [67] focusing on repair performances, performs it via simulation.

In addition, numerous repair mechanisms leverage the possibility to contact a larger number of nodes in order to decrease the repair bandwidth consumption. For example, in CNC we contact $k + 1$ nodes instead of k to perform the repair. However the consequences of contacting more nodes during the repair process are unclear in practice. Intuitively, the more nodes involved in the repair process, the higher the probability that one of them fails during the repair. Moreover, being able to contact a larger number of nodes may not be possible, typically in peer-to-peer systems. Indeed, even optimal repair mechanisms may become inefficient due to churn. To illustrate our purpose, we provide a simple example of the impact of node availability when using *regenerating codes*.[3]

Impact of nodes availability on the repair bandwidth

As already said, an important point is to make the difference between the number of nodes which are effectively failed, and the number of nodes a newcomer can contact at the moment of the reparation. Typically, in a peer-to-peer storage system, nodes can simply be temporary disconnected from the system.

Assume a $1GB$ file is stored using regenerating codes with $k = 16$ on $n = 32$ nodes, such that any 16 out of these 32 nodes suffice to recover the file.

If a failure occurs, it is showed in [23] that it is possible to repair the failed node downloading only $121MB$ when the newcomer can contact all the 31 remaining nodes (ie. $d = n - 1$). This is an impressive bandwidth reduction. Note that with classical erasure codes the quantity transferred would be the entire file size, *i.e.*, $1GB$. Thus there is nearly 88% of bandwidth savings compared to the necessary bandwidth when using classical erasure codes. However on a typical peer-to-peer system, with medium availability (peers are online only half of the day for example), the probability that a newcomer can contact all these 31 nodes is nearly zero (7.45×10^-9). This means that on the one hand, when accessing to extra nodes, it is possible to greatly reduce the quantity of data transferred during a repair, but on the other hand the probability to access these extra nodes and thus to succeed the repair must also be taken into account in the evaluation. Thereby we argue that the exact improvement of regenerating codes[4] in a practical system has to be evaluated based on the average quantity of data transferred during a repair.

Hereafter we describe how to compute the expected value of the amount of data transferred during a repair, depending on the code used (*i.e.,* the value of k and n) and the availability of nodes participating in this repair. Let D be the discrete random variable representing the number of nodes a newcomer can contact to be repaired. Let $P_d = P(D = d)$ with $k \leq d \leq n - 1$. In fact, if there is only one failure, the newcomer can contact at most all the $n-1$ surviving nodes and if $d < k$ it is impossible to repair as the data is not available (We only consider MDS codes). If we assume homogeneous

[3]Regenerating codes are optimal in terms of the repair bandwidth consumption.

[4]As well as every repair mechanisms requiring to contact a larger number of nodes

and independent availability of nodes $\bar{p}$, thus the probability that the newcomer can contact exactly d nodes is given by :

$$P_d = C_n^d (\bar{p})^d (1 - \bar{p})^{n-d} \tag{4.1}$$

Let X_d be the quantity of data transferred when the newcomer can contact $D = d$ nodes. Then (see [23]):

$$X_d = \frac{Md}{k(d - k + 1)} \tag{4.2}$$

Let X be the discrete random variable representing the amount of data transferred during a repair. It takes values X_d with probabilities P_d respectively with $d \in [k, n-1]$. Then the expected value of this random variable is

$$E[X] = \frac{1}{\sigma} \sum_{d=k}^{n-1} P_d X_d \tag{4.3}$$

where $\sigma = \sum_{d=k}^{n-1} P_d$ normalizes the distribution (to be a valid distribution, $\frac{1}{\sigma} \sum_{d=k}^{n-1} P_d = 1$ must be satisfied).

We plot on Figure 4.4 and 4.5 the average amount of data transferred during a repair depending on the peers availability for various k, and various storage overheads. Figure 4.4 and 4.5 clearly show that the gain provided when using regenerating codes highly depends on the availability of nodes as well as the storage overhead. For example, to repair a file of $1GB$ with $(k = 16, n = 32)$ when the mean availability of peers is 0.5, the average quantity of data to transfer during the repair is nearly $500MB$. This is much higher than the optimal value *i.e.,* $121MB$ as explained above. This optimal value can only be achieved if nodes are always available. Relaxing this assumption on the peer availability has thus a strong impact on the real quantity of data to be transferred. The variety of such assumptions, as well as their practical consequences, call for the design of repair mechanisms tailor-made for networked storage systems, looking at adapted strategies to carry out the redundancy replenishment.

