Exploiting Rateless Coding in Structured Overlays to Achieve Persistent Storage

Heverson Borba Ribeiro

ANNÉE 2012

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

Heverson Borba Ribeiro

préparée à l'unité de recherche n^{o} 6074
Institut de Recherche en Informatique et Systèmes Aléatoires
IRISA

Exploiting Rateless Coding in Structured Overlays to Achieve Persistent Storage.

Thèse soutenue
le 12 octobre 2012

devant le jury composé de :

Michel HURFIN / Directeur de thèse
Chargé de Recherche à l'INRIA, Rennes, France.

Emmanuelle ANCEAUME / Encadrante
Chargée de Recherche au CNRS, Rennes, France.

Achour MOSTEFAOUI / Rapporteur
Professeur à l'Université de Nantes, Nantes, France.

Pierre SENS / Rapporteur
Professeur à l'Université Pierre et Marie Curie (Paris VI), Paris, France.

César VIHO / Président du jury
Professeur à l'Université de Rennes 1, Rennes, France.

Yann BUSNEL / Examinateur
Maitre de Conférence à l'Université de Nantes, Nantes, France.

Acknowledgments

I am grateful to many people that contributed both direct and indirect to this achievement. I would like to thank Professor Pierre Sens (University Pierre et Marie Curie - Paris VI) and Professor Achour Mostefaoui (University of Nantes) for accepting to be reporters of my thesis and also for spending their precious time examining my manuscript. I would like to thank Professor César VIHO (University of Rennes 1) for presiding over my thesis defense and Dr.Yann Busnel (University of Nantes) for accepting to be a member of the jury. Thank you very much for all your comments.

I also would like to express my gratitude towards my advisor Dr. Emmanuelle Anceaume and my thesis director Dr. Michel Hurfin for all the time they spent reading and correcting my work. Without their continous support and infinite patience none of this work would have been possible. All the other people from ADEPT team I spent a great time with. Aina Voaja, for all your help by the time I arrived in the team and also for being an example to me. Frederic Majorczyk and Izabela Elena Moise, thanks for the great time, you are amazing team mates. Romaric Ludinard, thanks for forcing me to speak french in the office and for the great "champagne experience".

A very special thanks goes out towards Antoine Boutet, Davide Frey and Anne-Marie Kermarrec for accepting me in the ASAP team when I needed the extra time to dedicate to my thesis. Without your support this work would not have reached the happy end. I also would like to thank the inspiring professors I had in Brazil, Dr. Lau Cheuk Lung my advisor during my master course, Dr. Altair Santin and Dr. Carlos Alberto Maziero.

Many thanks to all the special friends and collegues who made my time as a PhD candidate truly memorable. Sagar Sen, Marko Obrovac, Giulio Zecca, Alice and Nicolas Pépin-Hermann. Ayush Bhandari, thanks for your inspiring photographies. The brazilian "dream-team" Daniel Morés Aires, Fernanda Vanin, Nara Alonso, Fabrizia Lima and Ricardo Scholz. Mateus de Oliveira Oliveira, the bravest PhD student I ever met. My frenchy special ones, Jean-François Berrée and Rywal Meziere. Undoubtedly, a very special thanks goes towards André Lage and Livia Lage for all the great time we spent together. André, there are no words that can express my gratitude towards you man. Last but not least, I would like to immeasurably thank with all my heart Marianne Meusburger for being by my side in the last year and half, thanks for your support, care and love in all the moments I really needed.

Para finalizar, dedico este trabalho à minha família. À minha querida mãe Helena da Luz Borba Ribeiro, pelo esforço e coragem com os quais enfrentou todos os desafios que a vida lhe conferiu e a dedicação com a qual nos educou para que pudéssemos enfrentar os nossos próprios desafios. À memória do meu falecido pai, Santino Ribeiro, o qual tenho certeza que estaria orgulhoso de tal realização. Aos meus queridos irmãos Helaine e Heric Borba Ribeiro. Aos meus cunhados-irmãos, Haiko Abrahams e Kátia Ribeiro, à tia Delirce de Oliveira Borba e à minha querida super-sobrinha Handressa Ribeiro Abrahams ("abiamps"). Apesar da distância física dos últimos anos, vocês estiveram constantemente em meus pensamentos. Em todos os momentos mais difíceis dessa jornada era em vocês em quem pensava para buscar as forças necessárias para continuar a busca do meu objetivo.

For all the others I have forgotten or for space reasons I have not mentioned my heartfelt thanks.

Heverson BORBA RIBEIRO

Exploiting Rateless Coding in Structured Overlays to Achieve Persistent Storage

Abstract

The substantial increase in the amount of information over the Internet has contributed to an extraordinary demand for persistent data storage. Centralized storage architectures are expensive, weakly scalable and vulnerable to attacks as they represent single points of failure in the system. Over last few years, peer-to-peer architectures have emerged as an alternative for implementing persistent data-storage. Open peer-to-peer systems are fundamentally scalable and cheaper than client-server approaches. However, in order to successfully build persistent storage systems using the peer-to-peer approach two fundamental challenges need to be addressed. *a*) To cope with the transient connectivity of peers. *b*) To reduce the impact of misbehaving peers. Replication is a common approach used to cope with transient connectivity in peer-to-peer storage systems. However, depending on the frequency peers join and leave the system this approach can present negative impacts in terms of storage overhead and bandwidth consumption. Peer-to-peer overlays that focus on tolerating the presence of Byzantine peers usually make the assumption that no more than a bounded fraction of peers in the system are malicious. However, estimating the proportion of malicious peers in open peer-to-peer system is not reliable. Thus, finding a scalable architecture to provide reliable and persistent data storage while coping with these issues is an interesting achievement.

In this thesis we present the design of Datacube. Datacube is an efficient and scalable peer-to-peer storage architecture that provides data persistence by implementing a hybrid redundancy scheme on top of a cluster-based structured overlay. The hybrid redundancy scheme proposed by Datacube ensures data persistence and integrity despite the intermittent connection of peers and the presence of adversarial peers. Datacube relies on the properties of the new class of rateless erasure codes to implement its hybrid redundancy scheme.

The analytical evaluations have shown that Datacube performs notably well in terms of availability, storage overhead and bandwidth. Additionally, empirical evaluations have shown the performance of rateless erasure codes in the context of peer-to-peer storage systems. These evaluations helped to understand how the coding parameters impact on the performance of the architecture. To the best of our knowledge, this is the first comprehensive study that helps application designers in finding the values for the coding parameters to best fit their peer-to-peer context.

Keywords :
distributed systems, data storage, structured peer-to-peer networks, fountain codes.

Heverson BORBA RIBEIRO

L'Exploitation de Codes Fontaines pour un Stockage Persistant des Données dans les Réseaux d'Overlay Structurés

Résumé

L'importante augmentation de la quantité d'informations sur Internet a contribué à une forte demande pour un stockage persistant des données. Les architectures centralisées de stockage de données sont financièrement onéreuses, faiblement évolutives et vulnérables aux attaques car elles constituent un point unique de défaillance du système. Ces dernières années, les architectures pair-à-pair ont émergé comme une alternative pour la mise en place d'une architecture de stockage persistant des données. Les systèmes pair-à-pair sont fondamentalement évolutifs et moins chers que les modèles client-serveur. Cependant, pour construire des systèmes de stockage persistant en utilisant le modèle pair-à-pair, deux défis fondamentaux doivent être abordés. i) Faire face à la dynamique des pairs, en raison de leur connectivité transitoire. ii) Réduire l'impact du comportement malicieux des pairs. La réplication est une technique régulièrement utilisée pour faire face à la connectivité transitoire des systèmes de stockage pair-à-pair. Toutefois, selon le ratio d'arrivées et de départs des pairs dans le système, cette technique présente un impact négatif en termes de stockage et de bande passante. Les réseaux pair-à-pair qui offrent la tolérance aux fautes byzantins, font généralement l'hypothèse que seulement une fraction limitée des pairs dans le système sont des pairs de comportements malveillants. Toutefois, l'estimation de la proportion de pairs malveillants dans un système pair-à-pair est une opération peu fiable. Ainsi, créer une architecture qui fournit le stockage persistant de données fiables et qui permet de faire face à tous ces problèmes est une contribution souhaitable.

Dans cette thèse, nous proposons Datacube. Datacube est une architecture pair-à-pair de stockage de données scalable et efficace qui fournit un stockage persistant en utilisant un schéma de redondance hybride sur un réseau overlay structuré basée sur des clusters. Le schéma de redondance hybride proposé par Datacube assure la persistance et l'intégrité des données garantissant une forte résilience aux arrivées et départs de pairs même en présence de pairs malveillants. Datacube repose sur les propriétés des codes fontaines pour mettre en place son schéma de redondance hybride. Les évaluations analytiques ont montré que Datacube est notamment performant en termes de disponibilité, de surcharge de stockage et de bande passante. Nous avons aussi effectué des évaluations pratiques concernant les performances de deux types de codes fontaines dans le contexte de systèmes de stockage pair-à-pair. Ces évaluations ont aidé à comprendre l'impact des paramètres de codage sur les performances de Datacube. À notre connaissance, ceci est la première étude complète qui permet aux développeurs d'applications de trouver les valeurs des paramètres de codage adaptés au contexte des réseaux pair-à-pair.

Mots-clés :
systèmes répartis, stockage de données, réseaux pair-à-pair structurés, codes fontaines.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Contents

1.1 Context and Motivation

The Internet has experienced a significant rise in the number of users in the past few years. Many aspects have contributed to this growth, for instance the increased availability of cheap broadband Internet connections, the development of many mobile devices that allow users to be connected anywhere, among others. Moreover, the widespread use of content-sharing web applications (e.g., photo-sharing, video-streaming, social networks) have contributed to an extraordinary increase of the amount of digital information available on the Internet. According to the project Digital Universe [43], the volume of data created in the world in the last few years has reached the order of zettabytes, while the cost of storing information is continuously decreasing. These circumstances clearly show the need for efficient architectures to provide reliable and scalable storage while reducing the probability of data loss in order to satisfy this increasing demand for storage resources.

Traditional storage architectures based on centralized servers with internal disks or attached to high-end arrays through either SAN or NAS appliances (i.e., storage area network and network-attached storage, respectively) are strongly reliable and persistent architectures. However, these centralized architectures are extremely expensive since all dedicated resources (e.g. servers, storage, bandwidth, data centers, cooling systems, etc.) must be allocated in advance. Furthermore, a centralized server represents a single point of failure in any

architecture, leading the architectures to be more vulnerable to attacks. Likewise, centralized architectures do not scale easily, fast and unpredictable changes in the system workload (i.e., flash crowds) can potentially create bottlenecks, preventing users from retrieving their data. Besides that, by using centralized architectures the extraordinary capacity of free resources that can be exploited on the client side is completely ignored. For instance, with the decreasing cost of storage disks (i.e. internal and external hard disks) computers connected to the Internet are likely to have a great local storage capacity, which is often underused.

In the past ten years we have seen the emergence of peer-to-peer architectures as an alternative to centralized ones. Peer-to-peer systems are self-organizing architectures that do not depend on centralized entities. They are fundamentally scalable, members do not need to be conscious of the whole membership. In peer-to-peer systems, if each member has the knowledge of only a small fraction of all the members in the system it allows the system to reliably work. Peer-to-peer systems are composed of voluntary members that contribute with each other to create a robust architecture. Contrary to centralized architectures, in peer-to-peer ones the resources are spread over the system, the workload is consequently distributed among all members (i.e. peers), which improves the load balancing. The reliability of peer-to-peer systems makes them well positioned to provide scalable storage.

However, implementing storage in such architectures remains challenging due to two fundamental features of peer-to-peer systems. First, open distributed system such as peer-to-peer are susceptible to frequent members disconnections. In these systems members are able to join and leave the system without previous negotiation or information. This fundamental freedom of peer-to-peer systems jeopardizes the availability of data, therefore the employment of mechanisms to ensure durability is required. Second, the absence of trust relationship among peers encourages the inevitable presence of malicious members. Existing peer-to-peer overlays that focus on tolerating the presence of Byzantine nodes [34, 47, 5] make the assumption that no more than a bounded proportion of malicious nodes are present at any time and anywhere in the system. This is a fundamental and required assumption to be able to design Byzantine tolerant algorithms. However, differently from static distributed systems in which coverage of such an assumption is very high, in open systems, membership of the system evolves according to clients wishes. Hence, even if the proportion of malicious nodes in these systems can be roughly estimated, there are some corners of the system that may be potentially surrounded by more malicious nodes than expected [7]. As a consequence, correctness of Byzantine tolerant algorithms cannot be ensured, leading to the potential loss of integrity, or simply the non accessibility of the data stored at these nodes.

Thus, finding an architecture that benefits from the features of peer-to-peer systems to provide scalable, reliable and durable data storage while handling churn (see Section 2.1.4) and adversarial behavior issues is a strongly desired achievement.

1.2 Contributions

In this thesis we aim at designing a scalable infrastructure to provide a persistent storage by exploiting the properties of structured peer-to-peer overlays in order to guarantee durable access and integrity of data for highly dynamic environments despite the presence of adversarial peers. The main motivation for this research is the extraordinary increasing and continuous demand for reliable storage space over the Internet. The main contributions of

this thesis can be summarized as follows:

-The design of Datacube, a persistent data storage architecture for peer-to-peer systems. We propose a persistent storage architecture that relies on the properties of clustered DHT overlays. This clustered architecture efficiently limits the impact of churn and adversarial peers. We show that the hybrid redundancy scheme of Datacube (i.e., replication with erasure codes) provides durable access to data even in the presence of malicious peers in the cluster. The hybrid redundancy scheme of Datacube relies on a small set of constant replicas and on the properties of the recently proposed fountain codes. In this context, we also present some properties that we believe make this new class of codes more suitable for providing data storage on peer-to-peer systems than traditional erasure codes based on a fixed-rate coding, such as Reed-Solomon codes. In addition to that, Datacube provides a detection mechanism that periodically challenges clusters in order to identify and tag unreliable clusters. It allows Datacube clusters to isolate other clusters significantly populated by malicious peers and recover the data stored in these clusters by using rateless codes. Moreover, our proposed algorithm to generate encoded blocks employs a specific identification mechanism that allows Datacube to collect encoded blocks by using four different policies. Besides the default policy of randomly collecting encoded blocks, we propose three other policies to collect encoded blocks in Datacube.

-An analytical evaluation of the performance of Datacube. We perform an evaluation of Datacube in terms of data availability, storage and bandwidth overhead. We compare the stretch factor of Datacube redundancy scheme and the replication factor imposed by classical full replication required to have the data availability greater than a specified threshold (i.e., 0.99...0.9999). We also compare the number of fragments per node in Datacube with the ones needed in full replication and pure rateless erasure coding in order to guarantee a given data availability of 0.99. Finally, we derive the total bandwidth needed per node for maintaining Datacube redundancy mechanism in presence of churn.

-An empirical study of the benefits of using rateless codes for peer-to-peer storage architectures. We provide an extensive comparison of the two classes of fountain codes, namely LT and Online codes. We propose to compare the experimental performance of both LT and Online codes. This evaluation seeks not only to compare the performance of both codes in different adversarial environments and different application contexts (which is modeled through different size of data), but also to understand the impact of each coding parameter regarding the space and time complexity of the coding process. For the best of our knowledge, this work is the first comprehensive guideline that should help application designers configure these codes with optimal parameters values.

1.3 Publications

The current work has also resulted in the following publications:

- Datacube: A P2P Persistent Data Storage Architecture Based on Hybrid Redundancy Schema. H.B. Ribeiro, E. Anceaume, The 18th Euromicro International Conference on

Parallel, Distributed and Network-Based Processing (PDP). Pisa, Italy, February, 2010.

- Exploiting Rateless Coding in Structured Overlays to Achieve Data Persistence. H.B. Ribeiro, E. Anceaume, The 24th IEEE International Conference on Advanced Information Networking and Applications (AINA). Perth, Australia, April, 2010.

- A Comparative Study of Rateless Codes for P2P Persistent Storage. H.B. Ribeiro, E. Anceaume, The 12th Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS). New York, USA, September, 2010.

1.4 Outline

This manuscript is organized as follows:

Chapter 2 This chapter presents the state of the art of existing peer-to-peer architectures and highlights their distinguishing aspects. Following, it presents an overview of current storage architectures based on peer-to-peer architectures. We also discuss how important data redundancy is in order to preserve data durability and how redundancy can be explored. These architectures are evaluated according to three specific properties of storage architectures, namely data availability, reliability and maintenance.

Chapter 3 This chapter provides an overview of erasure codes. Together with replication, erasure coding is a mechanism used to provide redundancy. This chapter presents the fundamentals of erasure coding theory including a new recent class of rateless erasure codes. We highlight why this new class of rateless erasure codes is more efficient for peer-to-peer architectures than traditional erasure codes.

Chapter 4 This chapter presents our Datacube model and explains in details the hybrid scheme of redundancy used by Datacube. We also present a probabilistic analysis comparing the hybrid scheme of Datacube with traditional approaches. This analysis shows the benefits of Datacube approach in terms of data availability, storage overhead and bandwidth in different adversarial environments.

Chapter 5 This chapter presents the results obtained in our practical experiments. Our experiments focus on two main goals. First, to compare the performance of Datacube in different adversarial environments. Second, they aim at showing how the hybrid redundancy scheme of Datacube can affect its performance.

Chapter 6 This chapter concludes this work and identifies future research directions.

Part I

Context: The State of Art

Chapter 2

Data Storage in Peer-to-Peer Systems

Contents

2.1 Peer-to-Peer Systems

2.1.1 What is a Peer-to-Peer System - A Bit of Recent History

There exist many definitions of peer-to-peer (P2P) systems used in the literature. In order to show this variation of terminology we cite few examples. First, Schollmeier [81] defines a peer-to-peer system as "a distributed network architecture where participants share a part of their own hardware resources (processing, power, storage capacity, network link capacity, printers, etc.) that are directly accessed by other participants, without passing by intermediary entities". Second, Steinmetz and Wehrle [87] define it as "a self-organized system of equal and autonomous entities called peers, which aims for the shared use of distributed resources in a networked environment avoiding central services". Finally, Shikey [83] proposes the following definition, "peer-to-peer is a class of applications that takes advantage of resources (storage, cycles, content, human presence) available at the edges of the Internet."

Despite the different interpretations, there is a general agreement on a minimal characterization of a peer-to-peer system which is *a decentralized, symmetric and self-organizing network of computers, in which resources (or services) are distributed and provided by all participating members*. A symmetric network is a system where each existing computer performs the same kind of tasks. Symmetric network is an alternative for the traditional client-server model where the network is divided into providers of resources called servers and consumers, called clients. In a symmetric network, computers are simultaneously clients and servers, which has led to the term *servents*. Note that, such a term is not popular nowadays and the term peer is preferred.

Peer-to-Peer systems became known world wide in 2000 when the Recording Industry Association of America (RIAA) sued Napster [65] for copyrights infringements. Napster [65] is a peer-to-peer file sharing application released in 1999 by Shawn Fanning while he was attending Boston's Northeastern University. The main goal of Napster is to provide an easy way of sharing music files. Specifically, Napster can be defined as an application-level network of computers whose resources are distributed over the members, instead of using a single server to provide this service. Each computer shares its local music files by publishing a list as a global index service. End-user computers are called *peers* in the Napster network. Peers looking for music files check the global index in order to find out which peers store the targeted files. After receiving a list of peers, it is possible to download the files directly from those peers storing them. Differently from the traditional client-server approaches, file exchange in Napster is performed between end-users, therefore the name *peer-to-peer*.

The concept of peer-to-peer computing was not first conceived with Napster. In 1980, Tom Truscott and Jim Ellis had already used this concept to create Usenet. Usenet was a world wide discussion system built over a set of loosely connected servers. In Usenet, each client could post information to the server it was connected to. Servers shared the information by passing it to one another. As such, servers were peers transmitting information to one another, similarly to the peer-to-peer approach. A distinguishing feature of Usenet with respect to the subsequent peer-to-peer systems is that, in the former one the exchange of data was started by the peer acting as a sender, while in latter one it was started by the receiving peer.

In less than one year, Napster had become the fastest growing application in the Internet reaching over 50 million users. However, in March 2001 Napster was forced to stop its

activity by a legal injunction. The ability to quickly shut Napster services down was given due to its centralized structure. Without the centralized directory of indexes, the search capability of Napster peers is worthless. Napster represented the first generation of peer-to-peer systems that are characterized by the use of central servers to store indexes of files shared by peers. Such systems can quickly discover information. However, the scalability of this type of system depends on the capacity of central servers to respond to queries, and on the limitations of the database used to store these indexes.

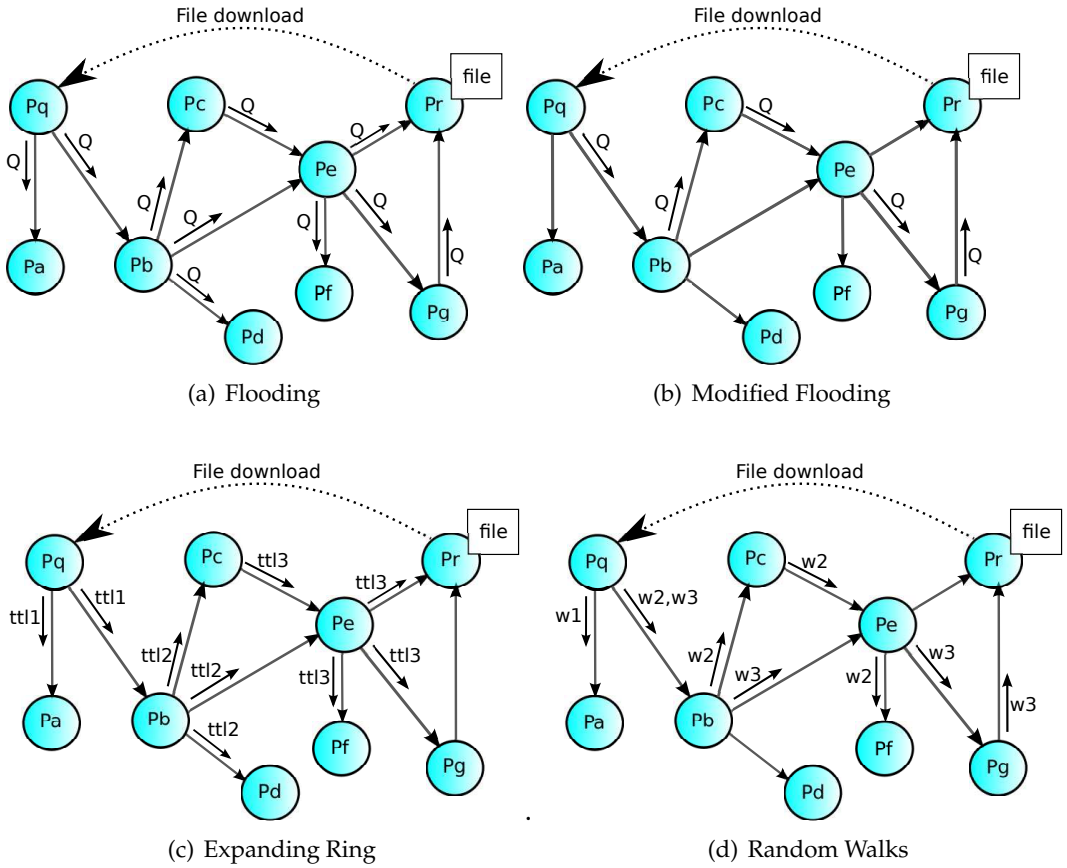
2.1.2 Unstructured Peer-to-Peer Architectures

Gnutella [1] is one of the systems that were later inspired by Napster. Gnutella is a file-sharing system that represents a second generation of peer-to-peer networks due to its distributed approach for searching and exchanging files. In order to perform a decentralized search, each peer joining the Gnutella network selects a random number of peers as neighbors. The resulting random graph formed by all logical links among peers represents the topology of the peer-to-peer overlay. A straightforward approach of performing this random selection is to disseminate a list of all existing peers in the system to joining peers, so that joining peers can randomly select peers from the global view of the system. In Gnutella, each peer joining the overlay receives a random set of existing nodes from the bootstrap service forming a random topology called *unstructured overlay*. Note that, Gnutella bootstrap service knows the set of peers that are currently online in the system.