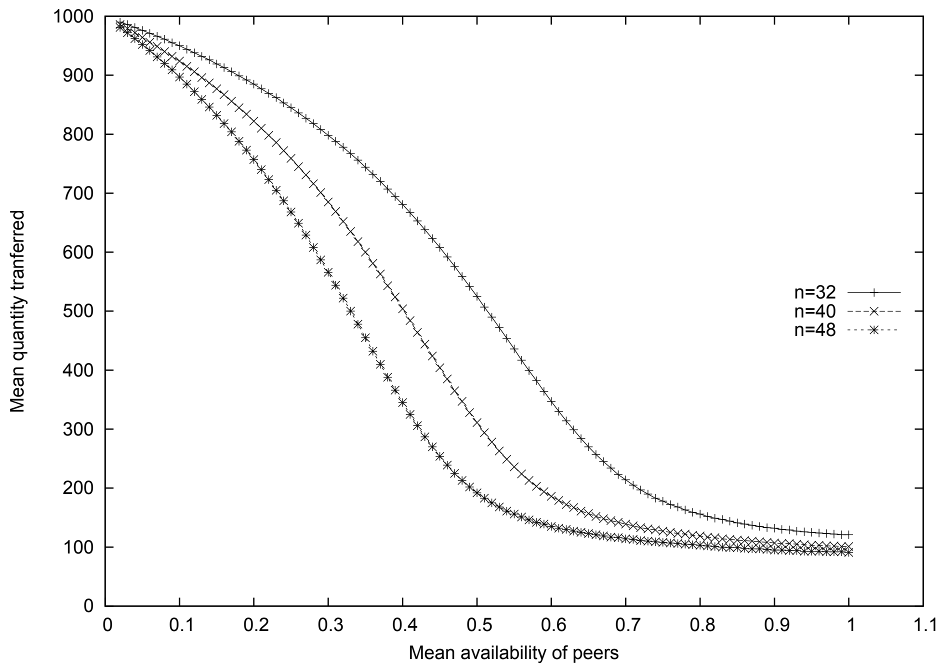


Figure 4.4: Mean quantity of data transferred during a repair for various storage overhead ($k = 16$, , $FileSize = 1GB$)

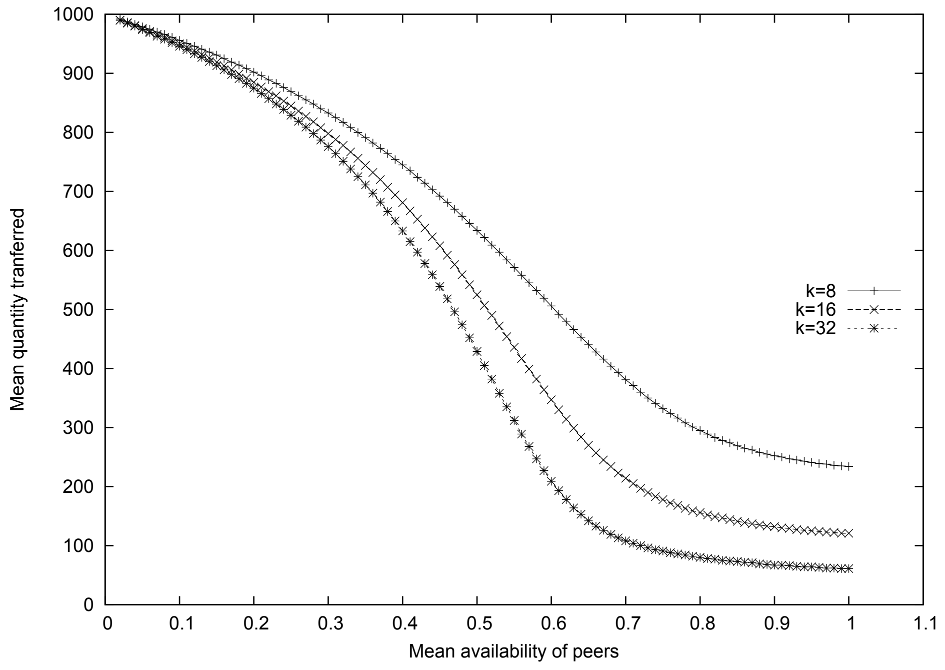


Figure 4.5: Mean quantity of data transferred during a repair for various k. ($n = 2 * k$, $FileSize = 1GB$)

Résumé en français

Les systèmes de stockage actuels font face à une explosion des données à gérer. Ils doivent d'une part en assurer le stockage de manière fiable et durable tout en étant capable de les restituer à la demande, et ce, le plus rapidement possible. A l'échelle actuelle, il serait illusoire d'imaginer une unique entité centralisée capable de stocker et de restituer les données de tous ses utilisateurs. Bien que du point de vue de l'utilisateur, le système de stockage apparait tel un unique interlocuteur, son architecture sous-jacente est nécessairement devenue distribuée. En d'autres termes, le stockage n'est plus assigné à un équipement centralisé, mais est maintenant distribué parmi de multiples entités de stockage indépendantes, connectées via un réseau. **Par conséquent, la bande passante inhérente à ce réseau devient une ressource à prendre en compte dans le design d'un système de stockage distribué.** En effet, la bande passante d'un système est intrinsèquement une ressource limitée, qui doit être convenablement gérée de manière à éviter toute congestion du système. Dans un système de stockage distribué, les deux principales fonctions consommatrices de bande passante sont (i) les opérations de stockage et de restitution, et (ii) la maintenance du système.

(i) Durant les opérations de stockage et de restitution, les données sont échangées via le réseau, entre les utilisateurs et le système. Ces échanges consomment une quantité importante de bande passante, de manière proportionnelle à la taille des données.

(ii) Un système de stockage subit assurément des défaillances, qu'elles soient temporaires ou permanentes, typiquement le crash d'un disque, impliquant la perte de données. Ces données perdues doivent être restaurées afin que le système reste dans un état sain. Ces opérations de maintenance entrainent de multiples transferts de données sur le réseau, afin de réparer les pertes liées à ces défaillances. Bien qu'essentielles au maintien de la fiabilité du système, ces réparations demeurent extrêmement couteuses en termes de consommation de bande passante.

Le problème de la gestion efficace de la bande passante revêt une importance accrue lorsque le système de stockage repose sur une architecture pair-à-pair. Dans un système de stockage pair-à-pair, les ressources de stockage sont directement fournies par les utilisateurs. Chaque utilisateur met à disposition du système une partie de son espace de stockage local. Le système aggrège cet *eldorado* d'espace de stockage afin d'offrir un service de stockage fiable à ces participants. Bien que les architectures pair-à-pair offrent d'intéressantes propriétés, telle qu'une tolérance aux pannes élevée, ou encore un fort potentiel de passage à l'échelle, ils souffrent cependant d'importantes limitations concernant la bande passante. En effet, la bande passante disponible dans une architecture pair-à-pair est considérablement réduite à cause du *churn* et des défaillances.

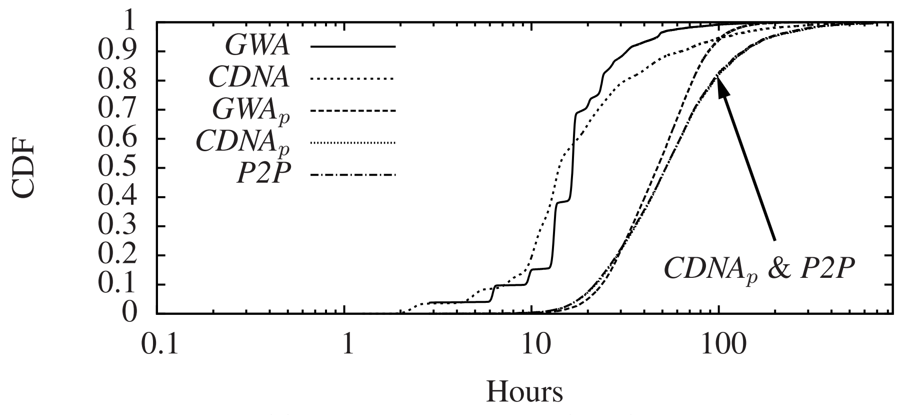
Dans un système pair-à-pair, les utilisateurs peuvent se connecter et se déconnecter du système selon leur bon vouloir, sans aucune forme d'avertissement. Cet intermittence dans leur période de connexion, phénomène communément appelé churn, provoque une asynchronie entre les périodes de connexion des utilisateurs. Intuitivement, deux utilisateurs ne peuvent échanger des données que si ils sont connectés en même temps. De ce fait, la quantité effective qu'ils peuvent échanger est réduite par ce churn. Ceci provoque donc une diminution de la bande passante disponible entre les utilisateurs, ce qui conduit directement à une augmentation des temps de stockage et de restitution des données. De plus, dans un système de stockage pair-à-pair, la totalité de la bande passante disponible est partagée entre la transmission des "données utiles", (c'est à dire les opérations de stockage et restitution) et les transferts liée aux opérations de maintenance. Ces opérations de maintenance consomment donc une part importante de la bande passante disponible. De ce fait, elles réduisent la bande passante disponible pour les sauvegardes et restitutions, et il devient donc primordial de limiter leur impact sur la bande passante du système.

Contributions

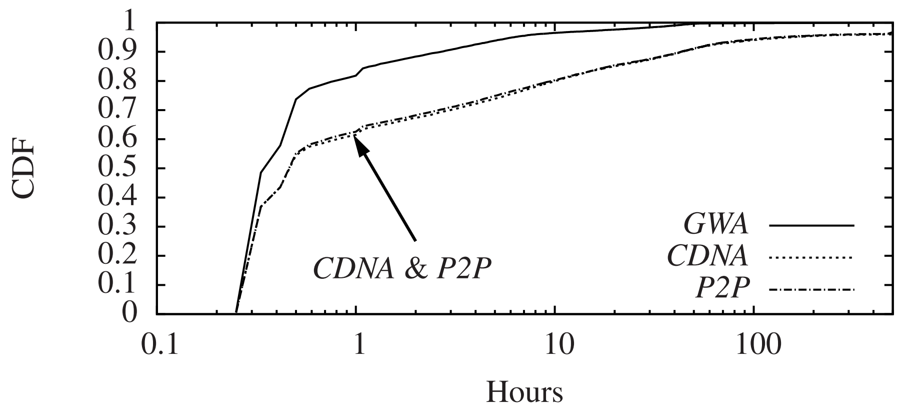
Cette thèse se propose **d'optimiser l'utilisation de la bande passante dans les systèmes de stockage distribués**, en limitant l'impact du churn et des défaillances. L'objectif est double, le but est d'une part, de maximiser la bande passante disponible pour les échanges de données, et d'une autre part de réduire la consommation de bande passante inhérente aux opérations de maintenance. Bien que la réduction de la consommation de la bande passante puisse être étudiée selon de nombreux angles, cette thèse se concentre sur deux points spécifiques que sont l'augmentation de la bande passante disponible entre les utilisateurs, et la réduction de la bande passante relative aux opérations de maintenance.

Architecture assistée par des gateways

Dans la première partie de ce manuscrit, nous étudions comment réduire l'impact de l'asynchronie, résultant du churn, sur la bande passante disponible pendant les opérations de sauvegarde et de restitution. La première contribution de cette thèse présente une architecture pair-à-pair hybride qui tient compte de la topologie bas-niveau du réseau, c'est à dire la présence de *gateways* entre les utilisateurs et le système. Ces gateways deviennent des composants actifs agissant comme des buffers, afin de compenser l'intrinsèque instabilité des machines. De par leur disponibilité élevée, les gateways fournissent un point de *rendez-vous* stable entre les périodes de connexion des utilisateurs, masquant de ce fait leur asynchronie. Cette architecture est évaluée par simulation en utilisant des traces de disponibilité provenant de réels systèmes. Les résultats montrent que le temps nécessaire aux sauvegardes (voir Figure 4.6a) et restitutions (voir Figure 4.6b) des données est considérablement réduit du fait d'une meilleure utilisation de la bande passante disponible.