The notion of *unstructured overlays* refers not only to the manner peers are connected to each other, but also how resources are placed at these peers. For instance, data placement evolves according to the behavior of existing peers, that means data files are placed according to the requests performed by existing peers. When peers search for files, the request is disseminated from peers to peers by flooding the network until reaching its destination. Specifically, a requesting peer p_q sends a query message to its neighbors. Any peer p_r receiving a query message for a file f replies to p_q with a query-hit message, if it holds the requested file f. Otherwise, p_r forwards the query to all its neighbors. By disseminating the query message in the network, eventually the existing file f is reached. Peer p_r holding the file replies to p_q with a query-hit message by using the reverse-path. When p_q receives back the query-hit message, it contacts the file holder p_r directly to download the file. Thus file f is now located at two distinct locations at peer p_r and peer p_q. Figure 2.1(a) shows an example of search using a flooding approach. This approach is effective at locating files due to the high degree of connections among peers. However the efficiency of the search cannot be ensured because searching files demands a large number of network hops and it creates a large number of query messages per request. Therefore, unstructured peer-to-peer overlays face a scalability problem as the number of messages per query and the workload at each peer largely increases with the number of peers in the system [56].

Following Gnutella, subsequent architectures were proposed to reduce the overhead of query messages in unstructured overlays. Instead of disseminating the query requests to all neighbors, Kalogeraki et al. [44] propose to forward the queries only to a fraction of neighbors. Figure 2.1(b) shows this approach. From peer p_q to peer p_r, at each hop request Q is forwarded only to a fraction of neighbors. Lv et al. [58] propose an expanding ring approach. In this approach, a TTL (i.e., Time To Live) is associated to each query by the requesting peer. At each hop this TTL is decreased. In order to search files, peers continually send query

searches starting with small TTL values and increase it until a query-hit reply is received as shown at Figure 2.1(c). By increasing the TTL value at each round, query messages eventually find the requested file and return it to the requester. Chawathe et al. [18] propose a random walk approach in order to drastically reduce the number of messages disseminated in the overlay network. In this approach, at each hop only one neighbor is randomly selected to forward the query and the TTL value is decreased. The query is forwarded while the TTL value is greater than 1 and the file is not found. If the requested file is reached, the requesting peer p_q downloads the file directly from peer p_r holding the file. Otherwise, a new search is started by p_q. Each search is called a *walk* and each walk takes a different search path in the network. Figure 2.1(d) shows an example of random walks, where three different walks are triggered by p_q to find the requested file at peer p_r.



(a) Flooding (b) Modified Flooding

(c) Expanding Ring (d) Random Walks

Figure 2.1: Search in Unstructured Overlays

All these approaches aim at reducing the number of messages disseminated in the network. However the latency to reach requested files is largely increased because good paths towards the requested file may not be found due to the reduced and random choice of neighbors to forward the requests to. Moreover, in such approaches many duplicated paths can be used, since the requesting peer does not learn about failures of previous requests. The success rate is highly variable and depends on the current network topology. Therefore, unstructured overlays still show scalability issues.

2.1.3 Structured Peer-to-Peer Architectures

The unprecedented growth of popularity of unstructured peer-to-peer overlays called the attention of the academic community. Structured models started being proposed to cope with the scalability issues of the unstructured approaches. Also known as the third generation of peer-to-peer systems, structured overlays present a well defined topology and resource management. Aberer et al. [2] presented a generic model to describe the structural and functional model of all structured peer-to-peer overlays. This model is detailed in the following.

Unlike unstructured overlays, in structured ones peers are logically organized into a structured graph. Peers must provide access to all their resources. In the following $\mathcal{P}$ represents the set of all peers existing in the system and $\mathcal{R}$ the set of all accessible resources. The resources in $\mathcal{R}$ are deterministically placed at peers in $\mathcal{P}$ as follows. Peers $\mathcal{P}$ and resources $\mathcal{R}$ are both mapped into an identifier space $\mathcal{I}$. For any peer $p \in \mathcal{P}$ and for any resource $r \in \mathcal{R}$ a unique identifier $i \in \mathcal{I}$ is assigned. A function $\mathcal{F}_i$ is used to map peers and resources to the same identifier space, i.e., $\mathcal{F}_i : \mathcal{P} \rightarrow \mathcal{I}$ and $\mathcal{F}_i : \mathcal{R} \rightarrow \mathcal{I}$. Identifiers associated to resources are called *keys*. Identifiers are used to establish an association among peers $\mathcal{P}$ and resources $\mathcal{R}$ using a deterministic metric. This metric function ensures that each existing peer is in charge of a specific subset of resources. For instance, this metric can be based on the geographical distance between any two nodes, or on the Hamming distance between peers and resources identifiers. Finally, to guarantee that resources can be accessed by any peer a graph of connections between peers is embedded in the identifier space. This graph is created based on structuring strategies that define neighborhood relationships and consequently the topology of the structured overlays.

Note that, structured peer-to-peer systems are actually distributed hash tables. A hash table is a data structure that uses hash functions to map identifiers called *keys* to their associated *values*. In structured peer-to-peer networks, the hash table is distributed over all existing peers. Peers and resources identifiers are *keys* and resources are *values* of the distributed hash table. Besides *join()* and *leave()* operations, respectively used to enter or leave the system, there are three other basic operations any peer should perform in a structured peer-to-peer overlay: *insert(key, value)*, *delete(key)* and *lookup(key)*. The *insert(key, value)* operation is used to add a resource to the system, with *key* representing the resource identifier and *value* the resource itself. Operation *lookup(key)* is used to retrieve the resource identified by the *key*. It returns the resource itself or the corresponding computer in the underlying network providing the requested resource. Operation *delete(key)* is used to purge the key from the system.

According to Aberer et al. [2], there are six aspects distinguishing structured overlays: Choice of the identifier space, mapping of peers and resources into the identifier space, management of the identifier space, graph embedding, routing strategy and maintenance strategy. Taking these aspects into account must guarantee the most important features of the structured approach of peer-to-peer overlays, namely efficiency, scalability, self-organization and fault-tolerance. We will discuss each aspect in the following. Prior to this we introduce some terminology.

2.1.4 Terminology and Definitions

Scalability: The concept of system scalability it not simple to define because it can involve many aspects of the system [41]. Bondi [16] defines system scalability as the ability of a system to accommodate an increasing number of elements or objects, to process growing volumes of work gracefully, and/or to be susceptible to enlargement. An intuitive notion of scalability is presented by Hill [41] which proposes to compare the system with a much larger version of the same system. In peer-to-peer systems, the scalability can be defined as the adaptability to changes in the system size (i.e., number of peers) without significant performance degradation [27, 2].

Self-Organization: The capacity of members of the system to organize themselves based only on local information without depending on any external coordination [27, 30].

Fault-Tolerance: A fault is malfunction that has many possible causes (e.g. design error, programming error, unexpected input, etc.). Some faults can induce system failures, others not. A fault-tolerant system is a system that operates correctly even in the presence of faults [52, 79]. In this work we consider crash, omission and byzantine faults. In the first case, system components suddenly halts and do not execute any further operation. In the second case, a process fails to respond to a request. Many reasons may lead a process to act like this. For instance, a send omission may happen when a send buffer overflows and the server handling the request is not ready for such a situation. The last case conducts the system components to arbitrarily work (e.g. the byzantine process can incorrectly perform a request, deliberately corrupt its local state or produce inconsistent outputs) [50]. Nodes or peers that exhibit Byzantine faults are called Byzantine nodes (or peers).

Byzantine Consensus Agreement: This is a fundamental problem in the domain of distributed fault-tolerant systems [35]. Suppose a distributed system composed of n nodes, such that m of which are byzantine. A byzantine node is a node that behaves arbitrarily. This behavior is either intentional (i.e., malicious behavior) or caused by a temporary fault. The consensus agreement is a process in which every node in n initially proposes a value and then they decide on a value [8]. The consensus agreement is defined by the three following properties: *i) termination:* every non-faulty process decides a value. *ii) agreement:* no two non-faulty nodes decide on different values. *iii) validity:* if all non-faulty nodes propose the same value v, then the value v is decided. Solutions to this problem does not exist if m does not satisfies: $m < \frac{n}{3}$ byzantine nodes in a system with n the total number of nodes.

Churn: Classically defined as the rate of changes in the system membership due to joins, graceful leaves and failures of peers [39].

2.1.5 Distinguishing Aspects of Structured Overlays

2.1.5.1 Choice of the Identifier Space

The identifier space $\mathcal{I}$ is the core of the overlay structure and it plays several roles in the network. First, it acts as an address space used to identify peers and resources. Second,

the address space also allows peers to communicate to each other independently of their physical location. Furthermore, the larger the identifier space $\mathcal{I}$ is, the more scalable the system will be. The identifier space must have a *closeness* metric $d : \mathcal{I} \times \mathcal{I} \rightarrow R$ associated to its identifiers, where R denotes the set of real numbers and this metric d (distance) must satisfy the following properties:

1. $\forall x, y \in \mathcal{I} : d(x, y) \geq 0$.
2. $\forall x \in \mathcal{I} : d(x, x) = 0$
3. $\forall x, y \in \mathcal{I} : d(x, y) = 0 \Rightarrow x = y$
4. $\forall x, y \in \mathcal{I} : d(x, y) = d(y, x)$
5. $\forall x, y, z \in \mathcal{I} : d(x, z) \leq d(x, y) + d(y, z)$

The identifier space is a metric space if all properties are satisfied. Otherwise, the identifier space must satisfy at least the properties 1, 2 and 3. For instance, as shown later, in Chord [88] the identifier space can be a subset of N natural numbers and the metric distance d is defined by: $d(x, y) = (y - x) \bmod N$.

2.1.5.2 Mapping to the Identifier Space

A function $\mathcal{F}_i$ is used to associate each peer and resource present in the system to unique identifiers in $\mathcal{I}$. When mapping peers from $\mathcal{R}$ into unique identifiers in $\mathcal{I}$, the function $\mathcal{F}_i$ can be either static or dynamically defined. In the former case, it can be based on some specific peers attributes, for instance their physical addresses. In the latter case, it can use some time-based attribute, for instance the time peers join the network. Function $\mathcal{F}_i$ can also satisfy some specific distribution properties. When mapping resources $\mathcal{R}$ into unique identifiers, load balancing is significantly affected by the distribution of the identifiers of each existing resource.

2.1.5.3 Identifier Space Management

The set of current peers $\mathcal{P}$ in the overlay is responsible for managing the whole identifier space $\mathcal{I}$ as a part of the self-organizing feature of a peer-to-peer system. The identifier space must be partitioned among all existing peers. A function $\mathcal{M} : \mathcal{I} \rightarrow 2^{\mathcal{P}}$ is used to associate each identifier $i \in \mathcal{I}$ of a resource $r \in \mathcal{R}$ to a set of peers responsible for r. Among other features, function $\mathcal{M}$ must be complete in order to guarantee that all identifiers of resources are associated to at least one peer. As the set of peers $\mathcal{P}$ often changes due to the dynamics of the system, the set of peers responsible for a resource must also be greater than one, in order to guarantee fault-tolerance. A common approach used to associate identifiers and resources is to associate the resource with identifier i with its closest peers. Therefore, locating a resource r with identifier $i = \mathcal{F}_i(r)$ in the structured overlay corresponds to finding a peer $p \in \mathcal{M}(_i)$ which is the closest peer to the identifier i.

2.1.5.4 Graph Embedding

Contrary to the random approach used by unstructured overlays, in structured overlays peers are deterministically connected to each other in order to define the topology of the overlay. The whole system is modeled as a directed graph $\mathcal{G} = (\mathcal{P}, \mathcal{E})$, where $\mathcal{P}$ denotes the set of existing peers in the overlay, while $\mathcal{E}$ represent the set of edges connecting these peers. A function $\mathcal{N} : \mathcal{P} \rightarrow 2^{\mathcal{P}}$ is used to define the neighborhood relationship. For any peer $p \in \mathcal{P}$, $\mathcal{N}_{(p)}$ represents the set of peers that have a connection with p. Specifically, for all q $\in \mathcal{N}_{(p)}$, it exists an $e \in \mathcal{E}$, where $e = (p, q)$ represents a direct edge between p and q. The properties of the structured overlay depend on the properties of the graph generated by $\mathcal{N}$. For instance, graphs with small diameters are more efficient in terms of number of hops to route messages through the network. Moreover, graphs providing vertices with minimal out-degrees is a way to ensure fault tolerance. Keeping a minimal out-degree guarantees that the existence of an alternative path to forward the message towards the destination if the next hop fails. Another important property is the maximal out-degree. Defining a maximal out-degree it ensures a bounded maintenance cost of outgoing connections because the map of neighbors must be updated whenever a neighbor peer leaves the system.

2.1.5.5 Routing Strategy

Routing is the basic service provided by a peer-to-peer overlay and in particular by structured overlays. By using the graph of connections, any peer $p \in \mathcal{P}$ can reach any other peer or resource in the overlay by routing messages to it. The routing process *route(p,i,m)* is a distributed process in which a message m is forwarded to a peer $p \in \mathcal{P}$ responsible for an identifier i, with $p \in \mathcal{M}_{(i)}$. In order to route message m the neighborhood relationship $\mathcal{N}$ is used. From the origin to the destination, at each peer $p \in \mathcal{P}$ in the path, the message is forwarded to a set of *next peers* $\mathcal{C}(i, p) \in \mathcal{N}_{(p)}$. The choice of the set $\mathcal{C}(i, p)$ may vary among different implementations of structured overlays. However, a common approach is to send message m to the peer q which is metrically closer to the destination peer p, responsible for i. Moreover, the performance evaluation of routing algorithms on different structured overlays can be analyzed in function of the number of hops and probability of successfully reaching the destination.

2.1.5.6 Maintenance Strategy

Churn is one of the distinguishing characteristics of a peer-to-peer system. The overlay must adapt itself according to this dynamic condition. As membership constantly changes the churn threatens the integrity of the overlay structure. Routing tables containing neighborhood relations may quickly become out of date. A maintenance strategy is required for the system to repair itself from these inconsistencies. This means that routing tables must be constantly repaired in order to maintain the basic services (e.g. routing service). Different mechanisms can be used to implement a maintenance strategy, either pro-actively or reactively. The strategy must guarantee a good level of consistency of routing tables to ensure the connectivity of the embedded graph. It is important to remind that the usability of a structured overlay can be defined by the efficiency of the maintenance strategies.

2.2 Existing Structured P2P Architectures

In this section we present some important topologies of structured overlays. The presented structures are used in the main structured data storage architectures. The discussion follows the aspects mentioned in the Section 2.1.3 for each topology.

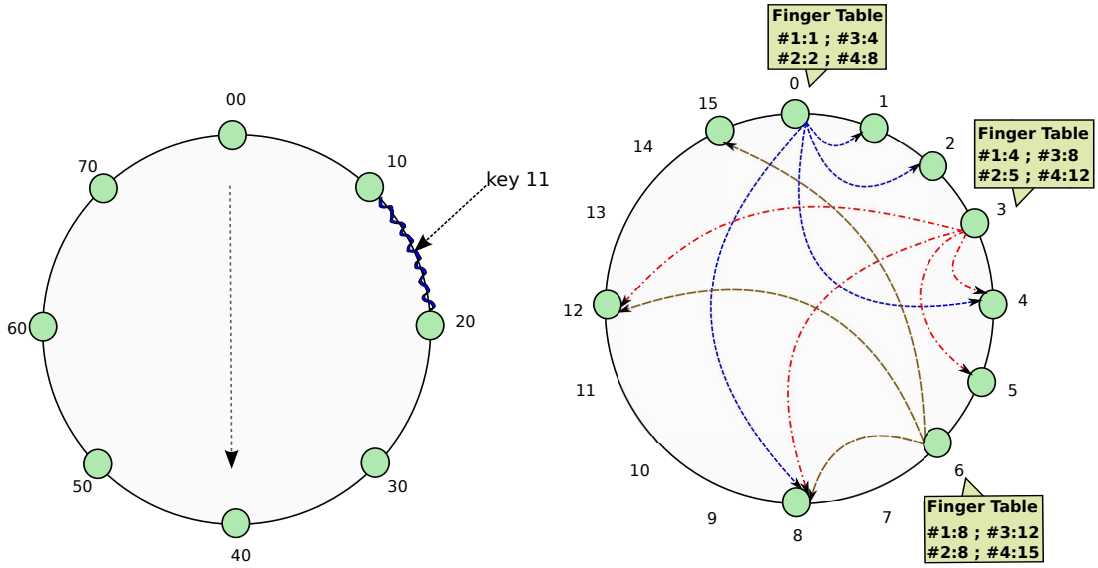
2.2.1 The Overlay as a Ring

Chord is the first structured overlay based on a ring topology. It has been proposed by Stoica et al. [88]. Chord is a scalable protocol that efficiently and correctly handles lookups of identifiers in the structured overlay. Besides its provable correctness and performance, Chord is distinguished from other protocols due to its simplicity.

The identifier space $\mathcal{I}$ in Chord, is a set of 2^m natural integers. All identifiers are ordered in a circle modulo 2^m, as shown at Figure 2.2(a). The metric associated to the identifiers is defined by the clockwise distance among them. That is, $\forall x, y \in \mathcal{I} : d(x, y) = (y - x) \bmod 2^m$. All peers and resources in Chord are assigned an m-bit unique identifier obtained from the circular ID-space. The identifier 2^m must be large enough to guarantee a negligible probability of having two peers or resources with the same identifier. Chord uses a hash function SHA1 [86] to map peers and resources to the circular identifier space $\mathcal{I}$. Each identifier has a specific position in the ring according to its identifier. Each peer, also called node in the Chord ring, is in charge of one or more keys. A peer is said responsible for key (k) if its identifier is the first identifier that follows k in the identifier ring space. This node is called *successor(k)*. For instance, given a key 11 as presented at Figure 2.2(a), if peer 20 is the first peer with identifier succeeding key 11 in the ring, then peer 20 is called successor of 11 and it is responsible for the key 11. Moreover, peer 20 is responsible for any key x whose identifier is in the interval between peers 10 and 20 (i.e., peer 20= $successor(x) \rightarrow 10 < x \leq 20$). The metric d is used to verify whether a node ID_c is the successor of ID_b or not. The ring overlay is formed by nodes as a cyclic directed graph.

The goal of Chord's protocol is to route lookup requests to the node responsible for requested keys. The routing strategy of Chord is simple, at each node the lookup request must be forwarded to the node whose identifier is closer to the requested key. To achieve this, Chord uses just a few data structures at each node. A list of r immediate successors nodes in the ring is maintained at each node. If every node knows at least one successor node in the ring, it suffices to guarantee the correctness of the lookup process. Requests are passed around the ring. Then eventually the node responsible for the key is reached. Chord uses r successors instead of only one to make the protocol fault tolerant. Chord also maintains a predecessors list with entries pointing to r nodes that precede the current node in the ring. Predecessors lists are important to absorb changes in the system membership.

However, lookups based only on *successor lists* are not efficient, requiring on average $N/2$ messages to reach a key, with N the number of nodes in the system. To reduce the number of messages and increase the efficiency of the protocol, Chord uses another data structure at each node, called *finger table*. Each finger table has at most m entries. Each entry i in the finger table contains the identifier of the node that succeeds the current node by at least 2^{i-1}. Specifically, at each node n there is a finger table FT_n. The finger table FT_n also has at most m entries. Each entry i in FT_n corresponds to the identifier of the node s such that $s = successor(n + 2^{i-1})$, with $1 \leq i \leq m$. Then, at node n instead of



(a) Identifier Space Management in the Chord Ring. (b) Chord Finger Tables

Figure 2.2: Examples of the Chord ring topology.

forwarding the lookup message to $successor(n)$, node n seeks in its finger table the node whose identifier immediately precedes the requested key and forwards the message to this node. Figure 2.2(b) shows an example of a Chord ring with $N = 2^m$ elements, with $m = 4$, allowing this ring to have up to 16 peers. The system has currently 10 peers ordered in the Chord ring according to their identifiers. At each peer, the finger table has 4 entries. As mentioned before, each entry i must point to the peer that succeeds the current peer by at least 2^{i-1}. Then, at peer 0 entry 1 points to peer 1, entry 2 points to peer 2, entry 3 points to peer 4 and entry 4 points to the peer 8. At peer 3, the third entry should point to peer 7. Since peer 7 is not currently present in the system, entry 3 points to the next successor which is peer 8. The same can be seen at the finger table of peer 6. It has been proven by Stoica et al. [88] that by using these data structures any request can be successfully completed in a number of hops that grows logarithmically with the size of the system.

To cope with nodes leaving and joining the ring, Chord has a maintenance strategy called stabilization protocol. The protocol guarantees the connectivity of the system under churn by maintaining the successor and predecessor lists up to date. In order to do that, Chord nodes run the stabilization protocol either periodically or when triggered by a specific event. During a *join* operation, a joining node n first seeks for its successor n_s by requesting another node n' in the ring that n is expected to know (e.g. obtained from a public bootstrap server). Then, node n runs the *stabilize* function of the stabilization protocol in order to update the successors and predecessors lists. The stabilization runs as follows: First node n notifies its successor n_s of being it new predecessor, n_s removes the link it has to node n_p (i.e., its current predecessor) and accepts n as its new predecessor. For a short period, node n_p keeps its successors list with n_s as its successor node. Then, n_p asks n_s its predecessor list and finds out that n is the new predecessor of n_s, it makes n_p change its successor list to acquire n as its new successor. Finally, n_p notifies n that it is its new predecessor and all successors and predecessors become up to date. Finger tables are also updated from time-to-time, but the impact of

obsolete finger tables in the performance of the protocol is less important than obsolete successor lists in the correctness of the Chord structure. Each node n updates all the entries in the finger table by searching for the ith successor of n such that $i\text{th}= findSuccessor(n+2^{i-1})$, with $1 \leq i \leq m$).

2.2.2 The Overlay as a Tree

Tapestry is the first structured overlay to use a tree structure to provide resource location and routing. Tapestry was proposed by Kubiatowicz et al. [96] and it is based on location and routing mechanisms of Plaxton mesh.

Briefly, Plaxton is a distributed data structure proposed by Rajaraman and Richa in 1997 [68]. Plaxton is organized as an overlay that provides message routing and objects (i.e., data objects) location. Plaxton is a static distributed data structure that does not assume membership changes. In Plaxton, objects and peers are randomly named by a fixed-length bit sequence in the same base. Any object has a root peer and roots are deterministically chosen from existing peers in the system. It implies that Plaxton must have a global view of the system before associating objects to root peers. Plaxton is represented by a set of embedded trees with the root of objects representing the root of these trees. Each peer in Plaxton has static neighbor maps used to find the peer which has one or more digits in common with the destination peer. The goal of the routing algorithm is to route messages to the closest peer to the destination. Therefore, each peer is a root for a different routing path.

In 2001, Tapestry adapted Plaxton to provide availability and resilience to failures in dynamic peer-to-peer networks. The identifier space $\mathcal{I}$ of Tapestry is an identifier space $\mathcal{I}$ using identifiers of base b. Each peer and resource, currently in the system, receive an identifier from this identifier space $\mathcal{I}$, for instance a sequence of 160 bits represented by 40 digits in the hexadecimal base. Peers identifiers are leaves in a tree of height $\log_b(N)$, with N the current number of peers. The metric distance among any two identifiers is the height of their smallest common subtree. Then, each peer p has $\log_b(N)$ neighbors, such that the hth neighbor is at distance h from p. Figure 2.3 shows an example of a simple binary Tapestry tree (i.e., base=2). In this example, the maximum height of the tree is 3. Therefore, the maximum distance among any two peers is 3 hops. Peer with ID=110 has peer ID=111 as its first level neighbor, because the smallest common subtree among them is of height 1. Similarly, peer ID=000 has peer ID=011 as its second level neighbor, since the smallest common subtree among them is of height 2.

In Tapestry, identifiers are obtained from the hash function SHA1 [86] as in Chord. In order to define the peers responsible for specific keys, Tapestry proceeds as follows: Each resource has a root key which is the root of a tree. Similar to Plaxton, the peer responsible for a key is called *root peer*. This peer has the identifier with more digits in common with the root key. Specifically, any peer P_m storing a resource k publishes that it holds the resource k to the key's root K_{root}. Note that, publishing is defined as the process of creating a pointer in the root peer K_{root} such as a pair $\langle k, P_m \rangle$, informing the resource whose identifier k is stored in the peer whose identifier is P_m. Tapestry does not use a single root peer K_{root} for each resource to avoid the single point of failure in each root. Tapestry assigns different and deterministically defined roots for each resource. To do so, an auxiliary integer number i is used to derive n different roots such that $1 \leq i \leq n$. Each value of i generates a different value of root k_i for the same resource. Each i is added to the resource's identifier and then mapped to the

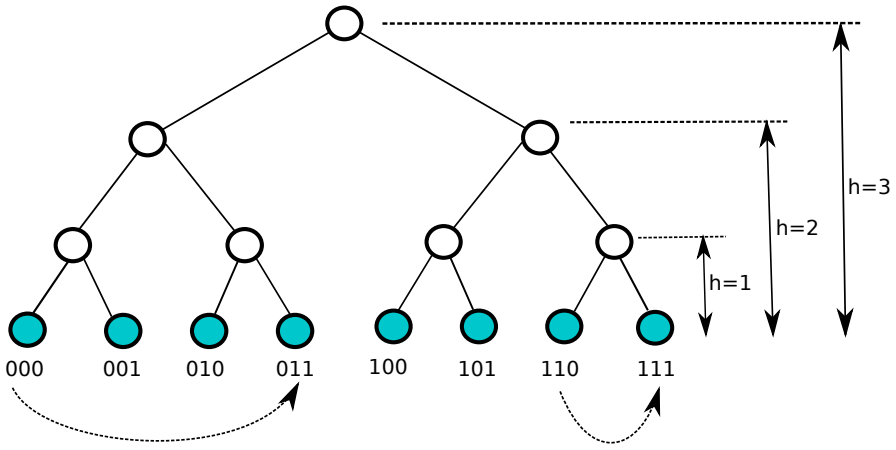


Figure 2.3: Tree structure of Tapestry overlay with identifiers of base b=2.

same identifier space $\mathcal{I}$. The resulting n values are the redundant roots used to publish the resources in Tapestry.

Another improvement of Tapestry is related to the connections among existing peers. Tapestry supports a dynamic membership. Neighbor maps are dynamically created at each Tapestry peer. These maps are used later by the routing protocol. Each joining peer P_j builds its own neighbor map. In order to create a neighbor map, joining nodes are supposed to know some existing node in Tapestry, called gateway. The knowledge of existing nodes is easily achieved either by a bootstrap service or an out-of-band communication mechanism. By using a gateway P_g, the joining peer P_j seeks to route a message to its own identifier ID_j. At each hop from P_g to P_j, a neighbor map of P_j is built by copying each ith level of the neighbor's map in the ith hop. For each level copied, P_j optimizes each entry by checking if the primary neighbors are closer to P_j than the secondary neighbors. Peer P_j can optimize closer neighbors based on these copied neighbors maps according to the distance between P_j and neighbors entries. This process is repeated until the next hop becomes empty. An empty next hop takes place when the routed message arrives in the closest neighbor to the joining peer P_j. This empty entry must point to the joining peer P_j itself. A notification is sent backwards by P_j through back pointers in order to promote P_j to a potential closer neighbor than old neighbors in their neighbor maps. All these neighbors maps are used to locate resources using the embedded resource trees.

In Tapestry, the routing mechanism is similar to the one used by Plaxton with some improvements. At each peer, neighbors maps are used to find the peer which has one or more digits in common with the destination key. The goal of the routing algorithm in Tapestry is to route a given message m to the numerically closest peer to the destination key id D_{id}. Specifically, any message m sent from the origin peer O_{id} to the destination peer D_{id} is incrementally routed digit-by-digit to the neighbor peer which is one-digit closer to D_{id}. Each peer has neighbors maps with multiple levels and entries used to route messages. Each level i represents a matching suffix at the position i in the ID. At each hop, the peer shares a suffix with length l with D_{id} and the next hop is chosen among the b entries in the level $l+1$. The message is sent to the neighbor which is the closest neighbor sharing $l+1$ suffix with the destination D_{id}. Figure 2.4 shows an example of routing in Tapestry [96].

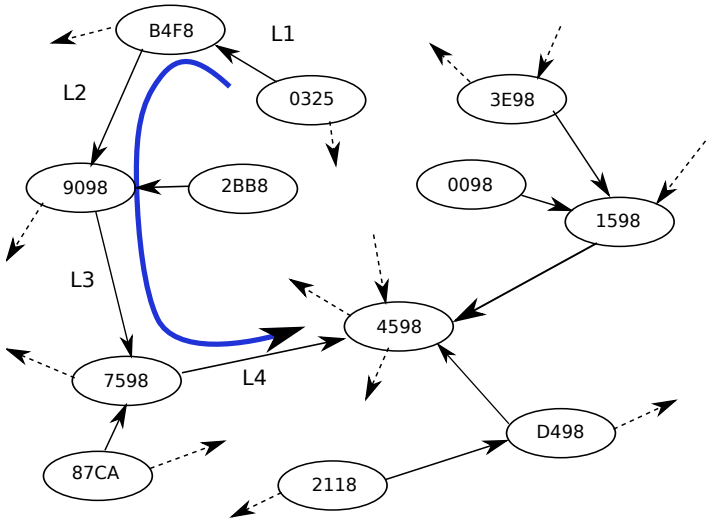


Figure 2.4: The example shows an overlay where a peer with identifier 0325 sending a message to another peer 4598. In this example, peer 4598 is a key root.

In this example, we assume that peer 0325 wants to send a message to peer 4598. Peer 0325 first looks in its first level which neighbor shares the destination suffix at level 1. Peer 0325 finds the neighbor peer B4F8, then sends the message to it. Then peer B4F8 which shares the suffix xxx8 with the destination peer proceeds by looking for another peer whose address shares one closer digit with the destination xx98, reaching 9098. The following nodes do the same looking for x598 and finally 4598 is reached. Hence, assuming to have consistent neighbors maps any key can be found in $\mathcal{O}(log_b N)$ hops, with N the number of identifiers of base b in the system. By requiring only one matching suffix at each hop, Tapestry provides a failure handler because failing peers can be easily routed around by choosing another neighbor with a similar suffix.

Tapestry implements a dynamic membership management, then a maintenance strategy is applied. For instance, in Tapestry each entry in the neighbors map maintains one primary neighbor pointer and two other backup neighbors pointers called secondary neighbors to tolerate possible peer failures during the routing process. In other words, if the primary neighbor fails the message is redirected to secondary nodes. Tapestry relies on TCP timeout to detect peer failures. In addition, each node has back pointers pointing to all the nodes that have itself as a neighbor. These back pointers are used to regularly send *I-am-alive* messages to peers that point to it, assisting nodes to identify failing neighbors. Note that, to avoid the cost of inserting/removing nodes that are transiently unavailable into/from routing tables, Tapestry initially sets any failure neighbor as an invalid peer and monitors the failure node during a predefined period called *second chance period*. By doind this, Tapestry verifies whether failures are repaired or not. In the negative case, the route to the invalid nodes must be definitely removed from the neighbors map.

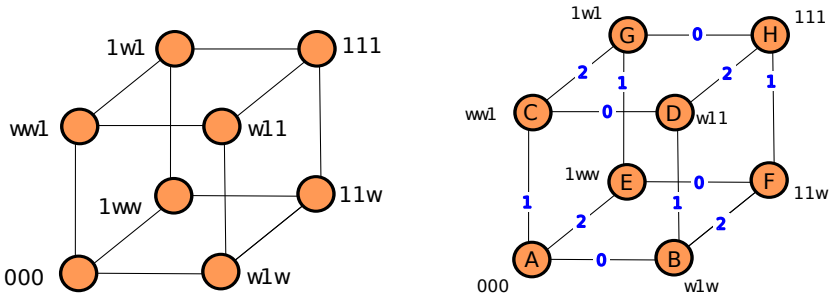
In Tapestry, all the replicas of a key are published in the key's root. Specifically, any peer P_n in the path from the peer holding the key to the root peer holds a replica of the key and publishes this information in the peer root. Consequently, in Tapestry a semantic flexibility is given to the application level which can define conditions to match in order to choose which

object should be returned in location requests. To maintain objects location despite possible failures Tapestry adopts the *soft state* approach [33] to keep all the pointers to its objects. In the soft state approach, if no refresh message arrives to ensure that the state of a pointer is alive the pointer is deleted. Hence, all the peers storing these objects need to republish the information about the location of objects.

2.2.3 The Overlay as a Hypercube

In 2002, Decker et al. [26] presented Hypercup, a peer-to-peer architecture based on a hypercube topology. In this topology, the overlay is structured in a d-dimensional space that is partitioned up into 2^d distinct zones. Each peer p is responsible for one specific zone and maintains links to other $\log(N)$ zones, with N the number of peers in the overlay.

The overlay is represented by a hypercube graph. A hypercube graph $\mathcal{Q}_d$ is a regular graph with 2^d vertices and $2^{d-1}d$ edges connecting one another. Each vertex has exactly d edges and is labeled (i.e., identified) by a set of elements such that, given any two vertices with label A and B, these vertices are linked by an edge if, and only if, labels A and B differ from each other by only one element. In Hypercup, vertices represent peers and edges represent connections among neighbor peers. Peers are identified by a $\log N$-bit identifier, with N the maximum number of peers in the overlay. Hence, dimension $d = \log(N)$ of the hypercube overlay denotes the number of bits used to identify peers in the system. Figure 2.5(a) shows an example of a hypercube graph with 3 dimensions. This is a complete hypercube since all vertices have the same number of $d = 3$ edges. In this example, we see each vertex connected to other vertices with exactly 1 different digit among their identifiers. For instance, vertex with ID=000 connected to vertices with ID 001, 010 and 100.



(a) A hypercube graph.

(b) An Hypercup example, with labeled peers and vertices.

Figure 2.5: Examples of hypercube structures.

Moreover, dimension d also defines the number of edges connecting each peer p to its neighbors. Given a peer p, the set $\mathcal{N}_{(p)}$ is the set of neighbors of p that contains peers with identifiers that differ from p by exactly one bit. These edges are identified by a label called *neighbor level* l (i.e., $0 \leq l < d$), as shown at Figure 2.5(b). In this example, the complete 3-dimensional hypercube has 8 peers represented by $\{A, B, \cdots, H\}$. Each peer has its corresponding unique global identifier of 3-bit length $\{000, 010, \cdots, 111\}$. And each edge is represented by a neighbor level $\{0, 1, 2\}$. We call peer B the 0-neighbor of peer A, as it is connected to A through the edge 0. Peer D is the 1-neighbor of B. Peer G is C's 2-neighbor.

Neighbor levels are important for dynamically building the hypercube. Since the number of peers in the system constantly changes, it is complex to maintain a complete hypercube dimension. Hypercup has construction and maintenance strategies called in the literature recursive construction of hypercubes [80] that adapt the dimension of the hypercube as peers come and go. It temporarily handles incomplete hypercube topologies according to the next biggest complete hypercube using neighbor levels.

From Figure 2.6(a) to Figure 2.6(h) examples of Hypercup handling peers arriving and leaving the system. It shows how Hypercup handles incomplete topologies by using neighbor levels. The construction and maintenance of Hypercup follows a few rules. First, peers can only have one neighbor per level. Whenever a peer joins the system, this peer must be integrated into the next free neighbor level of the peer which is handling the join operation. The peer handling the integration of a new peer is called the *integration controller*. The integration controller sets the new arriving peer to its next free level of neighbor, creating a link among them. At Figure 2.6(a) we assume that the system is initially composed only by peer A. When new peer B arrives, the peer A acts as an integration controller and answers the joining request from peer B. Peer A is the unique peer in the system, consequently peer B is integrated as its 0-neighbor. And peer A also becomes peer B's 0-neighbor. Now let us follow Figure 2.6(b) and assume that a new peer C arrives and sends a joining request to peer B. Peer B is the integration controller at this point. Peer 1 integrates peer C as its 1-neighbor level, since its next free neighbor level is level 1.

At this point (i.e., Figure 2.6(b)) peer A and peer C have only one neighbor level while peer B has two neighbor levels. However, during the construction and maintenance of Hypercup a second important rule must be followed. This rule says that, at the end of the integration process the joining peer must have the same level of neighbors as all existing peers. Consequently, Figure 2.6(b) shows that peer C is not yet completely integrated into the system. As shown in Figure 2.6(c), the integration controller must provide another neighbor level for peer C. Peer A becomes a temporary 0-neighbor level of peer C. It is temporary because peer A is already a 0-neighbor level of peer B, then it acts as if it had two different positions in the hypercube. Peer A becomes a 0-neighbor of peer C and a 1-neighbor of itself, as shown in Figure 2.6(c). This temporary function of peer A ends when a peer new peer D joins the system, for instance, as shown in Figure 2.6(d). At Figures 2.6(e) and 2.6(f) the Hypercub receives a new join request from peer E. In this case, peer A is the integration controller. Similarly to peer C, peer E does not fit in the current complete dimension of the hypercube. The integration controller (i.e., peer A) opens a new dimension to integrate peer E. Peer A sets peer E as its 2-neighbor level and provides temporary neighbor levels to peer E. Peers B, C and D are set as temporary neighbors of peer E. In this way, Hypercup admits temporarily incomplete topologies. Now suppose that a new join request arrives at peer A in the current topology shown in Figure 2.6(f). Following the current rules, peer A should open a new dimension and balance the neighbor levels creating new temporary ones. However, the current dimension is not already complete. Hence Hypercup must avoid the unbalanced topology to be continuously created before completing the current dimension. In order to do that, any peer receiving join requests must propagate the request to the system. By propagating the request, any unbalanced peer could take the integration control in charge avoiding new dimensions to be created without completing the current one.

When a peer suddenly leaves the system, the maintenance protocol decides which peer is going to take the responsibility for the position left by the leaving peer. The maintenance

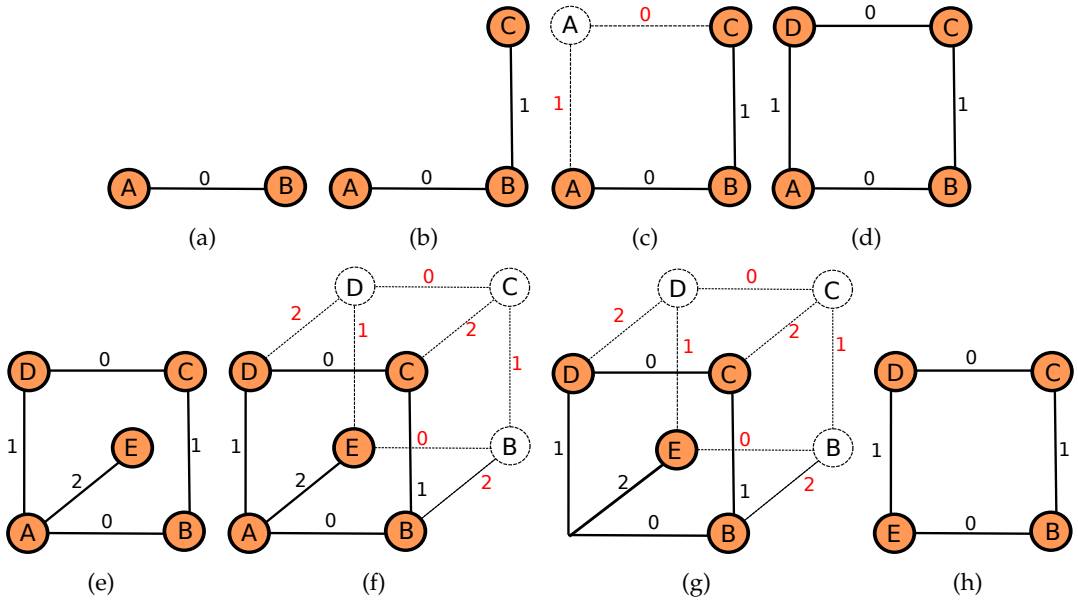


Figure 2.6: Example of a Hypercup structure inserting and removing peers.

protocol uses neighbor levels to substitute leaving peers and adapt the hypercube dimensions if necessary. The idea is similar to the second rule of the integration process: at the end of the process (i.e., removal process in this case) all the peers must have the same level of neighbors. Figure 2.6(g) shows an example of peer A leaving the system. In this example, peer A covers only one position in the hypercube topology. Then, only one substituting peer is needed when A leaves. In Figure 2.6(g) we see that peer E has the highest neighbor level (i.e., level 2), among all neighbors of peer A. Moreover, all the other neighbors of peer E are temporary neighbors. Then, peer E takes the responsibility of peer A and the dimension of the hypercube is reduced, as presented in Figure 2.6(h). Note that, whenever a peer responsible for more than one position in the hypercube leaves, the maintenance protocol must find substituting peers for all vacant positions.

Hypercup provides a high flexibility of routes (i.e., $\log N$ routes) among any two peers, which are called independent routes [80]. Similarly to other structured peer-to-peer overlays Hypercup provides logarithmic order of both routing table size at each peer and lookup performance. On the other hand, neighbor selection in Hypercup is not flexible, making the construction and maintenance protocol more complex than other topologies. Another weakness of Hypercup is that Hypercup is highly dependent on the hypercube structure, repairing mechanisms have a great impact on the robustness of the overlay. Whenever a single peer leaves, it takes several steps to redistribute the remaining peers. Anyway, Hypercup remains a good example to present the concept of hypercube overlays. In practice its performance could be severely affected in high churn environments.

2.2.4 Peercube: A Robust Hypercube Overlay

In this section we present Peercube [5]. Unlike Hypercup, Peercube is conceived to support the unpredictable high-rated changes in the system's membership. Due to its clustered

structure the degree of vertices in Peercube is lower than in Hypercup, leading Peercube to more efficient maintenance mechanisms. Peercube [5] is a self-organized structured overlay that prevents malicious peers from overthrowing the system as well as minimizing the effect of churn on the system's performance. Peercube is an overlay where peers gather together in cluster of peers and these clusters are organized on a hypercube topology. Different from Hypercup, in Peercube each vertex of the structured represents a group of peers, called clusters or clique. In order to limit the impact of churn in the whole system, Peercube classifies peers of each cluster into two different types: core and spare members. Different roles are attributed to each category. The set of core members are the elements that are actively involved in the fundamental operations of Peercube and they are responsible for maintaining the system's topology. Each core member stores two data structures called core and spare views. These views represent respectively the elements of the core and spare set of each cluster. Spare members do not participate in any fundamental cluster operations but they can store keys and corresponding data. Lookup, join and leave operations as well as maintenance of routing tables are examples of fundamental operations performed by core members.

In Peercube, each peer is assigned a m-bit random string of bits that represents its global unique identifier, obtained from the hash function of the peer's network address. Peers are organized into clusters according to a proximity metric applied to their identifiers. The metric used by Peercube is a logical metric based on XOR-distance between strings of bits. The rationale of using the XOR-distance approach is that it guarantees that for a given identifier p and a distance Δ there is only one identifier q at distance Δ. This is an important condition that guarantees that no more than one string of bits can be at the same distance Δ of a given identifier p, which is not satisfied with the Hamming distance. Clusters are also uniquely identified by labels corresponding to the position of the cluster in the hypercube topology. In Peercube, the label of a cluster is defined as the shortest common prefix shared by peers located in the cluster. The number of peers at each cluster is bounded. Each cluster has a minimum and maximum number of members, called S_{min} and S_{max} respectively. The lower bound is chosen such that it allows to implement byzantine tolerant protocols among the cluster members (i.e., $S_{min} \geq 3f + 1$ to tolerate f byzantine nodes), while the upper bound ensures the system's scalability (i.e., $S_{max} = \mathcal{O}(\log N)$, with N the total current number of peers in the system).

In Peercube, byzantine peers can devise strategies to successfully attack and pollute the core set and control the cluster operations. Since the presence of byzantine peers is supported by Peercube, the existence of few malicious peers could not affect cluster operations if a specific quorum is not reached. Hence byzantine peers can form coalitions in order to give their members benefits they could not grant themselves as individuals. We refer to this collaborative activity as collusion. It is important to note that in a cluster-based structure as Peercube the longer malicious peer can stay in the same position (i.e., in the core set) the greater is the probability of reaching specific quorums and overcoming the tolerated number of Byzantines.

Peercube also takes advantage of its hypercube structure to reduce the probability of having sent messages dropped by byzantine peers. To achieve that, Peercube profits from the property of independent routes of hypercube topologies [80]. Instead of choosing a single peer in the neighbor cluster to send a message to, it sends a message to sufficiently enough random core members in the neighbor cluster to guarantee that at least one correct peer will receive the message.

2.2.4.1 Peercube Operations

In this section we describe the main operations performed by Peercube.

2.2.4.2 Lookup Operation

As any other DHT overlay, Peercube provides a `lookup` function to search for $(key, value)$ pairs stored in the overlay. The `lookup` operation consists in recursively contacting the closest cluster to the identifier of the targeted key. If a key exists in Peercube the lookup operation retrieves the data associated to the searched key in $O(\log N)$ hops using $O(\log N)$ messages. To understand how the `lookup` operation is performed in Peercube, let us suppose that peer p searches for a key k in the system. Peer p has a m-bit identifier and belongs to the core set of cluster $\mathcal{C}$. Note, if p does not belong to the core set, the only difference in the protocol is that it needs to make this request to a peer that belongs to the core set. From cluster $\mathcal{C}$, at each hop one bit of p's identifier is modified to match the identifier of the target k and then the message is forwarded to the corresponding neighbor. By changing only one bit at each hop we know that the `lookup` request is walking one step towards the peer that holds the key k. In this way, the request is propagated until it finds a cluster labeled with the same prefix of key k, where the peer in charge of the key k is placed. Note that, if the request does not reach a cluster that perfectly matches the prefix of key k, it means that the last contacted cluster is the closest cluster to the searched key k and consequently this cluster should hold the key. In either cases, the last contacted peer returns the data corresponding to the pair $(key, value)$ to peer p.

Join Operation When a node p joins the system, it joins the cluster $\mathcal{C}$ whose label is the prefix of the p identifier. Thus cluster $\mathcal{C}$ is the closest cluster to the p identifier.

Before describing how the `join` operation is performed we introduced another type of peer that exists in Peercube, the *temporary* peer. Temporary peers are those peers that joined the network but their identifiers did not match any existing cluster in the system. During the `join` operation if a peer p must be inserted into a cluster $\mathcal{C}$ that does not yet exist, Peercube inserts p into the closest cluster to the label of $\mathcal{C}$. Temporary peers do not perform any operation and only core members are aware of their existence. The only thing that a temporary peer can do is wait until the condition to create the cluster it belongs to is satisfied. Once core members of a cluster detect that there are sufficiently enough temporary peers sharing a common prefix then a `create` operation is invoked and temporary peers join the new cluster. We will describe later how the `create` operation is performed.

Now, in order to join the system a peer p sends a `join` request to a peer p_y it previously knows in the system. In practice, the peer p_y can be easily obtained from a bootstrap service. Next, peer p_y forwards the `join` request of p to the cluster, among its neighbors, whose label is the closest label to the identifier of p. The `join` request will be routed until it reaches the cluster $\mathcal{C}$ whose label either matches p's identifier prefix, or is the closest label to p's prefix. In the former case, peer p is inserted into the spare set of cluster $\mathcal{C}$. In the latter case, it is inserted into the temporary set of cluster $\mathcal{C}$. Finally, the insertion information is broadcaster to all core members.

Leave Operation When peer p leaves the cluster, it simply leaves without performing any special task. Its absence must be detected by core members. Peercube requires that the depart of peer p must be detected by $\lfloor (2S_{min} + 1)/3 \rfloor + 1$ core members to avoid that malicious peers simulate nodes departure to try to devise some strategy for attacking the system. If peer p belongs to the spare set before leaving the cluster, its departure only needs to be detected by core members, as explained above. However, if p belongs to the core set, a maintenance procedure must be performed. This procedure consists in replacing $k - 1$ randomly chosen members of the core set with k peers randomly chosen from the whole cluster, where $1 \leq k \leq S_{min}$ (recall that S_{min} is the size of the core set).