(a) Temps de sauvegarde (TTB)

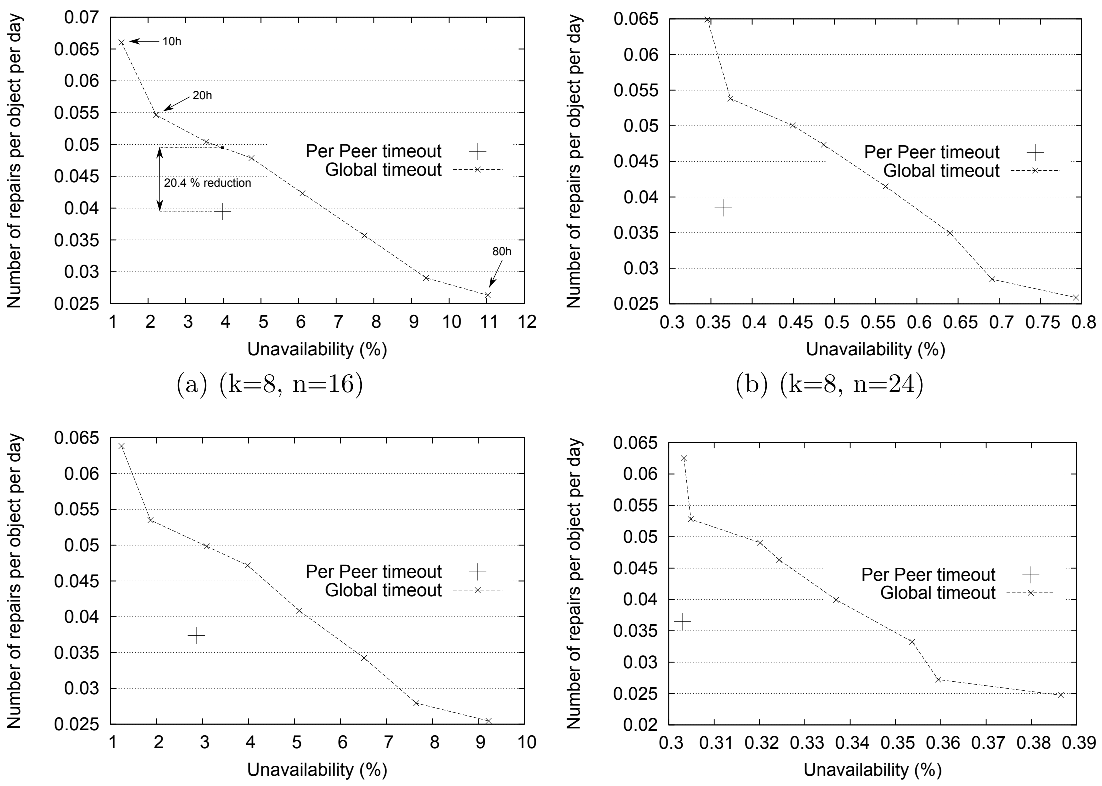


(b) Temps de restitution (TTR)

Figure 4.6: CDF des temps de sauvegarde et de restitution pour la trace Skype.

Mécansime de timeout adaptatif au niveau utilisateur

La seconde contribution de cette thèse se situe en amont des opérations de maintenance, c'est à dire durant la phase de détection des défaillances. Dans un système pair-à-pair, distinguer une défaillance permanente, nécessitant une réparation, d'une défaillance temporaire est particulièrement compliqué. Dans ce dernier cas, une réparation ne s'avèrerait pas nécessaire et consommerait donc de la bande passante inutilement. Afin de décider si une réparation doit être, ou non, effectuée, nous proposons un mécanisme de timeout adaptatif au niveau utilisateur. Ce mécanisme, basé sur une approche Bayésienne est évalué en simulation sur des traces de disponibilité tirées de réels systèmes. Les résultats montrent que, comparé aux mécanismes de timeout classiques, le nombre de réparations inutiles est réduit de manière significative, diminuant ainsi la bande passante inutilement consommée (voir Figure 4.7).



(a) (k=8, n=16)

(b) (k=8, n=24)

(c) (k=16, n=32)

(d) (k=16, n=48)

Figure 4.7: Résultats pour la trace Skype selon différents paramètres du code.

Protocole de réparation de données encodées

La troisième contribution décrit un protocole permettant la réparation efficace de données encodées via des codes à effacement. Les codes à effacement fournissent un moyen prometteur d'introduire de la redondance à moindre cout, en termes d'espace de stockage. Dans le cas d'une simple réplication de donnée, la réparation est immédiate puisqu'elle ne nécessite qu'un transfert de la donnée. Cependant, il est maintenant admis que la réparation de données encodées est extrêmement couteuse en consommation de bande passante. La plupart des travaux réduisant la bande passante inhérente à la réparation proposent des mécanismes au niveau fichier. Le mécanisme que nous proposons permet de réparer de multiples fichiers simultanément, factorisant ainsi le cout en bande passante. Ce protocole est implémenté et déployé sur un réseau de test public (GRID5000) afin de montrer ces performances dans un environnement réel. Comparé aux mécanismes traditionnellement implémentés, les resultats montrent que la bande passante nécessaire est réduite de moitié (voir Figure 4.8), tout en réduisant considérablement les temps de réparations (voir Figure 4.9).

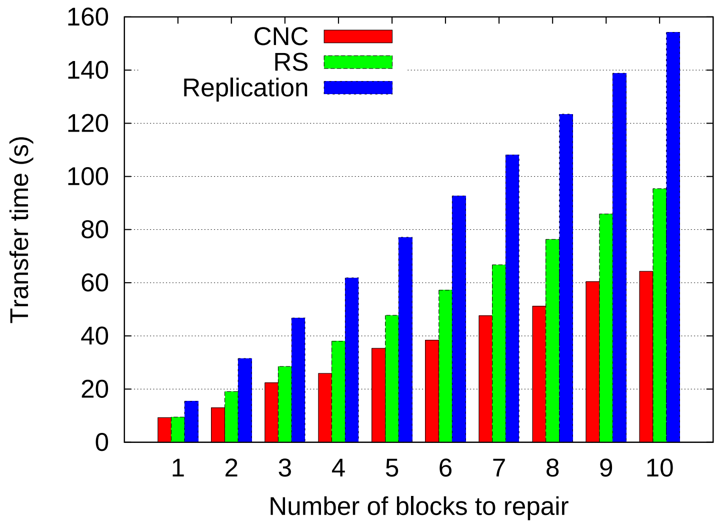


Figure 4.8: Temps pour transférer la quantité de données nécessaires afin d'effectuer la réparation.