Split Operation We have mentioned in Section 2.2.4 that clusters are lower and upper bounded. Once a cluster reaches these thresholds some actions must be taken. If the size S_{max} of a cluster is exceeded, a `split` operation is invoked and the cluster splits into two other clusters. Specifically, when peer p belonging to a cluster $\mathcal{C}$ detects that the threshold S_{max} is exceeded, it invokes a consensus among all core members to start a `split` process. If the agreement is achieved the split process starts. First, the shortest non-common prefixes shared by the identifiers of all elements in the cluster $\mathcal{C}$ are chosen to be the new labels of the new two clusters. Next, the core of both new clusters $\mathcal{C}'$ and $\mathcal{C}''$ are fed with core elements of the splitting cluster $\mathcal{C}$, according to the proximity metric. Then, the core set is complemented with peers from the spare set of $\mathcal{C}$ that are randomly chosen through a consensus agreement, exactly as performed during the leave operation. After that, each core member in the clusters must update its core and spare views to reflect the new membership sets, by removing peers that do not share prefix with the new label. Keys and corresponding data are also updated. Finally, the routing tables of clusters $\mathcal{C}'$ and $\mathcal{C}''$ are updated as well as the routing table of the clusters that was previously connected to $\mathcal{C}$. In order to update the routing tables of clusters that had connections towards $\mathcal{C}$, each core member in both clusters $\mathcal{C}'$ and $\mathcal{C}''$, that used to belong to $\mathcal{C}$'s core set contacts all the clusters stored in the predecessor's table of $\mathcal{C}$ to update their routing tables considering the distance between these predecessor clusters and the new clusters created.

Merge Operation The `merge` operation is performed once the number of peers in the cluster drops under the minimal threshold. The goal of the `merge` operation is to combine the cluster with all the clusters that share the longest common prefix with it. For the easiness of explanation, let us denote $c_0, c_1, \cdots, c_{d'-1}$ the label of a cluster $\mathcal{C}$. Once a peer p, that belongs to the core set of cluster $\mathcal{C}$, detects that the number of peers is under S_{min} it invokes the `merge` operation. First, p searches for all clusters that share $c_0, c_1, \ldots \bar{c}_{d'-1}$ as its label prefix. This search is based on a constrained flood approach which guarantees that a cluster is never contacted more than once and the search keeps a constant number of messages. Upon receipt the searched set Γ of clusters that share the longest common prefixes, p propagates the `merge` request to all members in the core set of $\mathcal{C}$ together with the set Γ. This is done to inform all the core members that they need to merge with Γ. Next, the core set of the cluster with the lowest label among all the clusters in $\Gamma \cup C$ is taken as the core set of the new merged cluster M. Then, the spare and core members of the remaining clusters are added to the spare set of M. After that, the keys from the merging clusters are copied to the core and spare members of M. And finally, the new routing table of M and the routing tables of clusters that used to point to the clusters $C \cup \Gamma$ are updated. The last ones must now point towards the new

cluster M. In order to do that, the predecessors tables (of each merging cluster) are used to send the update request to clusters that are pointing to all clusters in $C \cup \Gamma$.

Create Operation The `create` operation consists in the following steps. Let us denote p the peer belonging to the core set of a cluster $\mathcal{C}$ that has detected the condition to create a new cluster. First, peer p looks for the shortest common prefix shared by all the temporary peers in $\mathcal{C}$. This prefix is going to be the label of the new cluster N. Next, a consensus agreement is performed by all core members in $\mathcal{C}$ in order to choose S_{min} elements among all the temporary peers sharing the prefix with N's label. The agreed S_{min} peers are set as core members of the new cluster N and the remaining peers are set as spare members of N. Then, p verifies if the remaining temporary peers in $\mathcal{C}$ (as well as keys and their corresponding data) are closer to N than $\mathcal{C}$. If they are closer to N, these temporary peers and keys are assigned to N and removed from $\mathcal{C}$. After that, the routing tables of each core member of N is created based on N's label and its dimension d. Each of the d entries in the routing table points to the closest cluster to $N^{\bar{i}}$. Lookup operations are performed to find the closest cluster of each entry. Finally, the predecessors table is filled by using the search based on the constrained flood approach, similar as used in the `merge` operation to find the cluster that should point towards N. The total cost in number of messages exchanged to perform the `create` operation is $O(\log^2 N)$ messages [5].

2.2.5 Conclusion

In this section we have presented the topologies of peer-to-peer overlays. We have seen that structured overlays are more efficient than unstructured ones in terms of resource retrieval and scalability. On the other hand, when facing high churn they require a significant overhead of maintenance traffic. We have seen that clustered based DHT overlays such as Peercube, minimize the influence of churn and tolerate the existence of byzantine peers in the overlay. Next, we present some existing storage architectures based on peer-to-peer systems.

2.3 Storing Data in Structured P2P Systems

This section is devoted to the description of important properties that a storage system must present in order to provide persistent storage.

2.3.1 Introduction

In the recent years the demand for data storage has substantially increased. With the exponential growth of Internet usage, data storage has become one of the fastest-growing areas in last few years. The use of applications beyond email and file transmissions like social networks, video streaming, telephone, TV, photo sharing, among others, has required a massive amount of data to be stored. A work started in 2007 called Digital Universe [43] is trying to measure the amount of digital information created every year. Last year, the project registered a new record with 62% of growth during the year, impressively reaching 800 exabytes of digital data (i.e., 800 billions of gigabytes). It is expected that this number should reach

this year almost 1,8 zettabytes (i.e., 1,8 thousand of exabytes) and by 2020 an extraordinary growth to 35 zettabytes of digital data. They also have shown that in 2011 the cost of managing and storing information dropped to $\frac{1}{6}$ of what is was in 2005. These conjectures show how storage systems must be prepared to scale and keep data preserved. There are many advantages (e.g. reliability, load balancing, incremental growth) for using distributed systems to provide data storage instead of expansive centralized architectures. Peer-to-peer systems are naturally suitable for large scale systems and can play key roles on future architectures.

2.3.2 Properties of Data Storage in P2P Systems

In this section we present two important properties and one mechanism that any storage architecture should meet. In later sections we use these properties to compare existing storage architectures and address potential problems with them.

2.3.2.1 Data Availability

Availability is the property that qualifies data as ready for immediate use. In traditional storage systems availability properties are mainly based on the MTBF (i.e., Mean Time Between Failures) and MTTF (i.e., Mean Time To Fail) which are parameters of the hardware components in charge of the data. In storage systems based on peer-to-peer architectures, the total amount of stored data is distributed among highly dynamic peers that are constantly joining and leaving the system. This churn clearly affects the availability of data. Then, besides individual hardware failures at each peer, churn must be strongly considered in order to achieve the desired level of data availability. In order to achieve data availability, redundancy is required [75]. There are two main schemes [14, 19, 93, 19] used to provide data redundancy:

- **Replication:** Replication is the simplest scheme used to make data redundant. It consists in creating full replicas of data objects (e.g. data files) and placing each copy at m different peers in the system. Therefore, in order to guarantee data availability it suffices that at least one of these m peers is available. The level of availability is defined as the probability p that a data file D is currently available. To derive the level of availability using the replication scheme, we denote by Y the random variable representing the number of available replicas of D, and $P(Y \geq y)$ the probability that at least y replicas are available. For a given p, representing the average probability that a peer is currently available in the network (i.e., peer availability), the probability a that at least one of m replicas is available is defined by:

$$a = P(Y \geq 1) = 1 - (1 - p)^m \tag{2.1}$$

  From this equation, we can also derive the number m of replicas to create in order to guarantee a specific level a of availability, which is defined by:

$$m = \frac{\log(1 - a)}{\log(1 - p)} \tag{2.2}$$

  The replication scheme is also distinguished by its easiness of implementation. The only requirement to implement replication is to create new copies of the original data.

- **Erasure Codes:** Erasure coding is more complex than the replication scheme. It consists in dividing the data objects (e.g. data files) into blocks and introducing redundancy in these blocks. Specifically, to apply the erasure coding scheme to a data object D, D is first divided into k blocks. These k blocks are coded into s redundant blocks, with $s > k$. The ratio r of the coded redundant block s and the original number of blocks k is called stretch factor. The stretch factor can be compared to the number of replicas created in the replication approach. It is a key parameter that allows to understand the overhead introduced in the system. Coded redundant blocks are placed at s random peers in the system. The main property of optimal erasure codes is that the original data D can be recovered from any combination of k blocks among those s coded blocks. Consequently, when using erasure coding scheme the availability of the data depends on the availability of any k peers among those s peers holding coded blocks. As shown in [54], for a given p representing the expected probability that a peer is currently available in the network, the probability a that a given data file D is available is defined as

$$a = \sum_{i=k}^{s} \binom{s}{i} (p)^i (1-p)^{s-i} \tag{2.3}$$

with k the number of blocks the data D was fragmented into and s the number of coded fragments generated with erasure codes. In [19], the stretch factor r of erasure coding scheme is derived as follows.

$$r = \left(\frac{\sigma_a \sqrt{\frac{p(1-p)}{k}} + \sqrt{\frac{\sigma_a^2 p(1-p)}{k} + 4p}}{2p}\right)^2 \tag{2.4}$$

where σ_a is the value of standard deviations of a normal distribution for a required level of availability (e.g. $\sigma_a = 3,1$ for 99,9% of availability). Note that, the terms "redundant blocks", "coded blocks" or "check blocks" are equivalently used to express redundant coded blocks.

Some research works have already compared both schemes [19, 13, 75, 93, 54]. There is a consensus among these works that erasure coding saves more storage space than replication when comparing same levels of availability. However, some specific points must be considered. Rodrigo and Liskov [75] argue that coding, decoding and redundancy maintenance introduced by erasure codes are complex operations. Moreover, erasure coding scheme requires a significant download latency when using environments like the Internet since a minimum number of coded fragments need to be recovered before using the data. The larger the file is, the larger the latency to recover it will be. In replication scheme this latency is less significant since only one source need to be contacted and clients can choose the closest replica source to download the data from. In [19] the authors argue that replication saves more bandwidth than erasure coding to keep the data availability in a given level if the average availability of peers is higher than 48%. In Chapter 3 we discuss in details the use of erasure codes for data storage.

2.3.2.2 Reliability

According to the Communications Standard Dictionary [94], system reliability is the probability that a system satisfactorily performs the tasks for which it is intended to do, for a given period of time and environment. In the context of data storage, this property certifies that data is preserved as created. Clearly, errors can cause data to be disrupted reducing the reliability of the system. In distributed peer-to-peer systems, errors can occur either by chance or deliberately. For instance, assume that peer A sends a file f to be stored at peer B. Communication errors can prevent file f to be accurately saved at peer B. Another example, individual malicious peers in charge of data objects can delete data intentionally in order to save their local storage space. This behavior of benefiting from the system without contributing to it is called *free riding* in peer-to-peer literature. Moreover, these malicious peers can also attack the system by polluting the system with useless data. This is a common type of attack usually applied to systems that require an amount of shared storage from users, in order to provide storage resource to them. Besides individual attacks, a group of malicious peers can also collude in order to perform specific denial-of-service attacks. A denial-of-service attack is an attempt to prevent a computer resource to be accessed by its regular users. In storage architectures, such attacks, also called targeted attacks, are meant to prevent the data to be reached.

In the first property (*Data Availability*) we saw that data must be available whenever requested. However, besides its availability, data must also be reliable when retrieved. Therefore, high reliability is the second most important goal a distributed storage system must meet. To achieve reliability data must be completely free from errors or defects. Data Integrity must be checked in order to guarantee this accurateness. In order to protect the storage systems from errors or malicious attacks different security policies can be applied. Some of the proposed approaches used to ensure data integrity are: Hash functions, message digest, cyclic redundancy codes (CRC) and message authentication codes (MAC). In Section 2.4 we show how existing storage architectures handle data integrity in order to provide reliability.

2.3.2.3 Data Maintenance

Maintenance mechanisms are repairing methods used by storage systems to react to changes that jeopardize data availability and reliability. Repairing mechanisms are extremely important to preserve the levels of availability and reliability initially defined. Without repairing mechanisms data can be either useless or vanished from the system as time passes. These methods react to losses caused by the natural churn of peer-to-peer systems, communication errors among peers and malicious peers performing targeted attacks. In order to react to changes, storage systems implement different repairing mechanisms. Some systems implement a *threshold* based mechanism [97]. Specifically, in this approach the repairing mechanism constantly checks the stretch factor (i.e., number of replicas or check blocks, depending on the redundancy method used) of a data and eventually recreate all redundant data loss when the threshold is reached. This approach is also called *lazy* repair. Another type of repairing mechanism is referred to as *eager* repair. In this approach, the repairing mechanism reacts to every loss in the stretch factor by immediately recreating the lost replica (as well as check blocks). Note that, both approaches can be used to repair redundant data losses or

disrupted data.

2.4 Current Storage Architectures for Structured P2P

This section is devoted to the description of relevant architectures already proposed in the literature to provide persistent storage in structured peer-to-peer systems. For each redundancy scheme, we present a specific platform. That is, we present CFS [25] as a replication example, Reperasure [95] as an erasure coding example, and Total Recall [15] as a hybrid scheme (i.e., consists in replication and erasure codes together). Finally, we present Oceanstore [47] which is a storage system that aims at providing consistent, high-available and durable storage to billions of users. Oceanstore is also a hybrid scheme and one of the most cited works on distributed storage using peer-to-peer overlays. Oceanstore works as a pay-per-use global scale file system that supports read/write operations built mainly over untrusted peers. Oceanstore can tolerate malicious peers. It provides authentication, access control and an introspection mechanism to cache the data anywhere and everywhere to improve its performance. We describe these architectures according to their capability to meet availability, reliability and maintenance strategies.

2.4.1 CFS

Proposed by the Massachusetts Institute of Technology (M.I.T.), the Cooperative File System (CFS) [25] is a storage manager that benefits from the primitives of Chord in order to provide a scalable distributed storage. In CFS, distributed peers voluntarily join the system in order to use and share storage space. A DHT based on Chord is used to locate peers in charge of storing specific data. DHash is the layer of CFS responsible for distributing and storing data. Each CFS peer is a node in the Chord ring. In CFS, data objects are stored in blocks of data. Any data object (i.e., files or file system meta-data) is partitioned into fixed-size data blocks in the order of tens of kilobytes. These blocks are distributed and stored over the system and data requests are made by users in the File System layer. Note that, CFS blocks are not erasure coded blocks. They are simple partitioned fragments of the original data objects. In order to avoid any misunderstanding we will refer to these blocks as to CFS-blocks from now on. As mentioned in Section 2.3.2.1, by dividing data objects into smaller blocks it significantly increases the latency to retrieve it. CFS divides the data object into CFS-blocks to distribute the work load among several peers in order to provide load balancing. This is an important issue in the case of serving large popular files, because it reduces the influence of flash crowds (i.e., large spikes or surges in the network traffic) [6].

Specifically, peers and CFS-blocks are identified by a unique m-bit identifier UID from the same ID-name-space obtained from the consistent hash [45] mechanism used by Chord. A CFS-block with UID_b is stored at the peer C (identifier UID_c) if peer C has the closest successor identifier UID_c of the CFS-block UID_b. File system layer locally interprets a file as a set of CFS-blocks ID_s and requests these ID_s to DHash layer. Block requests are routed in the Chord layer using Chord routing and lookup protocols. Therefore, any CFS-block is retrieved with at most $O(\log N)$ hops, with N representing the number of peers currently in the system.

2.4.1.1 Data Availability

CFS uses replication as redundancy scheme based on Chord's successors list. Chord's successor list is a redundant key mechanism used to store pointers to the next r nodes at each peer in order to tolerate faults. The DHash layer of CFS uses the same idea to replicate each CFS-block stored at peer p to k peers succeeding p, with $k \leq r$. Therefore, as long as at least one out k replicas exists in the system the CFS-block will be available. Note that, replicating each *unit* at k peers, corresponds to creating k replicas of the original data. Then, CFS requires a replication factor k times larger than the amount of data stored in the system. When replying to requests, successor peers storing specific CFS-blocks send to requesters their own address together with the addresses of all the k peers holding replicas. On doing so, clients can choose replicas likely to be fast to download. A weakness of the CFS replication mechanism is the single point of failure created when the replication process is assigned to the first peer in the Chord successor's list. If attacked this peer can prevent the CFS-block from being replicated, jeopardizing the redundancy scheme. Besides that, a fraction of storage space at each peer is defined to be a cache storage space. The cache is used to reduce the download latency spent during the data location process. Cached CFS-blocks are set by peers after downloading a CFS-block or replica. Peers send these replicas (i.e., cached CFS-blocks) to the peers that are in the lookup path used to retrieve the CFS-block. Future requests tend to use these cached replicas that can be closer to the requesting peer, reducing the lookup and download latency.

2.4.1.2 Reliability

CFS uses three distinct mechanisms to protect the system against errors and malicious attacks. The first one guarantees the integrity of CFS-blocks in Dhash. CFS File System layer adopts a format proposed by Fu et al. [36] to authenticate CFS-blocks and the file systems where CFS-blocks are stored. Figure 2.7 presents how CFS ensures the integrity of CFS-blocks [25]. CFS-blocks are identified using a secure hash function (e.g. SHA1) of their contents, represented by $H(B1)$, $H(B2)$ and $H(Bn)$. Inode blocks have information about CFS-blocks including their identifiers. Directory blocks have information about inode blocks including their identifiers. The integrity of data blocks, inode blocks and directory blocks can be already verified just by checking their identifiers and the result of hashing of their contents. In addition to that, the root of the file system contains information about the directories stored as as *Root-blocks*. Different from other blocks, root-blocks are digitally signed and identified by the public key of the object's owner. The rationale of doing so is to guarantee not only the integrity of CFS-blocks but also the authenticity of the whole file system structure. By signing the file system root, CFS can guarantee that only the owner of the object can change the file system.

The second mechanism aims at preventing malicious peers from publishing large amount of polluted data in CFS. Specifically, CFS has a predefined expiration time for any CFS-block and its replicas stored in the system. CFS forces peers to continuously republish their data files in order to keep them persistent in the system. After the expiration time any non republished file is deleted. Note that, as mentioned below this mechanism also acts as a maintenance algorithm. Although CFS provides this automatic mechanism of garbage collection (i.e., a mechanism to remove all expired files in the system) by forcing peers to

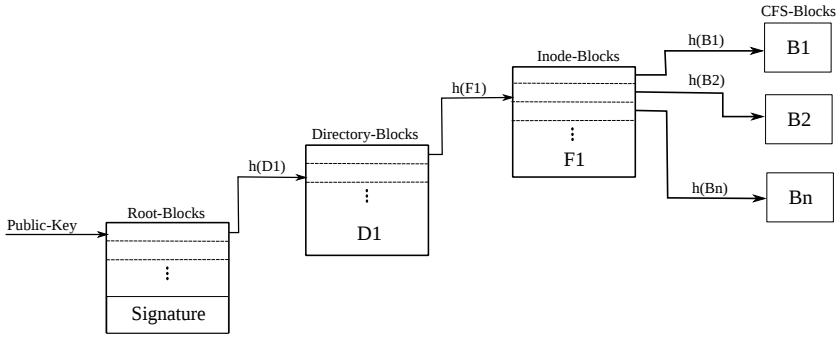


Figure 2.7: CFS mechanism used to ensure the integrity of CFS-blocks.

republish their data, it does not prevent the polluted data to be refreshed either. In order to reduce the amount of polluted data a malicious peer can introduce in the system, a third mechanism is proposed. This mechanism imposes that each peer should have a storage quota. A quota based mechanism limits the amount of data each peer can store in the system. This approach is effective to bound the amount of polluted data, however it penalizes both malicious and honest peers.

2.4.1.3 Data Maintenance

DHash layer follows the Chord successor's list approach to cope with churn, tolerating node's fault and repairing availability of lost blocks. Specifically, for each departing node C in charge of storing a set of blocks C_{blocks}, other $k-1$ replicas of C_{blocks} can be found in the $k-1$ nodes succeeding C. The first node D, succeeding C in the ring, becomes responsible for all data blocks previously stored in C. DHash layer in D is responsible for maintaining k replicas of C_{blocks}. Therefore, Dhash layer constantly repairs lost blocks to keep at least k replicas. Only if k nodes simultaneously leave the system data block will be unavailable.

The mechanism proposed to preserve the system reliability (by requesting the peers to constantly republish their data) can also be seen as a maintenance mechanism. The first objective of this mechanism is to reduce the impact of malicious peers in the system. However, at the same time it adjusts the number of replicas to the initial value k. Thus if any block is lost it should be recreated by the same mechanism, repairing the data availability.

2.4.1.4 Considerations

Some aspects that need to be taken into account when considering CFS as a storage architecture. First, data objects stored in CFS are not durable because the update-and-deletion mechanism of CFS requires peers that store data blocks in CFS to frequently update the blocks expiration time. Data blocks that do not have their expiration time up to date are deleted. This mechanism is used to avoid corrupted data to stay in the system for long periods. However, it punishes both malicious and good peers (i.e., well behaving peers) at the same time. Specifically, if a good peer p stores a file f in CFS and leaves the system for a long period, the blocks of f will eventually vanish after the expiration time and f will not be available. Second, high churn has a direct impact on CFS. Every time a peer join or leaves

CFS, the system needs to wait for the stabilization protocol of Chord to correct the successor and predecessor lists. Out-of-data successor lists can prevent requests for data from reaching its destination. Thus the performance of CFS lookups is susceptible to the ratio the peers join and leave as well as the frequency at which the stabilization protocol is executed. Finally, CFS does not present mechanisms to prevent (or limit) malicious peers from dropping messages they were supposed to forward or mis-routing messages to wrong destinations. Thus a collusion of malicious peers can compromise reliability of CFS. Later in Chapter 4 we will see that our approach handles all these problems.

2.4.2 Reperasure

Replication-with-erasure-codes (a.k.a. Reperasure [95]) is a storage manager that lies over a DHT in order to provide distributed storage. Reperasure was designed to benefit from basic operations of DHT overlays such as *lookup(key)*, *insert(key, value)* and *delete(key)*. However, it does not depend on any specific DHT implementation. Reperasure intends to be a generic model. Data files are assumed to have global unique identifiers called *doc-id*. Reperasure handles requests to store data to (and retrieve data from) any existing peer in the system.

2.4.2.1 Data Availability

Reperasure aims at demonstrating in practice that data availability can be provided in structured peer-to-peer systems efficiently only by using erasure codes as a redundancy scheme. Specifically, each data file is partitioned into chunks of file, called *units*. Units have predefined size of $\mathcal{S}_u = 4$kilobytes and also have their own identifiers called *unit-id*. Unit identifiers (unit-id) are basically the offset $\mathcal{S}_u$ of the *doc-id* (i.e., data file identifier). Instead of applying the redundancy scheme to the whole file, Reperasure applies the scheme to each unit (i.e., each chunk of data file). Specifically, units are fragmented into m blocks, called input blocks. Erasure codes are used to code these m input blocks into n coded blocks, as explained in Section 2.3.2.1. Reperasure uses Reed Solomon [70] algorithm to code input blocks. We explain in 3.3 the details of Reed Solomon Codes. The coded blocks are also called check blocks and they are the smallest fraction of data stored in Reperasure. Furthermore, for each data file check blocks are identified by integer numbers from 0 to $n-1$ and referred to as *block-ids*. However, the global unique identifier GUK of each check block is created from a secure hash function [86] applied to the concatenation of the *doc-id*, *unit-id* and *block-id*. The rationale of doing so is to provide a very simple mapping of file identifiers *doc-id* and check block identifiers GUK. Check blocks are uniformly spread in the form of pairs $< key, value >$ over the DHT structure it relies on, with *key* representing the GUK and *value* the check block itself. Note that, check blocks are stored in peers according to the metric function of the underlying DHT overlay. Afterwards, when data files are requested by clients, the Reperasure algorithm uses the $doc-id$ and global coding parameters to derive all GUK (check block identifiers) needed to recover the data file. Reperasure layer recovers check blocks from DHT by performing as many *lookup(GUK)* operations as needed. Reperasure layer decodes the data file from these check blocks and delivers the whole data files to the client.