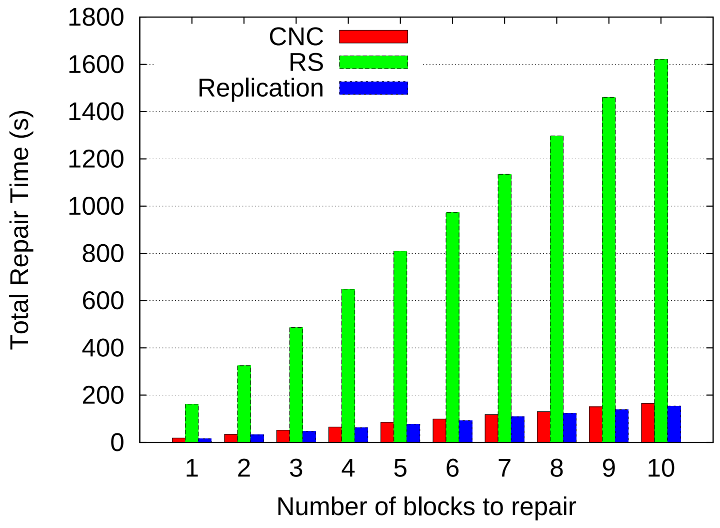


Figure 4.9: Temps de réparation total.

Appendix

Additional Contributions

The study of nodes availability in section 2 and 3.1, as well as efficient repair mechanisms in section 3.2 has led to additional contributions, not described in this thesis. As these contributions was not directly related to the bandwidth consumption, we briefly present them in this section.

Regenerating Codes: A System Perspective

In this work, we studied *regenerating codes* from a practical viewpoint. Regenerating codes offer the same properties as erasure correcting codes with respect to storage and availability. Yet, as opposed to erasure correcting codes, regenerating codes significantly lower the network traffic upon repairs. The seminal paper on regenerating codes applies network coding to storage systems and defines the optimal tradeoff between the amounts of data stored and transferred. Regenerating codes, designed to be as generic as possible, rely on many parameters, difficult to grasp in practice where device availability vary from a system to another. Numerous variants of regenerating codes thus exist.

In this work we provided an analysis of regenerating codes for practitioners to grasp the various tradeoffs. In order to help choose the right parameters and coding scheme, we made the following contributions: (i) We studied the influence of the various parameters at the system level, depending on storage device availability. We showed that the optimum at device level does not always apply at system level.
(ii) We compared the computational costs of various coding schemes for regenerating codes (random codes , product-matrix codes, and exact linear codes) to the costs of classical erasure correcting codes (Reed-Solomon codes). Our goal was to provide system designers with concrete information to help them choose the best parameters and design for regenerating codes. This work, in collaboration with Steve Jiekak, Anne-Marie Kermarrec, Nicolas Le Scouarnec and Gilles Straub has been published in the proceedings of the international workshop on Dependability Issues in Cloud Computing, DISCCO 2012.

On The Impact of Users Availability In OSNs

In this work we studied the impact of users availability in Online Social Networks (OSNs). While the availability has been extensively measured, studied and leveraged for computers and application runtimes, we know little about the online presence patterns of users and their impact on specific features in the case of OSNs. In this work, we showed that online presence is very heterogeneous among users, and that complex interactions may influence it. More specifically, we showed the online presence of users on a given OSN enables to extract finer characteristics and correlations such as the simultaneous presence of users with respect to their friends. This availability information can be of particular interest to various practical functionalities such as efficient information spreading and influence in OSNs, load prediction on the service provider's platform, or resource management in distributed social networks. In addition, while the "who-knows-who" relations of OSN structures are inherently time varying graphs, extracting at a finer granularity the user presence is interesting. In this work, we investigated the dynamic nature of OSNs, based on a trace we have extracted from Myspace. This trace contains both the social network structure (friendship relations), as well as availability information in each timeslot.

From our analysis, we concluded that the availability of a user is highly correlated with that of her friends. We then showed that user availability plays an important role in some algorithms and focused on information spreading. In fact, identifying central users i.e. those located in central positions in a network, is key to achieve a fast dissemination and the importance of users in a social graph precisely vary depending on their availability. This work, in collaboration with Antoine Boutet, Anne- Marie Kermarrec and Erwan Le Merrer has been published in the proceedings of the international workshop on Social Network Systems, SNS 2012.

Existence of solutions for k=2 (over $\mathbb{F}_{256}$)

	Node 1	Node 2	Node 3	Node 4
File X	[154, 176]	[213, 171]	[90, 13]	[114, 48]
File Y	[128, 237]	[42, 233]	[148, 116]	[199, 99]
File Z	[165, 117]	[123, 72]	[238, 48]	[114, 116]

Table 4.1: Initial coefficients of each encoded block for each file, stored on the four nodes.

	Node 1	Node 2	Node 3	Node 4
Alignment vector	[251, 26, 134]	[65, 1, 12]	[127, 23, 21]	[98, 234, 37]

Table 4.2: Coefficients of the linear combination between the three encoded blocks stored on each node

	File X	File Y	File Z
Node 1	[51, 69]	[129, 138]	[157, 59]
Node 2	[182, 62]	[42, 233]	[14, 71]
Node 3	[52, 65]	[126, 66]	[55, 215]
Node 4	[211, 178]	[132, 55]	[65, 159]

Table 4.3: Resulting coefficients of the RepairedBlocks sent by each node.

	to recover File X	to recover FileY	to recover File Z
Combination of repairBlocks	[171, 4, 15, 1]	[31, 16, 130, 1]	[206, 203, 210, 1]

Table 4.4: Coefficients of the linear combination of RepairedBlocks to cancel interference and restore one block for each file.

	block for File X	block for FileY	block for File Z
Stored blocks	[70, 10]	[55, 226]	[121, 235]

Table 4.5: Coefficients of the new encoded blocks stored for each file.