2.4.2.2 Reliability and Data Maintenance

In Reperasure peers are assumed to be inside of a well-defined administration domain, where connectivity and components failures randomly happen but the system is free from malicious peers. Therefore, one of the distinguishing features of Reperasure it that it does not propose any mechanism to handle neither churn nor data integrity. Reperasure can support $n - m$ failing check blocks for each *unit* of a data file. Beyond this threshold, it is not possible to retrieve the initial *unit* by the properties of erasure codes. Thus Reperasure estimates the average availability of peers in the system in order to define the stretch factor to code data *units*. Although *GUK* of each check block can be used to check the integrity of check blocks, Reperasure does not implement any mechanism to ensure integrity of the original data as only honest peers are assumed to be in the system. However in real distributed and self-organizing environments it is difficult to have such environments free from malicious peers.

2.4.2.3 Considerations

Some aspects that need to be taken into account when considering Reperasure as a storage architecture. First, Reperasure relies on an abstraction of a distributed, highly-reliable and well monitored environment. Thus it is not applicable for areas where this scenario can not be assured. Second, Reperasure coding technique is based on traditional Reed-Solomon codes which is a fixed-rate coding technique. We will show in Chapter 3 that such a type of code is well adapted only for environments (i.e., communication channels) with a predictable loss rate. In Chapter 4 we will see that our approach tackles both problems.

2.4.3 Total Recall

Total Recall [15] is a peer-to-peer storage manager also built on top of Chord DHT overlay. Similarly to CFS in Total Recall peers organize themselves into a ring structure according to their identifiers. Peers and data resources have a unique global identifiers (UID) from the same m-bit ID space of the underlying Chord DHT overlay, as explained in Section 2.2.1. Data files are stored at the first peer in the ring whose UID succeeds the UID of the data file. This peer is the *successor* node of a key k in the Chord ring and it is called *master nodes* in Total Recall.

2.4.3.1 Data Availability

Besides storing data files, *master nodes* are peers responsible for providing data availability. Total Recall provides data availability by using either replication or erasure codes as redundancy schemes. The choice of which redundancy mechanism to use is based on the file size and file *use rate*. Use rate is the rate at which operations are performed on the file, or simply the workload required to serve the specific file. Peers responsible for storing redundant data are called *storage nodes*. Clearly, a peer can be a master node for a data file and at same time a storage node for other redundant data. By default, Total Recall uses replication for small files and erasures codes for large files in order to reduce the storage overhead. However, this

default strategy can be changed according to the dynamics of the system. For instance, popular small files can use erasure codes instead of replication in order to balance the workload needed to provide this file.

2.4.3.2 Reliability

Total Recall relies only on the consistent hash used by Chord to identify data resources in order to check the integrity of data files and redundant data. The m-bit unique identifier UID of each data file, as well as redundant data such as full replicas and redundant coded blocks, is obtained by hashing the content of the data. The UID can be used to verify any change made into data content by applying again the same hash function used to create the ID and comparing the resulting hash and UID. This is a very simple and efficient way of checking if data is reliable or not before using it. Total recall assumes that the system is free from malicious nodes, thus data once available is considered reliable.

2.4.3.3 Data Maintenance

The distinguishing characteristic of Total Recall is its ability of reducing the impact of departures by repairing the availability when necessary. Total Recall continuously measures and analyses the overall data availability in order to predict future changes and reacts in advance to these changes by modifying current strategies. The solution proposed by Total Recall is based on three key points: *availability prediction*, *redundancy management* and *dynamic repair*. Availability prediction consists in continuously monitoring all existing peers in the system. Based on the past behavior of peers, it builds short and long term predictions. Redundancy management consists in selecting the most efficient redundancy scheme based on the short term predictions. Dynamic repair consists in adapting repair policies used by the system to react to losses based on the long term predictions. The main difference among the two last mechanisms is that the former one attempts to reduce the impact caused by peers transiently unavailable which does not affect data durability. While the last one attempts to repair losses caused by non-transient peers that permanently leave the system and therefore threats data durability.

Availability prediction is implemented by a mechanism called *availability monitor*(AM) which is responsible for collecting the information about the availability of peers. In order to collect information about the churn in the system, AM proceeds as follows. At each master node MS_i, AM stores a meta-data information about the location of all redundant data corresponding to the data files placed at MS_i. The meta-data is used to track the availability of all storage hosts in charge of redundant data.

Based on the assumption that past churn rate can be used to create a stochastic model of the future peers availability, AM derives two metrics for each peer: short-term host availability and long-term decay rate. The short-term host availability is used to provide redundancy management. It measures the average of peers available at any given time during the past 24 hours. Based on the short-term measure Total Recall can modify the redundancy scheme (e.g. number of replicas) in order to achieve the predefined level of availability. Long-term decay is used to provide dynamic repair. It measures the rate at which peers leave the system over days and weeks. It is used to define the repair policy that must be used to preserve the predefined availability level. AM also acts as a trigger to repair redundancy affected by

system dynamics. There are two policies used in order to repair lost data: eager and lazy repair policies. With the eager repair mechanism, a new redundant data is immediately created as soon as lost data is detected. The system reacts immediately to losses. In lazy repair the system works with a predefined threshold. No matter how many data is lost, the system waits to reach a specific threshold before repairing any. Under typical workloads Total Recall applies eager repair policy for small files and lazy repair for large files. The rationale of using this strategy is that it minimizes the impact of repairing losses on bandwidth [75].

Furthermore, Total Recall defines the placement of redundant data (i.e., replicas or erasure coded blocks) based on the repair policy. For data files using eager repair, the master node stores m replicas at m storage nodes succeeding the master node in the Chord ring. For data files using lazy repaired policy, the master node stores the erasure coded blocks at s randomly selected storage nodes, with s the total number of coded blocks generated by each data file. Note that, s can be very large. Consequently, there exists a non negligible probability that more than one check blocks are stored in the same peer. In order to keep track of all these fragments, the master node stores pointers to all storage nodes in charge of lazy repaired coded blocks. Note that, different from CFS which has a static replication factor based on the successor's list size, in Total Recall the amount of redundancy applied varies according to previously specified target levels of availability and the prediction made by Total Recall's availability monitor.

2.4.3.4 Considerations

Some aspects that need to be taken into account when considering Total Recall as a storage architecture. The first observation is that the hybrid redundancy scheme used by Total Recall is "exclusive". Meaning, either replication or erasure codes is used to implement redundancy for each data file, both techniques are never applied concurrently. The type of redundancy is chosen based on the average availability of the node, size of files it is in charge of and the file workload (i.e., the average number of requests for the file). As a consequence, Total Recall presents a variation on bandwidth consumption depending mainly on the node availability and on the file workload. Another point is that, Total Recall is not designed to handle byzantine peers. Similar to Reperasure, Total Recall relies on an abstraction of highly-reliable and well monitored environment.

2.4.4 Oceanstore

Oceanstore [47] is a peer-to-peer storage architecture conceived to be a pay-per-use worldwide persistent storage developed at the University of California, Berkeley. Its service is designed to be provided by a well defined group of enterprises called *utility providers*. Utility providers cooperate with the system by supplying a pool of highly connected and geographically distributed servers to build the core of Oceanstore structure. Users give their data to Oceanstore, but they can also collaborate with the system by providing remote storage and access to other users, in exchange for economic rewards.

Oceanstore is a storage layer that lies on Tapestry overlay (see Section 2.2.2) that supports read/write operations. To support read/write operations, each update performed in a data object (i.e data file) is handled by creating a new version of that object which is linked the old version. The latest version and all the old versions form together a chain of versions.

Therefore, data objects in Oceanstore can be viewed as streams of read-only versions of data objects. Each object is identified by a $GUID$ (i.e., globally unique identifier) from the same ID-space of Tapestry. Specifically, the whole stream of objects has its $AGUID$ (i.e., active-GUID) and each object representing a different version of the object inside the stream has its $VGUID$ (i.e., version-GUID). Moreover, each data object is stored in blocks (i.e., chunks of data file), and each version of a data object consists in a set of blocks. Each block has its own identifier called $BGUID$. Later, we discuss how identifiers are generated.

A distinguishing feature of Tapestry is that objects are independent from their location. In Tapestry data objects can be stored at any peer in the system. However, each data object has its own object's *root*. Object's root is the peer whose $GUID$ is the closest identifier to the object's $GUID$. Then, every peer is the object's root of a set of data objects. Peers storing data objects must inform the object's root that they are storing them. Note that, the act of informing the object's root that a peer is in charge of a data object is called *publish* as defined in Section 2.2.2.

2.4.4.1 Data Availability

In Oceanstore, data objects are *nomadic data*. Nomadic data are data objects that can be stored anywhere depending on the overload and latency of the system. Data availability is provided in Oceanstore by intensively using replication as redundancy scheme and an unbounded number of replicas of each data object is created. Oceanstore's authors argue that this *promiscuous caching* is an important mechanism to provide data durability in case of network partitions, servers failure and malicious attacks. The location of replicas does not depend on the identifiers of servers they are stored in. For this reason, replicas of data objects are named *floating* replicas.

Two forms of data objects coexist in Oceanstore: *active* and *archival* objects. Active objects are the latest version of data objects, while archival objects are permanent read-only old versions of data objects. In the case of loosing all active replicas, old version can be recovered from archival objects. Updates can be performed concurrently in the active data objects, thus Oceanstore needs to synchronize the order in which updates are applied. Active replicas of a data object can be classified in two different types: *primary* or *secondary* replicas. Primary replicas are responsible for synchronizing and ordering the updates before applying these updates to the other active replicas. Primary replicas are a small set of replicas lying on a group of well defined high-bandwidth servers. These well defined servers are used to perform byzantine agreement protocols among all primary replicas in order to decide the update order. The rationale of using this small set is that, Oceanstore reduces the number of messages needed to synchronize updates. Secondary replicas do not participate in the synchronization process, but they receive synchronized updates dispatched by primary replicas. In order to reduce the time to propagate updates to all the other replicas, secondary replicas are logically organized into multi-cast trees, rooted at primary replicas. These trees can provide smaller delay for disseminating updates than point-to-point approaches.

In addition to the replication scheme used to create primary and secondary replicas, Oceanstore also uses erasure coding as a secondary redundancy scheme to generate archival objects. Archival objects are redundant objects obtained by erasure coding updated replicas and spreading these redundant coded blocks over the system. The archival process takes place as soon as replicas are updated by primary replicas. Oceanstore uses the fixed-rate

Reed-Solomon [70] codes to create archival objects. By using a stretch factor of two during the coding process, Oceanstore doubles the amount of storage consumed by each archival object [71].

Therefore, Oceanstore relies on both replication and erasure coding redundancy schemes in order to provide data availability.

2.4.4.2 Reliability and Data Maintenance

To ensure the integrity of read-only data objects, in Oceanstore $VGUID$ and $BGUID$ are generated from the secure hash of their respective data, $VGUID$ from hashing a specific version of a data object and $BGUID$ from hashing blocks that compose a data object version. This is a simple and efficient way of verifying data integrity. It is also useful to avoid naming conflicts, due to the low collision probability of secure hash functions. For read/write objects (i.e., the whole streams of data objects) Oceanstore uses the approach first proposed by Mazieres [63] to generate $AGUID$ (active globally unique identifiers). Each $AGUID$ is generated from the secure hash of the data object owner's public key and some *human-readable* name that identifies the data object. Then, any peer can verify object's owner through its public key and provide accounting against any malicious peers. Moreover, Oceanstore also provides access control to data by requiring any data update to be digitally signed. The set of primary replicas check if peers have the right to perform the write operation. This is achieved by checking access control lists (ACL) previously created by owners of data objects.

In Oceanstore, churn has a small impact on the data availability because it is absorbed by the large number of replicas created for each data object (*promiscuous cache*). Moreover, the number and location of each secondary replicas depends on the latency and server workload constraints. These constraints are defined beforehand according to the round-trip time to respond a client request and the communication traffic a server handle. Therefore, replicas are dynamically and continuously created in order to satisfy these constraints. Oceanstore does not necessarily repair lost data objects due to the massive replication the system produces. Data objects that have their consistency disrupted are simply ignored and another replicas is chosen. If all replicas of a data object are lost, there are still archival objects spread over the system to recover data objects. Oceanstore's authors argue that only a global disaster could destroy information in the system due to its intensive promiscuous cache.

2.4.4.3 Considerations

Some aspects that need to be taken into account when considering Oceanstore as a storage architecture. The first observation it that, in Oceanstore users pay to have global access to their files. Second, Oceanstore uses a promiscuous caching to provide data availability. The promiscuous caching is an approach that creates a very large number of replicas to avoid the data files to vanish. Thus in Oceanstore data availability is ensured in detriment of the storage overhead. Third, data integrity is guaranteed in Oceanstore by using a collection of well defined nodes. These nodes store the primary replicas of any file stored in Oceanstore and run byzantine-tolerant protocols to ensure that data is not corrupted before creating secondary replicas.

2.4.5 Conclusion

In this chapter we have discussed peer-to-peer overlays. We have seen that overlay networks are classified according to the way peers establish the connections among each other and how resources are organized and distributed among these peers. We presented distinguishing aspects of structured overlays and their advantages over unstructured ones. We also explained the properties that any storage system based on peer-to-peer overlay should meet. We presented how existing architectures handle these properties. Table 2.1 presents a summary of features of the presented peer-to-peer based storage architectures.

Table 2.1: Summary of P2P Storage Features

	CFS	**Reperasure**	**Total Recall**	**Oceanstore**
Underlying Overlay	Chord	Generic DHT	Chord	Tapestry
Redundancy Scheme	Replication	Fixed-rate erasure codes	Hybrid (either one or the other)	Hybrid (both)
Churn Handling	Chord' successor list	N/A	Chord' successor list	Promiscuous replication
Byzantine Peers	N/A	No malicious peers	No malicious peers	At primary replicas

We have shown in Section 2.3.2.1 that for a given level of availability, the redundancy scheme based on replication (e.g., CFS) has a significant storage overhead, if compared to the redundancy scheme based on erasure codes (e.g., Reperasure). Although both Total Recall and Oceanstore use a hybrid scheme, they implement their schemes in two different ways. Total recall implements a mutually exclusive approach where the system chooses which scheme to implement (i.e., either replication or erasure codes) based on the node availability, file size and the use-rate of the file. While Oceanstore relies on both schemes concurrently. Differently than all the other systems, Oceanstore is based on the superabundance of both replicas and coded check blocks. The idea behind Oceanstore is to guarantee the data availability by spreading as much as the data is used, which makes the storage overhead employed by Oceanstore extremely high. On the other hand, among the presented architectures, Oceanstore is the only architecture that presents a mechanism that deals with byzantine peers by using a small set of well defined and trusted peers to perform byzantine tolerant protocols before broadcast any information to other replicas. If these nodes become corrupted by malicious peers, replicas of data are no longer reliable. Total Recall and Reperasure simply assume the absence of byzantine peers in the system, which is a relatively naive assumption for real open distributed peer-to-peer systems. In the next chapter, we will focus on the properties of erasure codes. In addition, we will show that the choice of Reed-Solomon codes for implementing redundancy, as performed by Reperasure, Total Recall and Oceanstore, is not suitable for dynamic peer-to-peer systems where high churn is realistic expected.

Chapter 3

Redundancy with Erasure Codes

Contents

This chapter is devoted to erasure codes. As mentioned before, coding techniques are used to provide redundancy and increase data availability. We aim at explaining with details these coding techniques, complementing the brief introduction given in Section 2.3.2.1.

3.1 Definitions

In this section we describe some definitions used along the chapter according to the literature.

Hamming Distance: Given an alphabet $\mathcal{A}_q^n$ of elements, with each element composed by a sequence of n q-ary symbols. The Hamming Distance $d(x, y)$ between any two elements $x, y \in \mathcal{A}_q^n$ is defined as the number of symbols that differ from each other in

their sequences [40], and it is denoted by:

$$d(x,y) = |\{i : x_i \neq y_i, 1 \leq i \leq n\}| \tag{3.1}$$

Hamming distance is a metric, therefore, given the sequences $x, y, z \in \mathcal{A}_q^n$ the Hamming distance satisfies the following properties:

- Positivity: The distance d is a non-negative integer, i.e., $d(x,y) \geq 0$, matching the equality if and only if $x = y$.
- Symmetric equality: $d(x,y) = d(y,x)$.
- Triangle Inequality: $d(x,y) \leq d(x,z) + d(z,y)$.

Minimum Distance: Given an alphabet $\mathcal{A}_q^n$ and a code $\mathcal{C} \in \mathcal{A}_q^n$, we call the Minimum Distance of a code $\mathcal{C}$ the minimum Hamming distance d_{min} between any two distinct codewords in $\mathcal{C}$ [40], denoted by:

$$d_{min} = min\{d(x,y) : x, y \in \mathcal{C} \wedge x \neq y\} \tag{3.2}$$

Monic Polynomial: A polynomial $f(x)$ with degree m is called monic if its leading coefficient f_m (i.e., coefficient with the higher order) is equal to 1, as follows.

$$x^m + f_{m-1}x^{m-1} + ... + f_1x^1 + f_0x^0.$$

Irreducible Polynomial: A polynomial $f(x)$ is called irreducible if it is impossible to decompose (i.e. factorize) $f(x)$ into a product of two or more nontrivial factors. A trivial factor is any divisor of $f(x)$ different from 1 and itself. In other words, a polynomial $f(x)$ with degree m is irreducible if it is not the product of two other polynomials of degree lower than m. For instance, given the set of integer numbers, the polynomial $x^2 - 9$ is reducible because it can be decomposed into the product $(x-3)(x+3)$. The polynomial $x^2 + 1$ is irreducible, because it can only be decomposed into a product of trivial factors. An irreducible polynomial is also called prime polynomial.

Polynomial Root: The root of a polynomial $p(x)$ is the number i that makes the polynomial equal to 0 (i.e. $p(i) = 0$). A fundamental theorem of algebra states that a polynomial $p(x)$ of degree m has at most m roots.

3.2 Introduction to Erasure Codes

In this section we aim at explaining the principles of erasure codes. We also present some historical facts of the coding theory that have contributed to the evolution of this class of codes.

Coding Theory was originated between 1948 and 1950 with two seminal, concurrent and complementary works by Shannon [82] and Hamming [40]. Hamming was working on protecting information against corruption when storing it on devices. He developed the combinatorial aspects of the theory called error-correcting codes. Shannon wrote the probability and statistics to formalize the concept of information. He developed a model to study how information could be communicated among two elements (i.e. sender and receiver) through

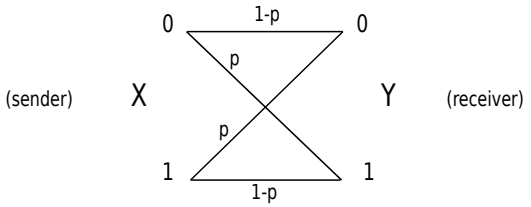
different communication channels. Shannon considered two types of channels: *Noiseless* and *Noisy* channels. In the noiseless channel, information does not change during the communication process (i.e. from sender to receiver). Consequently, it is possible to apply methods to reduce the amount of data to be communicated, representing the same original information. In the literature this process is referred to as *source coding* or *data compression*.

In noisy channels, there is a non null probability of disturbing the information during the communication process. Consequently, information can be altered in the communication channel. *Channel coding* is the process that enables reliable communication over unreliable communication channels. *Forward error-correction* (FEC) is the process of adding redundant data in order to make the communication process robust to any interference caused by noise. Error-correcting codes are the techniques used to introduce the required redundancy of forward error-correction (FEC). Both Hamming and Shannon concurrently proposed error-correcting codes to deal with errors in the communication channel. Both works of Hamming and Shannon are mutually complementary. While Hamming focus on the combinatorial aspects from an adversarial perspective, Shannon elaborates his theory in probabilistic models. The groundbreaking concepts presented by Hamming and Shannon have risen the entire field of information and communication theories [89].

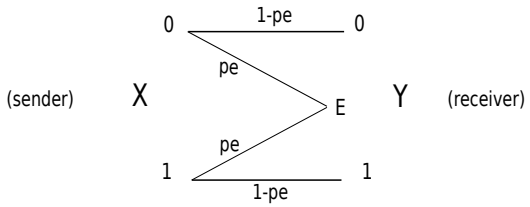
As a result of Shannon's theorem, the model of communication channel can be described as a channel with two possible input symbols $X = \{0, 1\}$, two possible output symbols $Y = \{0, 1\}$ and a probability p of error. This probability p represents the probability at which the information can be altered by noise in the channel, as mentioned above. This model is called *binary symmetric channel* (BSC) and it is shown at Figure 3.1(a). Suppose that X and Y are the random variables representing the possible input and output symbols, respectively. Given a specific input symbol, the conditional probability P of the output is denoted by:

$$P(Y|X) = \begin{cases} 1-p & \text{if } Y = X \\ p & \text{if } Y \neq X \end{cases}$$

In 1954, Peter Elias proposed a model for the communication channel different from the BSC channel. In this model, Elias considers that input symbols can not be corruptible. Unlike BSC model, once a symbol is received it is assumed to be correct. However, when the input symbols are sent through the channel there is a non null probability that these symbols get lost (i.e., erased). This model proposed by Elias is called *binary erasure channel* (BEC) [31] and it is represented in Figure 3.1(b).



(a) Noisy channel with error probability p, input symbols $X = \{0, 1\}$ and output symbols $Y = \{0, 1\}$.



(b) Erasure channel with erasure probability pe, input symbols $X = \{0, 1\}$ and output symbols $Y = \{0, 1, E\}$.

Figure 3.1: Noisy communication channels.

The model can be described as a channel with two possible input symbols $X = \{0, 1\}$

and three possible output symbols $Y = \{0, 1, E\}$, with E representing a symbol that is erased in the channel with probability pe. That is, the probability pe is the probability that the input symbol get lost. Then, suppose that X and Y are the random variables representing the possible input and output symbols and the erasure probability pe. Given a specific input symbol, the conditional probability P of the output is denoted by:

$$P(Y|X) = \begin{cases} 1 - pe & \text{if } Y = X \\ pe & \text{if } Y = E. \end{cases}$$

Erasure channels have received significant attention lately, since the Internet became largely used as a communication channel among applications. Erasure codes are the techniques used to cope with this probability of loosing information (i.e. erasures) in the communication channel. Erasure codes introduce redundancy in the information in order for the receiver to recover the original information in the presence of erasures. The difference between error-correcting codes and erasure codes lies on the type of protection they are providing. Erasure codes is a forward erasure correction designed to correct the erasures rather than errors. For instance, suppose that we have a network of computers (e.g. LAN, WAN, Internet) where the information is exchanged among computers through packet based communications protocols. It is a common approach to provide error protection for each packet in the physical layer of the network by using error-correcting codes. When sending a packet from computer A to computer B, at each hop the packets are checked at the link layer by applying error detection mechanism. If any error is detected, the packet is eliminated and it is not forwarded to the next hop avoiding the packet from reaching its destination (i.e. receiver). It is important to have an additional control in order to recover this "erased" packet. Erasure coding is the solution to protect these lost packets [91]. Let us present another simple and practical example. Suppose the sender A wants to transmit (or store) two information symbols (bits or packets) to a receiver B. The information symbols are respectively $x1 = 1$ and $x2 = 0$. When A sends $x1$ and $x2$ without code, it sends $x1 = 1$, $x2 = 0$. In this case, the receiver B must know what is the standard (i.e. protocol) being used, for instance $x1$ always comes first followed by $x2$. In order to deal with erasures in the communication channel, coding can be applied to the information. Let us assume a single parity symbol $x1 + x2$ is added to the information. Thus, now sender A has 3 pieces of information to send: $x1 = 1$, $x2 = 0$ and $x1 + x2 = 1$. If during the transmission some of the information get lost and B knows which one, then it is called erasure. Suppose that A sends $x1 = 1$, $x2 = 0$ and $x1 + x2 = 1$ and B receives only $x1 = 1$ and $x1 + x2 = 1$. The symbol $x2 = 0$ is lost. In order to correct the erasure, receiver B can solve the equation for $x1$ and $x2$ by using the parity information and then recovering the erased symbol $x2$. Now assume that all the information arrive at receiver B, but some of them are changed, for instance the second value is flipped: $x1 = 1$, $x2 = 1$ and $x1 + x2 = 1$. In this case, the value is changed and B does not know which one, then it is called an error. In this specific example, the error is detected using the information that arrived, however receiver B does not have enough information (parity symbols) to solve and correct the error.

As mentioned in Section 2.3.2, in erasure codes the information is divided into k blocks (i.e blocks of symbols or bits) called input blocks. These input blocks are encoded into s check blocks. We call the relation $r = \frac{k}{s} < 1$ the rate of encoding and this rate increases the storage cost by a factor of $\frac{1}{r}$. According to their rate of encoding erasure codes are classified into types: fixed-rate and rateless erasure codes. In fixed-rate codes, for every k blocks of

information, the coder generates s check blocks. Thus, the number of generated check blocks is bounded. Moreover, check blocks generated with fixed-rate codes are dependent of each other. Hence, after the coding process whenever a new check block is needed to be created, the whole coding process needs to be performed. It makes these codes well suitable for environments (i.e. communication channels) with a predictable loss rate. Fixed-rate erasure codes are also called traditional erasure codes in the literature. According to its degree of optimality fixed-rate erasure codes are also sub-classified into: a) Optimal erasure codes. b) Near-Optimal erasure codes. We call optimal erasure codes the class of codes for which any subset of only k blocks, among those s coded blocks, is enough to recover the original information. In this case, the code clearly supports up to $(s - k)$ lost blocks. Beyond this bound, the original information is not recoverable. Reed-Solomon Codes [70] is the best known example of optimal erasure codes. Near-Optimal erasure codes require at least $(1 + \epsilon)k$ coded blocks from s generated check blocks to recover the original message. Low-density parity-check [37] is an example of near-optimal erasure codes.

Fountain codes are not driven by a rate of encoding. Hence, a potentially infinite number of check blocks can be generated. Moreover, in fountain codes check blocks are independent from each other. As a consequence of that, new check blocks can be created whenever is needed. It makes fountain codes well suitable for communication channels with unbounded loss rate. According to its degree of optimality, fountain codes are classified as Near-Optimal fountain codes, requiring at least $(1 + \epsilon)k$ coded blocks to recover the original message. LT [57] and Online [61] are examples of erasure fountain codes. We discuss the principles of fountain codes later in Section 3.4.

Real communication channels such as Internet are clearly unreliable noisy channels as data transmitted over these channels are easily affected by errors and erasures. Implementing an erasure protection is of great importance to preserve information. Besides communication systems, erasure codes have been largely used in storage systems [3, 66, 28, 67, 15, 71, 25, 76, 92].

3.3 Reed-Solomon Codes

This section explains the fundamentals of Reed-Solomon coding.

In June of 1960, Irving Reed and Gus Solomon, then working at MIT Lincoln Laboratories, published a paper called "Polynomial Codes over Certain Finite Fields"[70]. This paper described a new class of codes that later became well known as Reed-Solomon codes. Since then, Reed Solomon codes have been used for error and erasure correction by many applications, from satellite and spacecraft communication to CD, DVD and arrays of disks RAID (Redundant Array of Independent Disks) [66].

Reed-Solomon code is a systematic linear block code. It is a block code because it splits the information to be transmitted (or stored) into k smaller blocks. Each block is a m-bit fixed-length symbol from a specific alphabet (i.e. a set of symbols). Encoded symbols are called codewords and they are linear combinations created from original symbols. It is a linear code because any linear combination of encoded symbols forms another encoded symbol. In other words, by adding any two codewords it produces a new one. In this section, we use the terms information symbols and encoded symbols (or codewords) to respectively designate input blocks and coded blocks. It is a systematic code because it does not change

the original information. Codewords are composed by original information symbols with redundant information attached to it, as shown in Figure 3.2.

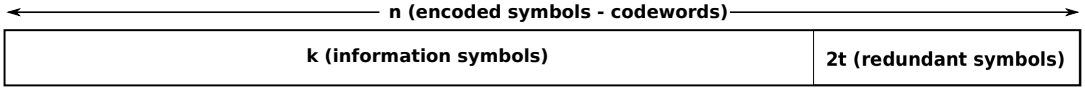


Figure 3.2: Representation of Reed-Solomon coding. Each symbol is a sequence of m-bits.

Reed-Solomon is also a non binary cyclic code. A cyclic code C is a code such that for every codeword belonging to C, a new codeword is obtained only by shifting the bit components of the codeword. Therefore, it forms a ring of codewords. Let s be the number of encoded symbols, k the number of information symbols to be coded (i.e. $1 \leq k \leq s$), m the number of bits of each symbol (i.e. $s \leq 2^m - 1$) and $2t$ the number of redundant symbols added to the original information (i.e. $2t = s - k$), then a Reed-Solomon code $RS(s,k)$ can be represented by:

$$\begin{aligned} RS(s,k) &= (s,k) \\ &= (2^m - 1, k) \\ &= (2^m - 1, s - 2t) \\ &= (2^m - 1, 2^m - 1 - 2t). \end{aligned}$$

Briefly, the information to be transmitted is divided into k information symbols. The $2t$ redundant symbols are generated from linear combinations of information symbols. An auxiliary generator matrix G of specific elements is used in order to create redundant symbols. Each original information symbol is multiplied by each element in G in order to create redundant symbols. A total of s encoded symbols (called codewords) are created and they consist of the original symbols appended with $2t$ redundant symbols. When codewords are received, Reed-Solomon codes can correct errors on few bits of symbols or recover erased symbols. Recall that, we are interested in the erasure correction ability of Reed Solomon codes. Specifically, Reed-Solomon's decoder solves a set of independent linear equations representing codewords. Thus, for an $RS(s,k)$ code any k from these s linear equations suffice to construct a system of k equations in k variables and recover the original information. Then, these $2t$ redundant symbols define both erasure and error correction abilities of Reed-Solomon codes.

The Hamming distance is of fundamental importance in order to correct errors and erasures, since it can identify the closest codewords to a corrupted codeword received and estimates which codeword was transmitted. In 1950, Hamming [40] has shown that the Hamming distance of a linear code satisfies the property of triangle inequality. In consequence of that, an (s,k) linear code with a minimum distance d is guaranteed to correct t errors and e erasures whenever $2t + e < d$. For instance, a code C with minimum distance 7 can correct up to 2 errors and 2 erasures. In 1964, Singleton [85] demonstrated that for any (s,k) linear code with minimum distance d_{min}, the condition $k + d_{min} \leq s + 1$ is always satisfied. It became later known as Singleton bound [59]. Codes matching the equality in Singleton bound are called Maximum Distance Separable (MDS) codes and they provide the

largest possible minimum distance among codewords. Reed-Solomon is a MDS linear code. Hence, the erasure correction ability of an $RS(s,k)$ code can be denoted by:

$$e = d_{min} - 1 = s - k \tag{3.4}$$

Up to $s-k$ encoded symbols can be erased and the original information can still be recovered.

3.3.1 Finite Fields

To proceed any further in the understanding of Reed-Solomon codes we recall some basic knowledge on finite fields. Reed-Solomon codes are based on arithmetics of finite fields. Finite Fields [55] is a part of the abstract algebra postulated by the French mathematician Evariste Galois. They are usually referred to as Galois Fields and denoted by $GF(q)$. A Galois Field $GF(q)$ is a nonempty finite set of q elements, with $q \geq 2$ which is called the order of the finite field. Addition and multiplication operations can be performed over elements of $GF(q)$ satisfying the following properties:

- Associative: for any $x, y, z \in GF(q)$, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$.
- Commutative: for every $x, y \in GF(q)$, $x \oplus y = y \oplus x$ and $x * y = y * x$.
- Distributive: for any $x, y, z \in GF(q)$, $(x \oplus y) * z = (x * z) \oplus (y * z)$.
- Identity: for any $x \in GF(q)$, there is an element a called additive identity such that $x \oplus a = a \oplus x = x$ and an element b called multiplicative identity such that $x * b = b * x = x$.
- Inverse: for any $x \in GF(q)$, there is an element x^{-1} such that $x * x^{-1} = 1$ and $x^{-1} * x = 1$.
- Closure: for any $x, y \in GF(q)$, the element $x * y \in GF(q)$ and $x \oplus y \in GF(q)$.

Galois has postulated that there exists a finite field with $q = p^m$ elements for any prime number p and a nonzero positive integer m, denoted by $GF(p^m)$ [55]. The finite field $GF(2^m)$ is used to build Reed-Solomon codes. The elements of a finite field $GF(2^m)$ are based on a primitive element called α. Each nonzero element of the field is represented by a power of α as follows.

$$GF(2^m) = \{0, \alpha^0, \alpha^1, \alpha^2, ..., \alpha^{2m-1}\} \tag{3.5}$$

Besides the α form, elements of the finite field $GF(2^m)$ are also represented by using a polynomial of degree $m-1$, such as:

$$GF(2^m) = \sum_{i=0}^{m-1} (a_i * x^i) = a_0 x^0 + a_1 x^1 + ... + a_{m-1} x^{m-1} \tag{3.6}$$

The coefficients $a_0 ... a_{m-1}$ take the values 0 or 1 according to corresponding binary coefficients of the element to be represented. For instance, the element with binary coefficients

(101) of a Galois Field with 8 elements, denoted by GF(8), with $m = 3$, is represented by the polynomial $x^2 + x^0$.

The closure property implies that operations performed over the elements of the finite field must result in another element of the field. To achieve that, the field arithmetic differ from the arithmetic used with integer numbers. In order to add elements we proceed by adding coefficient by coefficient of their corresponding polynomials, as shown:

$$(c_{m-1}x^{m-1} + ... + c_0x^0) = (a_{m-1}x^{m-1} + ... + a_0x^0) + (b_{m-1}x^{m-1} + ... + b_0x^0) \tag{3.7}$$

where, $c_i = a_i + b_i$, with $0 \leq i \leq m-1$, and

$$c_i = \begin{cases} 0 & \text{if } a_i = b_i \\ 1 & \text{otherwise.} \end{cases} \tag{3.8}$$

The result is a bitwise exclusive-OR function of the polynomial coefficients of each elements in their binary form [21]. For instance, in a finite field $GF(16)$ by adding the elements $x^3 + 1$ and $x^3 + x^2$ it produces $x^2 + 1$. The binary form of the elements are (1001) and (1100), respectively, and their corresponding decimal values are 9 and 12. Clearly, the result of the exclusive-OR between both elements is (0101), 5 in decimals and $x^2 + 1$ in its polynomial form. It shows that the resulting value, that would exceed 16 if integer arithmetic was used, is reduced to an element lower than $2^m - 1$, preserving the closure property for addition operations. Note that, subtraction operations of elements of a finite field is similar to addition because it also follows the Equation 3.8 to perform the subtraction of polynomial coefficients. Thus, addition and subtraction of two elements $a(x)$ and $b(x)$ of a finite field $GF(q)$ always produces the same result.

Multiplication of elements in the finite field also involves a particular process that requires the primitive element α. The primitive element (i.e. primitive polynomial) is a monic irreducible polynomial $f(x)$ from which all the other elements of the finite field are generated. In order to verify if the irreducible polynomial is a primitive polynomial we proceed as follows. Given a field $GF(2^m)$ and a polynomial $f(x)$ with degree m, if $n = 2^m - 1$ is the smallest integer for which $f(x)$ divides $x^n + 1$, then $f(x)$ is primitive polynomial. A primitive polynomial is represented by $p(x)$. Examples of primitive polynomials for different values of m are: for $m = 3$, $p(x) = 1 + x + x^3$, for $m = 4$, $p(x) = 1 + x + x^4$, for $m = 5$, $p(x) = 1 + x^2 + x^5$, and for $m = 6$, $p(x) = 1 + x + x^6$. The primitive polynomial $p(x)$ is used to multiply two elements $a(x)$ and $b(x)$ of the field $GF(2^m)$. In order to perform the field arithmetic multiplication, the first step is to multiply the polynomials $a(x)$ and $b(x)$ to produce an auxiliary polynomial $c(x)$ with higher degree and not a valid element of $GF(2^m)$. A modulo-primitive-polynomial mod-$p(x)$ division is performed on the auxiliary polynomial C. The reminder r of the mod-$p(x)$ division on $c(x)$ is the result of multiplication of elements on the finite field $GF(2^m)$. Taking r ensures the result to have a degree lower than m and therefore it is a valid element of the field $GF(2^m)$, satisfying the closure property.

Moreover, the primitive element is used to construct all nonzero elements of the finite field. Galois theory has shown that for any field $GF(q)$ with $q = p^m$ elements all nonzero elements of $GF(q)$ form a cyclic group such as $GF(q) = \{1, \alpha, \alpha^2, ..., , \alpha^q - 2\}$ under multiplication [51]. The primitive α is the polynomial root (i.e. $p(\alpha) = 0$). All nonzero elements of the field are generated by multiplying the root by itself, recursively. For instance, for a given a field $GF(16)$ with $m = 4$ and a primitive element $p(x) = x^4 + x + 1$,

all the others elements are generated as follows. If $p(x)$ is the primitive element, then $p(\alpha) = \alpha^4 + \alpha + 1 = 0$ and clearly $\alpha^4 = \alpha + 1$. Then, α^4 is defined as the sum of the elements α of low-order. Elements with order lower than α^4 are directly defined by the power of α, such that $\alpha^0 = 1, \alpha^1 = \alpha, \alpha^2 = \alpha^2$ and $\alpha^3 = \alpha^3$. From α^4 on, all the other elements are obtained recursively multiplying the current element by α and substituting $\alpha + 1$ at any possible simplification of α^4, as follows.

$$\alpha^5 = \alpha * \alpha^4 = \alpha * (\alpha + 1) = \alpha^2 + \alpha \tag{3.9a}$$

$$\alpha^6 = \alpha * \alpha^5 = \alpha * (\alpha^2 + \alpha) = \alpha^3 + \alpha^2 \tag{3.9b}$$

$$\alpha^7 = \alpha * \alpha^6 = \alpha * (\alpha^3 + \alpha^2) = \alpha^4 + \alpha^3 = \alpha^3 + \alpha + 1. \tag{3.9c}$$

This process produces the complete finite field $GF(16)$ for the primitive element $p(x) = x^4 + x + 1$. Table3.1 shows the representation of all elements on $GF(16)$.

Table 3.1: Elements of field $GF(16)$ with $p(x) = x^4 + x + 1$

Power Representation	Polynomial Representation
0	0
α^0	1
α^1	α
α^2	α^2
α^3	α^3
α^4	$\alpha + 1$
α^5	$\alpha^2 + \alpha$
α^6	$\alpha^3 + \alpha^2$
α^7	$\alpha^3 + \alpha + 1$
α^8	$\alpha^2 + 1$
α^9	$\alpha^3 + \alpha$
α^{10}	$\alpha^2 + \alpha + 1$
α^{11}	$\alpha^3 + \alpha^2 + \alpha$
α^{12}	$\alpha^3 + \alpha^2 + \alpha + 1$
α^{13}	$\alpha^3 + \alpha^2 + 1$
α^{14}	$\alpha^3 + \alpha + 1$
α^{15}	α

Note that, in Table3.1 the elements α^{15} and α^1 are the same. If we continue beyond α^{15}, (e.g. α^{16}, α^{17}, α^{18}...) we can see that the sequence repeats forming the cyclic ring with all valid values remaining in the field $GF(16)$. All these features are used in order to construct Reed-Solomon codes.

3.3.2 Reed-Solomon Coding and Decoding

We can now present Reed-Solomon codes. Coding using Reed-Solomon is performed as follows. For an $RS(s, k)$ code, all information symbols and redundant symbols are elements of m bits in a Galois Field with 2^m elements. Valid codewords are elements of $GF(2^m)$. To

create $s-k$ redundant symbols, that are appended in the k information symbols, a generator polynomial $g(x)$ is required. The generator polynomial $g(x)$ is composed of $s-k$ roots of $GF(2^m)$ and it is represented by:

$$g(x) = \prod_{i=0}^{s-k-1} (x - \alpha^i) \tag{3.10}$$

The generator polynomial must have $2t = s-k$ consecutive powers of α in order to correct t errors or $2t$ erasures [21]. Consecutive powers are chosen in order to maximize the distance property of the code. For instance, an $RS(15,11)$ uses the following powers: $(x-\alpha^0)(x-\alpha^1)(x-\alpha^2)(x-\alpha^3)$. The message (or information) to be encoded is divided into k information symbols and is represented by a polynomial $M(x)$ of degree $k-1$, as follows.

$$M(x) = M_0 * x^0 + M_1 * x^1 + ... + M_{k-1} * x^{k-1} \tag{3.11}$$

with $(M_0, M_1, M_2, ..., M_{k-1})$ valid elements in the $GF(2^m)$. To generate the redundant symbols, $M(x)$ is initially multiplied by x^{s-k} and the result is divided by $g(x)$. It produces a quotient $q(x)$ and a remainder $r(x)$, with degree lower or equal to $s-k-1$. The remainder $r(x)$ is the parity polynomial of $M(x)$ it is represented by:

$$r(x) = r_0 + r_1 * x^1 + ... + r_{s-k-1} * x^{s-k-1} \tag{3.12}$$

The polynomial $r(x)$ represents the redundant symbols created from the information symbols and the primitive elements α of $GF(2^m)$. Thus, after producing the redundant symbols in $r(x)$ the codeword should be formed by appending this information to the corresponding information symbols. Figure 3.2 shows the systematic representation of Reed-Solomon codes with redundant symbols appended to information symbols. The codeword C to be transmitted is created by combining $M(x)$ and $r(x)$ as follows:

$$C(x) = M(x) * x^{s-k} + r(x)) \tag{3.13}$$

with,

$$M(x) * x^{s-k} = M_0 * x^{s-k} + M_1 * x^{1+s-k} + M_2 * x^{2+s-k} + ... + M_{k-1} * x^{s-1}$$

Since, the encoded message $M(x)$ is divisible by the generator polynomial $g(x)$ with no remainder, by adding the reminder $r(x)$ to the message it is possible to check whether the polynomials arrive at the receiver with errors or not.

Recall that, with the erasure channel model the information that arrives at the receiver is always correct. However, it is not true for noisy channels where information can be changed. Although we are interested in the erasure correction ability of RS codes, we briefly describe the process of correcting errors. The first step in the decoding process for error correction is to verify if the received message produces remainders when divided by the generator polynomial. These remainders are called syndromes. Calculating syndromes produces a set of simultaneous equations that include terms representing errors. The second step, consists in evaluating these equations in order to find a so called location polynomial. This polynomial has roots which define the locations of errors. There are two algorithms used to resolve these equations, Berlekamp-Massey [11] and Euclid's Algorithms [53]. In the literature, the first

one is described as more efficient while the second one is described as easier to implement. In the third and last step, the location polynomial is used to find the error locations and error values. Chien's algorithm [20] is used to find the location of errors. This algorithm actually performs an exhaustive search, evaluating all possible symbols in the location polynomial until roots are found. Roots determine where errors are in the received symbols. Finally, Forney's algorithm [78] calculates the error values (or correct values for the received polynomial) to be substituted in the received polynomial. It finishes the error correction process. There exists a significant number of approaches to perform error correction with Reed-Solomon codes in the literature [38, 32, 46, 60].

Erasure correction is far easier than error correction, however we can find different approaches in the literature as well. We present here a generic method initially proposed by Reed and Solomon [70]. Let $M(x) = m_0 + m_1 * x^1 + m_2 * x^2 + ... + m_{k-1} * x^{k-1}$ be the polynomial representing the original message to be encoded, with $m \in GF(2^m)$ and $C(x) = m_0 + m_1 * x^1 + m_2 * x^2 + ... + m_{s-1} * x^{s-1}$ the polynomial representing the codeword, and $(\beta_0, \beta_1, ..., \beta_s - 1)$ be the evaluation of the primitive elements of $GF(2^m)$ in the polynomial $C(x)$ when $x = 0, 1, 2, ...s - 1$, such that $\beta_x = C(x)$. All β_s elements are transmitted from the sender to the receiver and during the transmission some values can be lost (i.e. erased). Since an RS code cas support up to $s - k$ losses, assume that k out of s values arrive at the receiver, denoted by $(\beta_0, \beta_1, ..., \beta_{k-1})$. Then, receiver can construct the following equations:

$$
\begin{aligned}
\beta_0 &= m_0 \\
\beta_1 &= m_0 + m_1 * \beta_1 + m_2 * {\beta_1}^2 + ... + m_{k-1} * {\beta_1}^{(k-1)} \\
\beta_2 &= m_0 + m_1 * \beta_2 + m_2 * {\beta_2}^2 + ... + m_{k-1} * {\beta_2}^{(k-1)} \\
&\vdots \\
\beta_{k-1} &= m_0 + m_1 * \beta_{k-1} + m_2 * {\beta_{k-1}}^2 + ... + m_{k-1} * {\beta_{k-1}}^{(k-1)}
\end{aligned}
$$

Thus, the receiver has k linear equations with k variables that must be solved in order to find the original m_i values. Moreover, from the algebra we know that if we have a set of k discrete points there is a unique polynomial of order (n-1) passing through these points. Hence, received values $(\beta_0, \beta_1, ..., \beta_{k-1})$ can be used find a unique polynomial going through these values. Once the polynomial is found the original information can be determined from it. This process is called Legrange's polynomial interpolation [12] method and it is equivalent to the method mention above. Solving Legrange interpolation leads to a problem of linear algebra, where the system of linear equations can be solved, for instance, with Vandermonde's matrices [42].

Standard implementations of Reed-Solomon codes for erasure correction require $\mathcal{O}(k)$ time per symbol and at least $\mathcal{O}(k^2)$ for decoding.

3.4 Fountain Codes

In this section we present the basic principles of a new class of erasure codes called Fountain or Rateless codes. The term fountain comes from the correlation to an everlasting fountain

of water, where you do not care what drops of water fill your glass when you are thirsty, you just need to go to the fountain, fill your glass and drink, as much as you need.

The characteristic that distinguishes rateless codes from traditional erasure codes is that a rateless code can produce for a given number k of input symbols (e.g. $b1, b2, ..., bk$) a potentially infinite stream of outputs (e.g. $CB_1, CB_2, CB_3, ...$). The absence of a fixed-rate coding gives the name rateless for the class of codes. In rateless codes, the output symbols must be generated randomly and independently from input symbols and any m output symbols should suffice to recover the original input blocks with some probability $\mathcal{P}_{(r)}$. The information about the input blocks that generated the output must be somehow communicated to the decoder (e.g. in the header of the output symbol). There exist three rateless erasure codes proposed up to now: Online Codes [61], LT Codes [57] and Raptor Codes [84]. Online and LT codes were proposed almost simultaneously in 2002, respectively by Maymounkov [61] and Luby [57]. Raptor codes appeared a little later in 2004 proposed by Shokrollahi [84]. The idea behind raptor codes is to code the input blocks of a message using fixed-rate codes before applying them to LT codes. In our work we will focus only on the two fundamentally rateless codes, LT and Online due to the benefits of rateless codes over rated-based ones (as presented in Section 3.4.1).

3.4.1 Highlights of Rateless Codes

The advantages of rateless erasure codes over traditional erasure codes are the following. In rateless codes, the ratio between input and encoded symbols is not fixed, which eliminates the need for estimating the loss-rate beforehand. Traditional erasure codes perform well only when the amount of loss can be previously estimated. Another point is that, in rateless codes the encoded symbols can be independently generated on-the-fly. Therefore, rateless codes properly work over unbounded loss channels and in highly dynamics systems, such as peer-to-peer ones. Moreover, in fixed-rate codes such as Reed Solomon, the finite size of the field limits the number of distinct encoding symbols that can be generated. In practice, a larger field introduces a significant overhead for the field arithmetic used on *RS* codes. Standard algorithms to decode Reed-Solomon codes require quadratic time, which can be too slow even for a small number of recovered symbols.

3.4.2 LT Codes

This class of erasure codes was created by Luby [57] and presents interesting coding and decoding complexities. For input massages composed of k input symbols, encoded symbols are generated on average by $\mathcal{O}(k \ln(k/\delta))$ symbol operations. While k original symbols can be decoded from any $k + \mathcal{O}(\sqrt{k} \ln^2(k/\delta))$ encoded symbols, with probability $1 - \delta$, with the same number of $\mathcal{O}(k \ln(k/\delta))$ symbol operations. We discuss these complexities later on Section 3.4.2.3.