Bibliography

[1] Amazon Web Services. http://s3.amazonaws.com.

[2] EMC Study. http://www.emc.com/about/news/press/2011/20110628-01.htm.

[3] FON. http://corp.fon.com.

[4] Google Drive. http://drive.google.com/.

[5] Microsoft SkyDrive. http://skydrive.live.com/.

[6] Repository. http://www.cs.uiuc.edu/homes/pbg/availability/.

[7] Hussam Abu-Libdeh, Paolo Costa, Antony Rowstron, Greg O'Shea, and Austin Donnelly. Symbiotic routing in future data centers. *SIGCOMM Comput. Commun. Rev.*, 41(4):–, August 2010.

[8] Szymon Acedanski, Supratim Deb, Muriel Medard, and Ralf Koetter. How good is random linear coding based distributed networked storage. In *NetCod*, 2005.

[9] Nitin Agrawal, William J. Bolosky, John R. Douceur, and Jacob R. Lorch. A five-year study of file-system metadata. In *ToS*, volume 3, 2007.

[10] Rudolf Ahlswede, Ning Cai, Shuo-Yen Li, and Raymond Yeung. Network Information Flow. *IEEE Transactions On Information Theory*, 46:1204–1216, 2000.

[11] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, 2008.

[12] Ranjita Bhagwan, Stefan Savage, and Geoffrey Voelker. Understanding Availability. In *IPTPS*, 2003.

[13] Ranjita Bhagwan, Kiran Tati, Yu-Chung Cheng, Stefan Savage, and Geoffrey M. Voelker. Total recall: system support for automated availability management. In *NSDI*, 2004.

[14] Charles Blake and Rodrigo Rodrigues. High availability, scalable storage, dynamic peer networks: pick two. In *HOTOS*, 2003.

[15] William J. Bolosky, John R. Douceur, David Ely, , and Marvin Theimer. Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs. In *SIGMETRICS*, 2000.

[16] Viveck R. Cadambe and Syed Ali Jafar. Interference alignment and the degrees of freedom for the K user interference channe. *IEEE Trans. Inf. Theory*, 54:3425–3441, 2008.

[17] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: a highly available cloud storage service with strong consistency. In *SOSP*, 2011.

[18] Stéphane Caron, Frédéric Giroire, Dorian Mazauric, Julian Monteiro, and Stéphane Pérennes. Data life time for different placement policies in p2p storage systems. In *Globe*, 2010.

[19] Meeyoung Cha, Pablo Rodriguez, Sue Moon, and Jon Crowcroft. On next-generation telco-managed P2P TV architectures. In *IPTPS*, 2008.

[20] Byung-Gon Chun, Frank Dabek, Andreas Haeberlen, Emil Sit, Hakim Weatherspoon, Frans Kaashoek, John Kubiatowicz, and Robert Morris. Efficient replica maintenance for distributed storage systems. In *NSDI*, 2006.

[21] Landon P. Cox, Christopher D. Murray, and Brian D. Noble. Pastiche: making backup cheap and easy. *SIGOPS Oper. Syst. Rev.*, 36:285–298, 2002.

[22] Frank Dabek, M. Frans Kaashoek, David Karger, Robert Morris, and Ion Stoica. Wide-area cooperative storage with CFS. In *SOSP*, 2001.

[23] Alexandros G. Dimakis, P. Brighten Godfrey, Yunnan Wu, Martin O. Wainwright, and Kannan Ramchandran. Network Coding for Distributed Storage Systems. In *INFOCOM*, 2007.

[24] Alexandros G. Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A Survey on Network Codes for Distributed Storage. *The Proceedings of the IEEE*, 99:476–489, 2010.

[25] Marcel Dischinger, Andreas Haeberlen, Krishna P. Gummadi, , and Stefan Saroiu. Characterizing Residential Broadband Networks. In *IMC*, 2007.

[26] John Douceur and Roger Wattenhofer. Competitive hill-climbing strategies for replica placement in a distributed file system. In *In DISC*, 2001.

[27] John R. Douceur. Is remote host availability governed by a universal law? In *SIGMETRICS*, 2003.

[28] John R. Douceur and Roger Wattenhofer. Optimizing file availability in a secure serverless distributed file system. In *SRDS*, 2001.

[29] Alessandro Duminuco and Ernst Biersack. A Pratical Study of Regenerating Codes for Peer-to-Peer Backup Systems. In *ICDCS*, 2009.

[30] Alessandro Duminuco, Ernst Biersack, and Ernst Biersack. Hierarchical codes: How to make erasure codes attractive for peer-to-peer storage systems. In *P2P*, 2008.

[31] Alessandro Duminuco, Ernst Biersack, and Taoufik En-Najjary. Proactive replication in distributed storage systems using machine availability estimation. In *CoNEXT*, 2007.

[32] Richard J. Dunn, John Zahorjan, Steven D. Gribble, and Henry M. Levy. Presence-based availability and p2p systems. In *P2P*, 2005.

[33] Gilles Fedak, Jean-Patrick Gelas, Thomas Herault, Victor Iniesta, Derrick Kondo, Laurent Lefevre, Paul Malécot, Lucas Nussbaum, Ala Rezmerita, and Olivier Richard. DSL-Lab: A Low-Power Lightweight Platform to Experiment on Domestic Broadband Internet. In *ISPDC*, 2010.

[34] Daniel Ford, François Labelle, Florentina I. Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, 2010.

[35] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition edition, 1979.

[36] Abdullah Gharaibeh and Matei Ripeanu. Exploring data reliability tradeoffs in replicated storage systems. In *HPDC*, 2009.

[37] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The google file system. In *SOSP*, 2003.

[38] Christos Gkantsidis and P. Rodriguez. Network coding for large scale content distribution. In *INFOCOM*, 2005.

[39] P. Brighten Godfrey, Scott Shenker, and Ion Stoica. Minimizing churn in distributed systems. In *SIGCOMM*, 2006.

[40] Kevin M. Greenan, Parascale Inc, James S. Plank, and Jay J. Wylie. Mean time to meaningless: MTTDL, markov models, and storage system reliability. In *HotStorage*, 2010.

[41] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *IPTPS*, 2006.

[42] Andreas Haeberlen, Alan Mislove, and Peter Druschel. Glacier: highly durable, decentralized storage despite massive correlated failures. In *NSDI*, 2005.

[43] Yuchong Hu, Henry C. H. Chen, Patrick P. C. Lee, and Yang Tang. NCCloud: Applying Network Coding for the Storage Repair in a Cloud-of-Clouds. In *FAST*, 2012.

[44] Yuchong Hu, Chiu-Man Yu, Yan Kit Li, P.P.C. Lee, and J.C.S. Lui. Ncfs: On the practicality and extensibility of a network-coding-based distributed file system. In *NetCod*, 2011.

[45] Cheng Huang, Minghua Chen, and Jin Li. Pyramid codes: Flexible schemes to trade space for access efficiency in reliable data storage systems. In *NCA*, 2007.

[46] Hai Huang, Wanda Hung, and Kang G. Shin. FS2: dynamic data replication in free disk space for improving disk performance and energy consumption. In *SOSP*, 2005.

[47] Vaishnav Janardhan and Henning Schulzrinne. Peer assisted VoD for set-top box based IP network. In *P2P-TV*, 2007.

[48] Osama Khan, Randal Burns, James Plank, William Pierce, and Cheng Huang. Rethinking Erasure Codes for Cloud File Systems: Minimizing I/O for Recovery and Degraded Reads. In *FAST*, 2012.

[49] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, Chris Wells, and Ben Zhao. OceanStore: an architecture for global-scale persistent storage. *SIGPLAN Not.*, 35:190–201, 2000.

[50] Martin Landers, Han Zhang, and Kian-Lee Tan. PeerStore: Better Performance by Relaxing in Peer-to-Peer Backup. In *P2P*, 2004.

[51] Nikolaos Laoutaris, Georgios Smaragdakis, Pablo Rodriguez, and Ravi Sundaram. Delay Tolerant Bulk Data Transfers on the Internet. In *SIGMETRICS*, 2009.

[52] Pietro Michiardi Laszlo Toka, Matteo Dell'Amico. Online Data Backup: A Peer-Assisted Approach. In *P2P*, 2010.

[53] Stevens Le Blond, Fabrice Le Fessant, and Erwan Le Merrer. Finding good partners in availability-aware p2p networks. In *SSS*, 2009.

[54] T. Leighton. Improving Performance on the Internet. *CACM*, 52, 2009.

[55] Mark Lillibridge, Sameh Elnikety, Andrew Birrell, Mike Burrows, and Michael Isard. A cooperative internet backup scheme. In *Usenix ATC*, 2003.

[56] W. K. Lin, D. M. Chiu, and Y. B. Lee. Erasure Code Replication Revisited. In *P2P*, 2004.

[57] Yunfeng Lin, Ben Liang, and Baochun Li. Priority random linear codes in distributed storage systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(11):1653–1667, 2009.

[58] Fangming Liu, Ye Sun, Bo Li, Baochun Li, and Xinyan Zhang. FS2You: Peer-Assisted Semipersistent Online Hosting at a Large Scale. *IEEE Trans. Parallel Distrib. Syst.*, 21:1442–1457, 2010.

[59] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma, and Steven Lim. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Surveys and Tutorials*, 7:72–93, 2005.

[60] James W. Mickens and Brian D. Noble. Exploiting availability prediction in distributed systems. In *NSDI*, 2006.

[61] Ramsés Morales and Indranil Gupta. Avmon: Optimal and scalable discovery of consistent availability monitoring overlays for distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 20(4):446–459, 2009.

[62] Radhika Niranjan Mysore, Andreas Pamboris, Nathan Farrington, Nelson Huang, Pardis Miri, Sivasankar Radhakrishnan, Vikram Subramanya, and Amin Vahdat. Portland: a scalable fault-tolerant layer 2 data center network fabric. In *SIGCOMM*, 2009.

[63] Daniel Nurmi, John Brevik, and Rich Wolski. Modeling machine availability in enterprise and wide-area distributed computing environments. In *Euro-Par*, 2005.

[64] Frédérique E. Oggier and Anwitaman Datta. Self-repairing homomorphic codes for distributed storage systems. In *INFOCOM*, 2011.

[65] Lluis Pamies-Juarez, Pedro Garcia-Lopez, and Marc Sanchez-Artigas. Heterogeneity-aware erasure codes for peer-to-peer storage systems. In *ICPP*, 2009.

[66] Lluis Pamies-Juarez, Pedro García-López, and Marc Sánchez-Artigas. Availability and Redundancy in Harmony: Measuring Retrieval Times in P2P Storage Systems. In *P2P*, 2010.

[67] Lluis Pamies-Juarez, Frédérique E. Oggier, and Anwitaman Datta. An empirical study of the repair performance of novel coding schemes for networked distributed storage systems. *CoRR*, abs/1206.2187, 2012.