3.4.2.1 Coding process

The data-item N (or message) to be coded is partitioned into $k = N/l$ same size symbols. The length l of symbols can be chosen as desired, there is no theoretical bound. For instance, for applications that are expected to transmit data over a network the size l can be chosen

based on the packet payload to be transmitted. After dividing the data into k blocks, each encoded symbol (called check block) CB_i is generated as follows:

- Choosing a *degree* (d_i) from a particular degree distribution, as explained in Section 3.4.2.3.
- Randomly choosing d_i distinct input symbols among k input symbols, we refer to these chosen input blocks as neighbors of CB_i.
- Combining the d_i neighbors into a check block CB_i by performing a bitwise XOR operation.

This procedure can be indefinitely repeated, a potentially infinite number of check blocks can be generated. Note that, the degree and the set of neighbors information d_i associated with each check block CB_i needs to be known during the decoding process. There are different ways of communicating this information during the coding process. In order to explicitly communicate this information, we represent any check block CB_i as a pair $\langle c_i, x_i \rangle$, with c_i the check block generated by combining d_i neighbors and x_i the set of indexes representing the d_i combined neighbors. Figure 3.3 shows the LT coding process represented by a Tanner graph [90], which is a bipartite graph where the first set of vertices represents input symbols k_1, k_2 and k_3 and the second set represents the generated check blocks CB_1, CB_2, CB_3, CB_4 and CB_5. Later we derive the lower bound K on the number of check blocks that need to be generated in order to guarantee the success of the LT process with high probability.

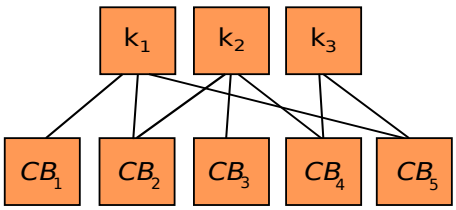


Figure 3.3: Check blocks $CB_1 = k_1, CB_2 = k_1 \oplus k_2, CB_3 = k_2, CB_4 = k_2 \oplus k_3$ and $CB_5 = k_1 \oplus k_3$ coded from input symbols k_1, k_2 and k_3.

3.4.2.2 Decoding process

Given $\mathcal{R}_{cb}$, a set of recovered check blocks, with each check block accompanied by its degree and set of neighbors information, such as

$$\mathcal{R}_{cb} = \{\langle c_1, x_1 \rangle, \langle c_2, x_2 \rangle, ..., \langle c_n, x_n \rangle\}, \tag{3.15}$$

the decoding process is performed on $\mathcal{R}_{cb}$ as follows.

The key idea is to rebuild the Tanner graph based on the set of received check blocks. Upon receipt of check blocks, the decoder performs the following iterative procedure:

- Find any check block CB_i with degree equal to one (i.e. each degree-one check block CB_i has only one input symbol k_i as neighbor).

- Copy the data c_i of $CB_i\langle c_i, x_i\rangle$ to k_i. The neighbor k_i of check block CB_i with degree-one is an exact copy of CB_i.
- Remove the edge between CB_i and k_i.
- Execute a bitwise XOR operation between k_i and any remaining check block CB_r that also has k_i as neighbor ($CB_r = CB_r \oplus k_i$).
- Remove the edge between CB_r and k_i.

This procedure is repeated until all k input symbols are successfully recovered. Given a set of received check blocks, we say that decoding is successful if the set of input symbols is fully decoded. Otherwise, it fails and more check blocks are required to complete the process.

3.4.2.3 Soliton Degree Distribution

The key point of LT codes is the design of a proper degree distribution. The distribution must guarantee the success of the LT process by using first, as few as possible check blocks to ensure minimum redundancy among them and second, an average degree as low as possible to reduce the average number of symbol operations to recover the original data. Moreover, in order for the decoding process to succeed the coding process must ensure that all the input blocks are covered (i.e. all input blocks must have at least one edge).

The first approach proposed by Luby to distribute degrees was to rely on an *Ideal Soliton Distribution* inspired by Soliton Waves [77]. The idea behind the Ideal Soliton distribution is that, at each iteration of the decoding algorithm the expected number of degree-one check blocks is equal to one. Specifically, if we denote by $\rho(d)$ the probability of an encoded symbol to be of degree d (i.e. $1 \leq d \leq k$), the Ideal Soliton distribution is defined as follows:

$$\rho(d) = \begin{cases} \frac{1}{k} & \text{if } d = 1 \\ \\ \frac{1}{d(d-1)} & \text{if } d = 2, \ldots, k \end{cases} \tag{3.16}$$

Unfortunately, the Ideal Soliton distribution performs poorly in practice since any small variation on the expected number of degree-one check blocks at each iteration leads the recover to fail. Then, Luby introduced the *Robust Soliton Distribution*, referred to as $\mu(d)$ in the following, to fix this weakness. By ensuring a larger expected number of degree-one check blocks the Robust Soliton distribution aims at guaranteeing the success of the LT process with probability at least $1 - \delta$, with δ arbitrarily small. The Robust Soliton distribution is based on three main parameters k, δ and C, where k is the number of input symbols to be coded, δ the failure probability of the LT process and C a positive constant that affects the probability of generating degree-one check blocks.

The Robust Soliton distribution is defined as follows, let $\tau(d)$ define

$$\tau(d) = \begin{cases} \frac{S}{k}\frac{1}{d} & \text{if } d = 1, \ldots, (\frac{k}{S}) - 1 \\ \\ \frac{S}{k}\ln(\frac{S}{\delta}) & \text{if } d = \frac{k}{S} \\ \\ 0 & \text{if } d > \frac{k}{S} \end{cases} \tag{3.17}$$

with S the expected number of degree-one check blocks in the decoding process given by

$$S = C \ln(\frac{k}{\delta})\sqrt{k}. \tag{3.18}$$

The Robust Soliton distribution $\mu(d)$ is the normalized value of the sum $\rho(d) + \tau(d)$, that is

$$\mu(d) = \frac{\rho(d) + \tau(d)}{\sum_{d=1}^{k} \rho(d) + \tau(d)}. \tag{3.19}$$

Luby [57] proved that by setting the estimated minimum number $CB0$ of check blocks to be

$$CB0 = k * \sum_{d=1}^{k} \rho(d) + \tau(d) = k + \mathcal{O}(\sqrt{k} \ln^2(k/\delta)) \tag{3.20}$$

the input symbols are recovered with probability at least $1 - \delta$.

Complexities of LT codes are determined by the number of edges in the generated graph. The assignment of degrees for the coding and decoding processes is closely related to the classical problem of throwing balls into bins. From classical probability theory we know that by throwing s balls uniformly and independently at random into k bins, one needs $s = k \ln(k/\delta)$ balls to ensure that all k bins have at least one ball with probability $1 - \delta$. Using the ball-into-bins analogy for distribution of degrees, bins are input blocks and balls are edges thrown from the generated check blocks to form the bipartite graph. By performing the same analysis, it is easy to verify that $\mathcal{O}(k \ln(k/\delta))$ edges are required to guarantee the coverage of all k input blocks, with probability $1 - \delta$. The total number of edges also corresponds to the total number of symbol operations performed during the coding as well as in the decoding process because it is the number of edges that ensure input blocks coverage. Thus, k input symbols are coded and decoded on average by performing $\mathcal{O}(k \ln(k/\delta))$ symbol operations. Moreover, if the required number s of check blocks to be recovered is close to k, then each check block is generated on average with $\mathcal{O}(\ln(k/\delta))$ symbol operations.

3.4.3 Online Codes

Created by Maymounkov [61], Online codes is another type of near optimal rateless erasure codes also based on sparse bipartite graphs. Online coding consists in three phases respectively called pre-coding, coding and decoding phases. The message (or information) to be coded is divided into k input blocks. During a pre-coding phase a small number of auxiliary blocks are created, these blocks provide an interesting property that with a fraction $1 - \epsilon/2$ of it, suffice to recover the original message. These auxiliary blocks are appended to the original message to generate a composite message. From the composite message, a potentially infinite number of output symbols called check blocks are generated. The decoding process is the inverse process, where the composite message is first decoded from check blocks and then the original input blocks are recovered from the composite message.

Online codes [61] are based on two main parameters ε, and q. Parameter ε infers how many blocks are needed to recover the original message (i.e., a message of k input blocks can be decoded with high probability from $(1 + \varepsilon)k$ coded blocks) while q affects the probability of reconstructing the original message (i.e., the decoding process may fail with negligible probability $(\varepsilon/2)^{q+1}$) [61].

3.4.3.1 Pre-Coding Phase (or Outer Encoding):

Consider an original message M (or data object) divided into k equal-sized input blocks. The pre-coding phase consists in generating a small number $A = \delta\varepsilon qk$ of auxiliary blocks and appending them to the original k input blocks. Specifically, for each original input block we associate q randomly chosen numbers $i_1, \ldots, i_q$ with $i_j \in [1, \ldots, A]$ such that each auxiliary block a_{i_j} is computed by XORing the content of all the input blocks we have associated to it. This is the same basic idea performed to encode check blocks in LT codes, as presented at Figure 3.3. The number of edges linking auxiliary blocks to input blocks is called degree of the block . The number A of auxiliary blocks are then appended to the original k input blocks to form the so called composite message F', which is used in the next coding phase. Briefly, we append the auxiliary blocks to the input blocks to form the composite message F'. Thus, the composite message of size $k' = k(1 + \delta\varepsilon q)$ is composed of k composite blocks with degree 1 (i.e., copies of input blocks with edges to the original input blocks) and A auxiliary block with a randomly chosen degree. Note that, similar to LT codes, a bipartite graph is created between input blocks and composite blocks.

3.4.3.2 Coding Phase (or Inner Encoding):

The coding phase consists in generating check blocks from the composite message F'. Specifically, each check block CB_i is generated by XORing the content of d_i blocks of the composite message, with d_i the degree of the check block sampled from a probability distribution ρ. The maximum degree F is a constant that depends on ε and is defined as:

$$F = \left\lceil \frac{\ln(\frac{\varepsilon^2}{4})}{\ln(1 - \frac{\varepsilon}{2})} \right\rceil \tag{3.21}$$

The biased distribution $\rho = (\rho_1, \rho_2, \rho_3, ..., \rho_F)$ is defined such that a degree i is chosen with probability ρ_i, defined as follows.

$$\rho(i) = \begin{cases} 1 - \frac{1+\frac{1}{F}}{1+\varepsilon} & \text{if } i = 1 \\ \\ \frac{(1-\rho_1)F}{(F-1)i(i-1)} & \text{for } 2 \leq i \leq F \end{cases} \tag{3.22}$$

The check block is then the pair $\langle c_i, x_i \rangle$ with x_i the set of d_i blocks chosen from F' based on the distribution ρ to compute the check block CB_i. A possibly infinite number of independent check blocks can be generated this way. Any set of $CB0 = (1 + \varepsilon)k'$ checks blocks are sufficient to recover a fraction $1 - \varepsilon/2$ of the composite message which guarantees to recover the original message with probability $1 - (\varepsilon)^{q+1}$.

3.4.3.3 Decoding Phase

Decoding amounts in rebuilding the bipartite graph composed by all recovered blocks $\langle c_i, x_i \rangle$ and its adjacencies x_i. An *adjacent* block is a block in the set x_i XOR-ed to produce each check block. In the bipartite graph the decoding algorithm continuously looks for received check blocks with only one unknown adjacent block. It recovers the adjacent composite block by

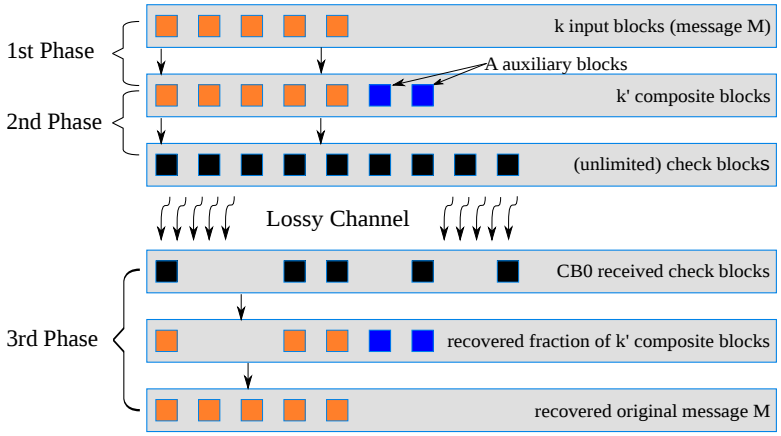


Figure 3.4: Overview of Online Codes. Pre-coding, coding and decoding phases.

XOR-ing the check blocks and all adjacencies. Hence, check blocks with adjacency-degree 1 are direct copies of the corresponding composite block. At each round, any recovered composite block increases the probability of recovering other blocks through its edges. At the same time input blocks are recovered from recovered composite blocks likewise. Figure 3.4 presents an overview of the 3 phases of Online codes.

Maymounkov [61] (appendix A, B and C) proved that Online codes perform the coding process of message of length k in a constant time ($\mathcal{O}(1)$) and the decoding of the same message in linear time ($\mathcal{O}(k)$).

3.5 Conclusion

In this chapter we have discussed the fundamentals of erasure codes. Erasure coding is a method used to provide efficient data redundancy with low storage overhead. Traditional rate-based erasure codes such as Reed-Solomon decoding presents quadratic time that makes them prohibitively expensive for large files. Decoding cost in terms of symbol operations is another advantages of rateless codes over traditional erasure codes. Another feature of traditional erasure codes is that, if created blocks are lost the complete coding process is required to be performed in order to recreate the lost blocks. Rateless erasure codes present some advantages over traditional ones. In rateless codes check blocks are coded independently, if peers storing some of the check blocks fail, new check blocks can be independently created without employing processing overhead of the whole coding process. Moreover, independently coded blocks can also be decoded independently. Thus, in rateless codes we do not need to restore a specific number of check blocks to start the decoding process, as we need in traditional erasure codes. We can take advantage of this feature to increase the efficiency of the decoding process, as we show later. All these features make this new class of erasure codes more suitable for dynamic peer-to-peer systems than traditional ones. In the next chapter we will present our proposed architecture for providing data persistence on distributed and dynamic peer-to-peer systems taking advantage of these features.

Part II

Contribution: Datacube

Chapter 4

Datacube: A Distributed Storage Architecture

Contents

4.1 Introduction

In this chapter we propose Datacube, a peer-to-peer structured architecture designed to provide persistent data storage and data integrity despite high churn, the presence of byzantine nodes and malicious targeted attacks.

Datacube provides persistence by implementing a hybrid redundancy scheme on top of Peercube [5], the structured overlay presented in Section 2.2.4. We will see in the course of this chapter that Datacube clusters are able to provide: *a*) Data durability with low storage overhead (i.e., only a constant number of full replicas is created at each cluster), *b*) a monitoring mechanism used to identify and isolate data stored at unreliable clusters, *c*) a recovery mechanism that is able to recover the whole set of data stored at any cluster, whenever a cluster becomes polluted due to malicious attacks. According to the investigated literature, this is the first work that addresses data-persistence on structured overlays in the presence of both high churn and collusion attacks.

In the following sections we will present the Datacube system and adversarial models. After that, we present an overview of the architecture and how Datacube provides data durability in both adversarial safe and adversarial prone situations. Next, we follow by describing the details of Datacube's architecture, presenting its operations and algorithms. Then, we present some analysis of the Datacube model. We finish this chapter by comparing Datacube with the architectures presented in Section 2.4.

4.1.1 System Model

Datacube is modeled as an overlay composed by N peers, each peer is identified by a global unique identifier of size m-bits. Identifiers are randomly chosen from a binary identifier space S. Each peer contributes with a partial amount of the total storage space available in the system. Hence, each peer is responsible for storing a subset of all data objects in Datacube. Data objects also have identifiers from the same identifier space S of peers identifiers. Data objects identifiers are obtained by applying a hash function on their data contents. The identifier of a peer is assigned when it joins the overlay, while identifiers of data objects are assigned when data objects are stored at Datacube. In the following, we use the terms identifiers and ID equivalently to name a peer identifier or a data object identifier. In Datacube the data object's ID and the data object itself represent respectively the key and value of the DHT-pair $(key, value)$. Data objects are placed at peers according to a proximity metric applied on their identifiers. Data objects exist in two forms in Datacube: *a)* data-items: data files (e.g. binary files, text files, images, etc.) and their replicas. *b)* check blocks: redundant chunks of data-items used in the hybrid scheme of redundancy of Datacube. Datacube derives other identifiers from the original data-item identifier to implement its redundancy scheme. We explain later how Datacube derives these identifiers.

4.1.2 Adversarial Model

We model the adversarial behavior in Datacube as follows. We assume that peers can join and leave the system at any time. Data availability is affected by leave operations. Regardless of the type of departure (whether temporary or permanent) there is a maintenance strategy that allows to recover the data stored at leaving nodes. The system is also composed of malicious peers. Such peers present unpredictable behavior. Malicious peers can manipulate data-items and check blocks they are in charge of by violating their integrity. Malicious peers can prevent data to be retrieved by dropping lookup messages, by refusing to forward lookup and replay messages. Malicious peers can also delete the data objects they are in charge of. These peers are called byzantine peers (or byzantine nodes) in the

distributed systems community. They act individually or by forming collusions in order to design strategies to attack the system. For instance, a collusion of malicious peers can perform consecutive join and leave operations in order to map the system topology. The goal of forming a collusion is to reach a majority quorum among core members in order to control the cluster operations. If the number of malicious peers in the core is greater than the limit established by byzantine tolerant protocols, the set of data objects stored in this cluster is considered unreliable. We assume that there exists a fraction of byzantine peers that always populate the system.

4.1.3 Architecture Overview

In this section we present an overview of how Datacube implements its hybrid redundancy scheme on top of the clustered DHT overlay.

4.1.3.1 Data Persistence in Datacube

Providing data persistence in distributed peer-to-peer systems requires the implementation of redundancy. In order to implement redundancy in a clustered DHT overlay, one could simply take advantage of the clusters and replicate data-items at all cluster members. However, such a straightforward solution could result in high bandwidth consumption, due to churn. Indeed, whenever a new peer would join the cluster a new replica of the data-item would have to be created and then the replica would have to be sent to the new peer. This approach would satisfactorily work in systems where the rate of peers joining and leaving the system is low. However, in systems with high churn this constant replication of data results in high bandwidth overhead. Figure 4.1 shows an example of this approach. In this Figure, the black arrows represent the join and leave events in the cluster. Rectangle boxes represent the replicas of a single data-item, with white rectangle boxes representing the existing replicas at a time t_0 and black rectangle boxes representing new replicas created due to the arrival of new peers at any time $t_i > t_0$. The tiny dashed arrows represent the bandwidth consumption needed to transfer a replica to any new peer in the cluster. This scheme clearly requires a significant computation cost and bandwidth consumption due to the constant creation of new replicas when exposed to high churn. Therefore, this approach should be avoided.

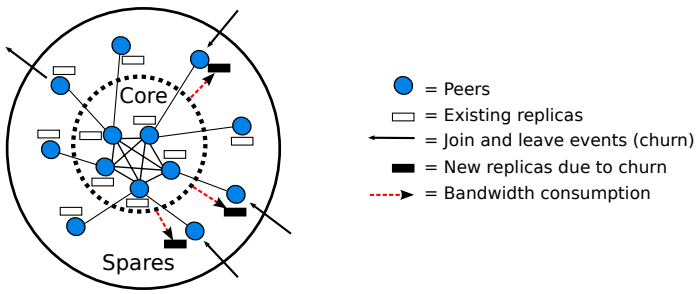


Figure 4.1: An example of a simple scheme for data redundancy on clustered overlays.

A better approach would consist in taking advantage not only of the clustering of the DHT overlay, but also to take advantage of the different roles taken by peers inside of the cluster (i.e. core and spare peers). By separating concerns of spare and core members, spare members limit the impact of join and leave events experienced by the cluster, while core members perform the fundamental operations of the overlay. Therefore, instead of replicating data-items at all cluster members, each data-item is replicated only at core members. The bandwidth overhead incurred in the previous solution is avoided because new replicas would be created and replicated only when core members leave. Since core members perform byzantine tolerant consensus for any operation executed in the cluster, as long as the bounded number $\lfloor \frac{S_{min}-1}{3} \rfloor$ of malicious peers required to perform byzantine protocols is preserved (with S_{min} the number of core members), it is possible to guarantee that data-items will be durably stored and malicious peers would not interfere in the output of the byzantine tolerant consensus performed by core members. In Figure 4.2 we see the arrows representing join and leave events and the rectangle boxes representing replicas stored only at core members. The dashed red arrow represents the bandwidth consumption whenever a new replica is created. We can see that the bandwidth consumption is not affected by churn as much as in the previous approach, join and leave events involving spare members do not trigger the generation of new replicas. New replicas will be created only if a core member leaves the cluster. In this case a new peer will join the core and a new replica is created and transferred to the new core member, as represented by the dashed red arrow and the black rectangle box in the Figure 4.2. Another positive point of this approach is that, the storage overhead is reduced (when compared to the previous approach) as replicas are stored only at core members.

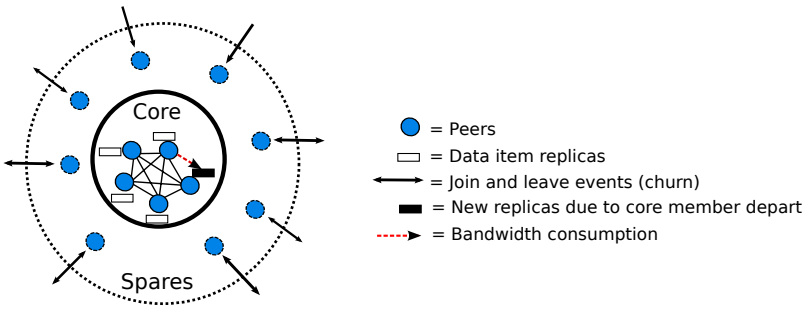


Figure 4.2: An enhanced scheme for data redundancy on clustered overlays.

However, keeping the number of malicious peers under the bound $\lfloor \frac{S_{min}-1}{3} \rfloor$ at each core is very difficult to be achieved in practice. Malicious peers can employ complex strategies to attack the system, thus there will be some regions of the system that may be populated by more malicious peers than expected. Join-leave attacks are an example of how malicious peers can use the determinism of DHTs mapping functions in order to map specific regions of the overlay to try to populate specific corners of the system [7]. In this case, if the fraction of malicious peers exceeds the allowed bound, the correctness of the byzantine tolerant protocols can not be ensured anymore. Consequently, the access or integrity of the stored data-items become potentially threatened. We call polluted cluster (or corrupted cluster) the cluster for which the fraction of malicious peers in the core is exceeded.

One of the main contributions of Datacube model comes from the fact that in Datacube the bounded number of malicious peers does not need to be simultaneously preserved at all clusters in the system. Instead, for any given cluster neighborhood if the number of corrupted clusters is bounded Datacube guarantees the access and integrity of any data-item stored at the clusters in this neighborhood. Datacube redundancy scheme is a compound of full replication of data-items in the core members and a rateless-based erasure coding that creates an extra redundancy (also called cluster-level redundancy) with a small storage overhead. This cluster-level redundancy is used to recover the data stored at the cluster whenever it becomes unreliable (i.e., a polluted cluster).

4.2 Infrastructure in Details

In this section we discuss the details of Datacube architecture.

Datacube hybrid scheme consists of two main operations: The first is the already mentioned replication of data-items into all S_{min} core members. Specifically, in Datacube each data-item D is placed at a cluster $\mathcal{C}_d$ whose identifier is the closest identifier to D. The second operation consists in partitioning the data-item D into same-sized blocks and coding these blocks into check blocks. These check blocks are specifically identified and placed at randomly chosen clusters. In the following, we use a top-down approach, as shown in Figure 4.3, to explain the details of Datacube architecture.