[68] Dimitris S. Papailiopoulos, Jianqiang Luo, Alexandros G. Dimakis, Cheng Huang, and Jin Li. Simple Regenerating Codes: Network Coding for Cloud Storage. In *INFOCOM*, 2012.

[69] Sriram Ramabhadran and Joseph Pasquale. Durability of replicated distributed storage systems. In *SIGMETRICS*, 2008.

[70] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[71] Rodrigo Rodrigues and Peter Druschel. Peer-to-peer systems. *Commun. ACM*, 53(10):72–82, October 2010.

[72] Rodrigo Rodrigues and Barbara Liskov. High availability in dhts: Erasure coding vs. replication. In *IPTPS*, 2005.

[73] David S. H. Rosenthal. Keeping bits safe: how hard can it be? *Commun. ACM*, 53(11):47–55, November 2010.

[74] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[75] Antony Rowstron and Peter Druschel. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility. In *SOSP*, 2001.

[76] Krzysztof Rzadca, Anwitaman Datta, and Sonja Buchegger. Replica placement in p2p storage: Complexity and game theoretic analyses. In *ICDCS*, 2010.

[77] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *MSST*, 2010.

[78] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM*, 2001.

[79] Kiran Tati and Geoffrey M. Voelker. On object maintenance in peer-to-peer systems. In *IPTPS*, 2006.

[80] Ashish Thusoo, Zheng Shao, Suresh Anthony, Dhruba Borthakur, Namit Jain, Joydeep Sen Sarma, Raghotham Murthy, and Hao Liu. Data warehousing and analytics infrastructure at facebook. In *SIGMOD*, 2010.

[81] Jing Tian, Zhi Yang, Wei Chen, Ben Y. Zhao, and Yafei Dai. Probabilistic failure detection for efficient distributed storage maintenance. In *SRDS*, 2008.

[82] Vytautas Valancius, Nikolaos Laoutaris, Laurent Massoulié, Christophe Diot, and Pablo Rodriguez. Greening the Internet with Nano Data Centers. In *CoNext*, 2009.

[83] Kashi Venkatesh Vishwanath and Nachiappan Nagappan. Characterizing cloud computing hardware reliability. In *SoCC*, 2010.

[84] Hakim Weatherspoon and John Kubiatowicz. Erasure Coding Vs. Replication: A Quantitative Comparison. In *IPTPS*, 2002.

[85] Zhi Yang, Yafei Dai, and Zhen Xiao. Exploring the cost-availability tradeoff in p2p storage systems. In *ICPP*, 2009.

Résumé

Les systèmes de stockage actuels font face à une explosion des données à gérer. A l'échelle actuelle, il serait illusoire d'imaginer une unique entité centralisée capable de stocker et de restituer les données de tous ses utilisateurs. Bien que du point de vue de l'utilisateur, le système de stockage apparaît tel un unique interlocuteur, son architecture sous-jacente est nécessairement devenue distribuée. En d'autres termes, le stockage n'est plus assigné à un équipement centralisé, mais est maintenant distribué parmi de multiples entités de stockage indépendantes, connectées via un réseau. Par conséquent, la bande passante inhérente à ce réseau devient une ressource à prendre en compte dans le design d'un système de stockage distribué. En effet, la bande passante d'un système est intrinsèquement une ressource limitée, qui doit être convenablement gérée de manière à éviter toute congestion du système. Cette thèse se propose d'optimiser l'utilisation de la bande passante dans les systèmes de stockage distribués, en limitant l'impact du churn et des défaillances. L'objectif est double, le but est d'une part, de maximiser la bande passante disponible pour les échanges de données, et d'une autre part de réduire la consommation de bande passante inhérente aux opérations de maintenance. Pour ce faire, nous présentons trois contributions distinctes. La première contribution présente une architecture pair-à-pair hybride qui tient compte de la topologie bas-niveau du réseau, c'est à dire la présence de *gateways* entre les utilisateurs et le système. La seconde contribution propose un mécanisme de timeout adaptatif au niveau utilisateur, basé sur une approche Bayésienne. La troisième contribution décrit un protocole permettant la réparation efficace de données encodées via des codes à effacement. Enfin, cette thèse se conclut sur la possibilité d'utiliser des techniques d'alignement d'interférence, communément utilisées en communication numérique afin d'accroitre l'efficacité des protocoles de réparation de données encodées.

Abstract

Modern storage systems have to face the surge of the amount of data to handle. At the current scale, it would be an illusion to believe that a single centralized storage device is able to store and retrieve all its users' data. While from the user's viewpoint the storage system remains a single interlocutor, its underlying architecture has become necessarily distributed. In others words, storage is no longer assigned to a centralized storage equipment, but is now distributed between multiple independent storage devices, connected via a network. Therefore, when designing networked storage systems, bandwidth should now be taken into account as a critical resource. In fact, the bandwidth of a system is intrinsically a limited resource which should be handled with care to avoid congestion. The focus of this thesis is to optimize the available bandwidth of distributed storage systems, lowering the impact of churn and failures. The objective is twofold, on the one hand the purpose is to increase the available bandwidth for data exchanges and on the other hand, to decrease the amount of bandwidth consumed by maintenance. We present three distinct contributions in this manuscript. The first contribution of this thesis presents an hybrid peer-to-peer architecture taking into account the low level topology of the network i.e., the presence of gateways between the system and the users. The second contribution proposes an adaptive and user-level timeout mechanism, based on a Bayesian approach. The third contribution describes a repair protocol especially designed for erasure-coded stored data. Finally, this thesis concludes on the possibility of employing interference alignment techniques in order to increase the efficiency of repair protocols especially designed for encoded data.