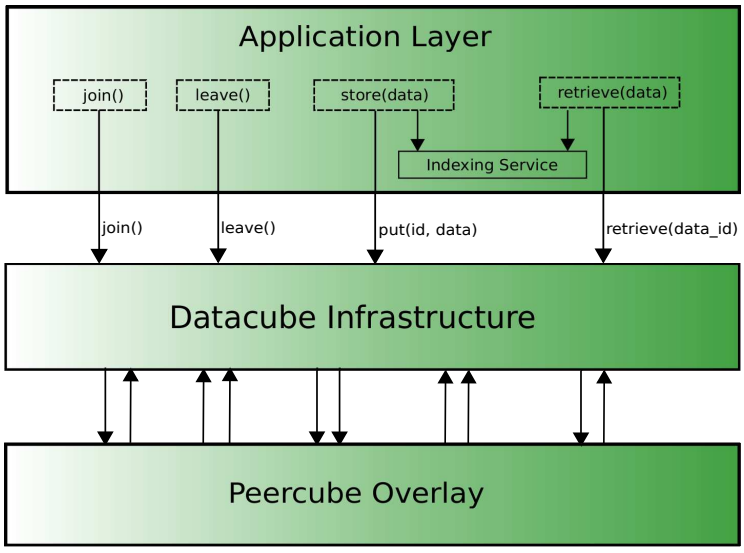


Figure 4.3: Datacube Infrastructure.

4.2.1 Application Layer

From the application point of view four main methods are performed in Datacube: `join()`, `leave()`, `store(data)` and `retrieve(data)`. Figure 4.3 shows how the application interacts with Datacube by invoking these methods. In the following we describe the function of

each method. First, the methods `join()` and `leave()` are the methods performed to connect to (and respectively, disconnect from) Datacube storage architecture. Second, the method `store(data)` is used to store data files in the system. Datacube requires the implementation of the `store(data)` method at the application layer. First of all, the method must generate the data-item identifier of the data-item and map this identifier to the respective data file name The data-item identifier corresponds to the hash of the data-item's content and the mapping is a simple indexing service that locally stores the identifiers of each stored file. In Figure 4.3 the mapping is represented as an indexing service embedded in the application layer. The mapping is used afterwards for both storing and retrieving data on Datacube. Next, the method `store(data)` triggers the execution of Datacube's redundancy scheme by using the data-item identifier generated at the application layer and the data-item itself. In order to trigger the redundancy scheme, `store(data)` sends a `put`(*key*(D), *D*) message to the core members of the cluster it sits in. In Section 4.2.2 we will discuss how the `put`(*key*(D), *D*) message triggers Datacube's redundancy scheme. Third, the method `retrieve(data)` is used by the application to access the data-items stored in Datacube. The method consists in searching the cluster in charge of the requested data and fetch the data back to the application layer. Before retrieving the data from Datacube the application must retrieve the corresponding data-item identifier from the indexing service. Thus, the application uses this data-item identifier to request the data-item on Datacube by sending the message `retrieve(data_id)` to its core members. In the Datacube layer, the method `retrieve(data_id)` can present two distinguished behaviors. The *modus operandi* of the method changes according to the potential influence of byzantine peers in the cluster that is in charge of storing the requested data. If the number of byzantine peers in the core of the destination cluster is under a specific bound on the number of malicious peers supported by byzantine protocols, the method `retrieve(data_id)` simply fetches the data back to the application layer. If the destination cluster exceeds this bound, the method `retrieve(data_id)` must trigger a recovery process, before sending the requested data back to the application. Note that, clusters do not know the number of byzantine peers that sit in their neighbour clusters. However, in Datacube each cluster periodically verifies the integrity of the data stored at neighbors by using a specific checking mechanism. Once a neighbor cluster is identified as corrupted it is tagged as well. These tags are used during the execution of `retrieve(data_id)` method to decide whether or not to trigger the recovery process. Later in Section 4.2.3 we will present and discuss how Datacube tags clusters as corrupted. Note that, any operation performed in Datacube layer is kept transparent to the application layer.

4.2.2 Datacube Layer

Now we explain the details of Datacube infrastructure and how it employs its redundancy scheme. We start by explaining the replication of data-items at core members. Following, we explain the extra redundancy applied to recover polluted clusters, also called cluster-level redundancy.

4.2.2.1 Replication at Core Members

As previously mentioned, core members of a given cluster are responsible for storing the set of data-items whose identifiers are closer to the label of this cluster than to any other cluster

in the system. Hence, any peer belonging to the core set stores a replica of each data-item the cluster is in charge of.

Now lets us explain that in details. First, we assume that Datacube's application is running at peer p_i which belongs to cluster $\mathcal{A}$. Specifically, in order to store a data-item D on Datacube the application invokes the method `store(D)`, which generates and maps the data-item identifier *key*(D) for D. After that, the invoked method `store(D)` triggers the redundancy scheme by sending the `put`(*key*(D), D) message to the core members of cluster $\mathcal{A}$. Any peer p belonging to the core set of $\mathcal{A}$ receiving the message of `put(key(D), D)` must deliver the data-item to the cluster in charge of storing it. Let us call $\mathcal{C}_d$ the closest cluster to the identifier *key*(D), then cluster $\mathcal{C}_d$ is the cluster in charge of data-item D. In order to deliver the data-item D to $\mathcal{C}_d$, any peer p receiving the message `put(key(D), D)` uses the underlying Peercube overlay to forward message to cluster $\mathcal{C}_d$. The `put`(*key*(D), D) message is forwarded from cluster to cluster by relying on Peercube overlay until it reaches the target destination $\mathcal{C}_d$. When the `put`(*key*(D), D) message arrives in the core of the destination cluster $\mathcal{C}_d$, any peer q in the core set of $\mathcal{C}_d$ receiving the message `put`(*key*(D), D) proceeds as follows. Peer q locally stores the data-item D and broadcasts the data-item to other core members in order to ensure that all core members will receive the data-item D to be stored, as shown at line lines 1 Figure 4.4. Note that, the method `storedata`($key(D)$, D) is not the same method `store(data)` invoked by the application layer.

Upon receipt `put`($key(D)$, D) **do**
1: broadcasts `storedata`($key(D)$, D) to all core members of cluster $\mathcal{C}_d$;
2: $inputBlockSet \leftarrow$ `uniformDataSplit`(D);
3: **if**($currentCoder$.`hasPrecode()`$= false$) **then**
4: $\langle inputBlockSet', key(D)' \rangle \leftarrow$ run consensus on ($inputBlockSet$, $key(D)$) among core members of $\mathcal{C}_d$;
5: invoke `codeBlock`($key(D)$, $inputBlockSet'$, $key(D)'$, 0, $CB0$);
6: **else**
7: $cMsg[\,] \leftarrow$ `generateComposite`($inputBlockSet$);
8: **foreach** (composite block $j \in cMsg$) **do**
9: $merkleLeafSet$[j] $\leftarrow$ `hash`($cMsg$[j]);
10: **enddo**;
11: $merkleRoot \leftarrow$ builds the Merkle tree on $merkleLeafSet$[j];
12: $\langle cMsg', merkleRoot' \rangle \leftarrow$ run consensus on ($cMsg$, $merkleRoot$) among core members of $\mathcal{C}_d$;
13: invoke `codeBlock`($key(D)$, $cMsg'$, $merkleRoot'$, 0, $CB0$);
14: **endif**;
enddo;

Figure 4.4: Algorithm run by each core member q of a cluster $\mathcal{C}_d$ in charge of storing data-item D.

Although it is a straightforward procedure, the replication at core members is sufficient to guarantee both persistence and integrity of data-items through Byzantine-tolerant protocols, as long as the number of malicious peers remains under a third of total number of core members at each cluster.

4.2.2.2 Leveraging Persistence at the Cluster Level

Now we explain the extra redundancy based on erasure coding. Differently from full replicas that provide persistence of data-items relying on peers in the core set, this additional scheme

extends the data persistence to the cluster-level. It allows Datacube to recover the entire set of data-items stored at clusters.

Datacube relies on the properties of rateless erasure codes to implement the cluster-level redundancy. As presented in Section 3.4.1, there are significant advantages of using rateless codes in peer-to-peer systems instead of traditional erasure codes. Datacube benefits from two significant features of a rateless codes: *a*) The absence of a fixed rate of coding. *b*) The independence among generated check blocks. The first feature guarantees that the number of check blocks to be generated does not need to be previously defined and is potentially unlimited. The second, ensures that check blocks do not depend on each other, thus whenever a check block is needed, it can be generated without imposing the re-computation of the whole set of check blocks. These features fit to dynamic peer-to-peer systems, such as Datacube.

Creating Redundant Blocks After broadcasting the data-item D at core members (at line 1 Figure 4.4), any peer q in the core of cluster $\mathcal{C}_d$ starts the coding process of D to spread check blocks in the system. The first step of the coding process is to divide data-item D into data fragments of same size called input blocks. Peer q, obtains the set of input blocks by invoking the method `uniformDataSplit(`D`)`, shown at line 2 of Figure 4.4. The next step is to check whether the type of coder being used to code check blocks is based or not on a pre-code phase. As mentioned in Section 3.4, the presence or not of a pre-coding phase is one distinguishing characteristics between the types of rateless codes. The pre-coding phase consists in generating a small number of auxiliary blocks to be appended to the input blocks. After creating the set of input blocks, each peer q in the core set of $\mathcal{C}_d$ checks whether the type of the current coder requires a pre-coding phase or not by invoking the method `hasPrecode()`, as shown at line 3 of Figure 4.4. If the type of coder does not require pre-coding, peer q uses the input blocks to generate the check blocks, as shown in lines 4 and 5 of Figure 4.4. If pre-coding is required, any peer q initially generates a set of composite blocks by invoking the method `generateComposite()`, as shown in line 7 of the same figure. Specifically, in the pre-coding phase for each original input block b_i we associate a randomly chosen number i of auxiliary blocks, with $i \in [1, ..., A]$. The content of each auxiliary block is computed by XOR-ing the content of all the input blocks associated to it. The auxiliary blocks are then appended to the input blocks, forming a so called composite message. The blocks of the composite message (a.k.a. composite blocks) are the blocks that will be coded to generate the coded check blocks of Datacube, as shown at line 13. Section 3.4.3.1 has more information about the pre-coding phase.

Handling Consistency Issues We have mentioned that in addition to data-item replication at core members, Datacube spreads check blocks outside of clusters. Datacube relies on these check blocks spread on remote clusters to recover corrupted clusters. On doing so, Datacube faces new challenges to ensure that these check blocks are reliable by the time a recovery process is invoked. Datacube must cope with potential weaknesses that prevent check blocks from either being durable and accessible.

Let us identify the cases where the integrity of check blocks are threatened in Datacube. First, during the coding process performed by core members, existing malicious peers can deliberately corrupt data used to generated check blocks. Second, after coding check blocks

can also be corrupted in the remote clusters they are stored. In the last case the consistency of check blocks is threatened in the following situations. First, check blocks sent to a remote clusters are corrupted by malicious peers in the clusters they are generated. Second, check blocks have their integrity violated by malicious peers located in remote clusters. Finally, check blocks vanishes due to leave operations in remote clusters. In any of these situations, check blocks stored at remote clusters may be useless due to manipulation by malicious nodes. Datacube must provide mechanisms to handle any threatening situation that can prevent the original data from being recovered.

Now lets explain how Datacube tackles these potential threats. The first mention threat assumes that existing malicious peers in the core of the cluster that is generating check blocks can deliberately corrupt the data used to create blocks before the coding process. The use of corrupted data during the coding process drives the decoding process to fail. Thus, Datacube employs a mechanism to ensure that only reliable data (i.e. non corrupted) is used during the coding process. This mechanism is implemented by requiring core members to agree on a unique set of data blocks (i.e., input blocks) before the coding process to start. From the fundamental consensus problem, we know that in a consensus agreement each process proposes its initial value and decides on a value v (e.g., $v \in 0, 1$ for binary consensus). If all correct processes propose the same value, at the end of the consensus the decided value is the value provided by a correct process [24]. Many byzantine consensus protocols have been proposed in the literature [4, 9, 10, 17, 22, 23]. Datacube relies on the same protocol used by Peercube which is proposed by Correia et al. [22] and tolerates up to $\frac{n-1}{3}$ byzantine processes, given a set of n processes. Thus, core members are required to perform a byzantine-tolerant consensus agreement in order to define a unique set of input blocks proposed by honest peers. Note that, depending on the type of coder used, the set of blocks to be proposed by each core member in the consensus can be either a set of input blocks or a set of composite blocks.

The second threat mentioned is that the check blocks can be manipulated by malicious peers located in remote clusters, where the check blocks are stored. Datacube employs another mechanism to verify the integrity of check blocks during the recovery process. Later we will see that at recovery time, this mechanism plays an important role helping Datacube to verify whether the recovered check blocks are corrupted or not. A straightforward approach to solve that would consist in safely storing a hash key obtained by hashing the content of each generated check block. However, it would require a significant storage overhead to hold hash keys of each check blocks.

A simple and efficient way of checking the integrity of recovered check block by storing only one hash key is to store the data-item identifier and compare the hash of the decoded data with this identifier at the end of the decoding process. If the hash of the decoded data and the stored data-item identifier match, the integrity of the recovered check blocks are preserved. Otherwise, other check blocks need to be recovered from remote clusters and the decoding process must be performed again. Despite its simplicity, the time spent on the process of recovering, decoding and checking the integrity of the check blocks can vary depending on the type of coder used during the coding process. Checking the integrity of check blocks takes potentially longer with two-phase coder (i.e., Online Codes) than with a single-phase coder (i.e., LT Codes). In order to make the time required to validate the integrity of check blocks more equivalent in both types of coders, Datacube uses another mechanism that checks the integrity of the recovered check blocks at the end of the first phase

of decoding process for any type of codes. When using pre-code based codes, Datacube relies on a Merkle tree to validate the whole set of composite blocks recovered in the first phase. If corrupted check blocks were used, the second phase of the decoding process is not performed and other check blocks must be recovered.

Merkle Tree [64] is a data structure scheme that contains a tree of hashes in which, each parent node is obtained from the hash of the concatenation of the hashes of their respective children. The element on the top of the hash tree is called root-hash, in Datacube we call that Merkle Root. Merkle Tree was created by Ralph Merkle in 1979. The original goal of a Merkle tree in cryptography was to provide digital signatures based on Lamport's one-time signature scheme [48]. However, it has been successfully used to verify the integrity of data sent through communication channels in peer-to-peer systems. A Merkle tree also allows Datacube to store only one hash key corresponding to the Merkle root instead of storing a whole set of hash keys corresponding to each check block. We discuss the remaining consistency issues in Section 4.2.2.2.

Now, let us return to the algorithm of Figure 4.4 to see how Datacube employs the mechanisms we have just mentioned. Depending on the type of the coder used, the required consensus agreement is performed in two different ways, as follows. First, if pre-coding phase is not required, the core members perform a consensus agreement on a set of input blocks and its corresponding identifier, as shown in line 4 of Figure 4.4. If the bounds on the number of malicious peers is preserved, the consensus agreement ensures that only non-corrupted input blocks are used in the coding process. After reaching an agreement, each peer q invokes the method `codeBlock()` in order to code the check blocks. We denote $CB0$ the minimum number of check blocks required to guarantee a successful recover of a given data-item in rateless erasure codes. Figure 4.4 shows in line 5, the invocation of method `codeBlock()` to generate $CB0$ check blocks, starting from the very first check block (i.e., argument initialCB=0). Second, if the coder requires a pre-coding phase, after generating the composite message, any peer q in the cluster $\mathcal{C}_d$ generates the leaves of a Merkle tree by hashing its each composite block, as shown from lines 8 to 10 of Figure 4.4. Having the leaves of the tree, each peer q builds the Merkle tree by hashing the children hashes until reach the Merkle root, shown in line 11 of Figure 4.4. The Merkle root and the composite blocks are then used by each peer q in the consensus agreement, line 12 of the same figure. Having reached an agreement on a composite message and a Merkle root (respectively represented by $cMSG'$ and $merkleRoot'$) core members of cluster $\mathcal{C}_d$ are ready to start coding check blocks. The coding process is performed by invoking the method `codeBlock()`, shown in line 13 of Figure 4.4.

Creating Datacube Check Blocks The `codeBlock()` method is a loop that creates a new check block at each iteration, as shown in line 1 of Figure 4.5. The invocation of the method `codeBlock()` requires the data-item identifier, the agreed set of blocks to code depending on the type of coder used (i.e. input blocks for LT code and composite blocks for Online codes), the agreed hash key used to check the integrity of recovered check blocks during the recovery process and the sequence of check blocks to be created (i.e., from the initial check block $initialCB$ to $totalCB$). When coding a new data-item, by default the number of check blocks that are initially generated is set to $CB0$. The number $CB0$ varies according to the type of the coder used and its respective set of coding parameters. See Section 3.4 for more information on coding parameters. Any peer q in the core of cluster $\mathcal{C}_d$ calculates the

Upon invocation `codeBlock(`$key(D)$, $agMsg$, $agKey$, $initialCB$, $totalCB$`)` **do**
1: **for** ($i = initialCB$ **to** $initialCB + totalCB$) **do**
2: $\langle c_i, x_i \rangle \leftarrow$ `generateCheckBlock(`$key(D)$, $agMsg$, $\mathcal{G}(key(D)+i)$, i`);`
3: $CB_i \leftarrow$ `hash`$^{(i)}$`(`$key(D)$`);`
4: p sends `put_block(`CB_i, $\langle c_i, x_i \rangle$, $key(D)$`)` to the closest cluster to CB_i;
5: **enddo**;
6: `register(`$key(D)$,$agKey$`)` at all neighbor clusters of $\mathcal{C}_d$, if not already done;
enddo;

Figure 4.5: Algorithm run by each core member q of a cluster $\mathcal{C}_d$ in order to code redundant data blocks from a data-item D.

minimum number $CB0$ of check blocks that it needs to generate, before invoking the method `codeBlock()`. Although Datacube generates initially C_0 check blocks, it is possible to create more to overcome unpredicted losses. For instance, suppose that a peer p_i belonging to a cluster $\mathcal{C}'$ stores check blocks $\{CB_2, CB_5\}$, which were generated from data-item D at cluster $\mathcal{C}''$. If p_i leaves the system, core members of $\mathcal{C}'$ need to request the cluster $\mathcal{C}''$ to code another two check blocks. When such a request arrives in cluster $\mathcal{C}''$, peers in the core set of $\mathcal{C}''$ use the method `codeBlock()` in order to code two new check blocks. Hence, Datacube can easily react to losses of check blocks. We will discuss more about this topic later.

Let us go a little further and explain how each check block is created. As presented in Section 3.4, we know that a check block is a combination of either composite blocks or input blocks, depending on the type of coder used. In order to generate a check block, the coder randomly chooses a number d of adjacencies (also called degree) for the check block. Then, it randomly chooses d composite blocks among those in the composite message to be combined and form the check block in Online codes and similarly chooses d adjacent input blocks for LT codes. The check block CB_i is represented as a pair $\langle c_i, x_i \rangle$, with c_i the check block generated by XOR-ing d_i neighbors and x_i the set of the d_i neighbors chosen to form c_i. Datacube adapts this procedure by using a pseudo random number generator (PRNG) function $\mathcal{G}(x)$ to generate check blocks. A PRNG is an algorithm that generates sequences of numbers with similar properties of random numbers. A PRNG $\mathcal{G}(x)$ must be initialized with a seed x, also called state seed. The main characteristic of a PRNG function is that, if a PRNG $\mathcal{G}(x)$ is initialized with the same seed x, it always produces the same sequence of numbers. Datacube benefits from this feature to synchronize the creation of check blocks among all core peers, with no message exchange. Datacube uses the function $\mathcal{G}(x)$ to select the degree (i.e. number of adjacencies) of each check block, instead of using the random approach. The degree of each check block is chosen by using the data-item identifier and the sequence number of the check block that is going to be created to form the seed of $\mathcal{G}(x)$. Specifically, suppose the coder is creating the j-th check block CB_j from data-item D. The degree d_j of the pair $\langle c_j, x_j \rangle$ that represents CB_j is obtained from

$$d_j = \mathcal{G}(key(D) + j). \tag{4.1}$$

Besides the synchronization among core members, the main motivation to use a PRGN to select the degree of check blocks is to create a mechanism called selective recovery used at recovery time. With selective recovery Datacube can improve the recovery process. We will discuss about selective recovery, later in Section 4.2.4. Line 2 of Figure 4.5 shows the

method `generateCheckBlock()` which is responsible for creating check blocks from composite blocks. Datacube follows the procedure described at 3.4.3.2 for Online codes and the procedure described at 3.4.2.1 for LT codes to generate each check block.

After creating each check block $\langle c_i, x_i \rangle$, an identifier must be assigned to each of them. The identifier defines the placement of the check block, the cluster it is going to be stored at. Identifiers must be random to prevent malicious peers from devising strategies to generate them. However, at recovery time identifiers retrieval should not require significant storage overhead. A straightforward solution would consist in storing the identifier of each check block in order to retrieve them for the recovery process. However, the storage overhead associated to this solution would be intolerable, since the number of check blocks that can be generated is potentially unlimited. Thus, Datacube employs a method based on a hash-chain [49] to identify check blocks. The hash-chain was first proposed by Lamport as a scheme to protect passwords. The benefit of using a hash-chain is that it can provide many keys from a single key. The chain is built by recursively applying a one-way hash function on a given key. Datacube exploits the hash-chain by using the data-item identifier to establish the chain of identifiers, with each iteration providing an identifier to a check block. Specifically, given a data-item D and its respective identifier $key(D)$ the identifier CB_n of a check block $\langle c_n, x_n \rangle$ is obtained from

$$CB_n = \texttt{hash}^{(n)}(key(D)) \tag{4.2}$$

as shown in the line 3 of Figure 4.5. Thus, at recovery time, identifiers of any check block can be retrieved from the identifier of the data-item they were originated from, with no extra storage cost.

Spreading Datacube's Check Blocks Having the check block $\langle c_i, x_i \rangle$ generated and identified by CB_i, any peer q in the core set of $\mathcal{C}_d$ is ready to send CB_i to a remote cluster. Let us call $\mathcal{C}''$ the remote cluster in charge of storing the check block CB_i. Each peer q in the core set of $\mathcal{C}_d$ sends the generated check block CB_i to the cluster $\mathcal{C}''$ by sending a `put_block()` message. As shown in line 4 of Figure 4.5, the message requires the identifier CB_i of the check block, the check block $\langle c_i, x_i \rangle$ itself and the identifier $key(D)$ of the data-item D from which CB_i was generated, as parameters. These parameters are used by the remote cluster $\mathcal{C}''$ for storing and maintaining CB_i available, as we show later. The message is forwarded by using the underlying Peercube overlay until it reaches the target destination $\mathcal{C}''$.

The last action taken by core members of cluster $\mathcal{C}_d$ is to make its neighbors able to recover the set of data-items stored at it, in the case $\mathcal{C}_d$ is detected as corrupted. Thanks to the hash-chain approach used by Datacube to identify check blocks, the only information the neighbors need to know to safely recover the set of data-items stored at $\mathcal{C}'$ is the data-item identifier $key(D)$ (in the case of Online codes, the agreed Merkle root is also stored). Note that, in the case of LT codes only the data-item identifier suffices to the neighbors to be able to recover any check block since the agreed key and the data-item identifier must be the same in this case. Hence, the last thing the `codeBlock()` does is to register these information at all neighbor clusters of the $\mathcal{C}'$, as shown at line 6 of Figure 4.5. We will see how recovery is performed later in Section 4.2.4.

Upon receipt `put_block(`CB_i, $\langle c_i, x_i \rangle$, $key(D)$`)` **do**
1: **if** (p.`spareView.length` $\geq \alpha$) **then**
2: $\alpha List[] \leftarrow$ `p.getClosestSpare(`α, CB_i`);`
3: $finger_{CB_i} \leftarrow$ `hash(`$\langle c_i, x_i \rangle$`);`
4: **foreach** (spare member $i \in \alpha List[]$) **do**
5: p sends `storeblock(`$CB_i, \langle c_i, x_i \rangle$`)` to αList[i];
6: $p.blockSpareMapping$`.add(`αList[i], $\langle CB_i, finger_{CB_i} \rangle$, $key(D)$`);`
7: **enddo**;
8: **else** p broadcasts `storeblock(`$CB_i, \langle c_i, x_i \rangle$`)` to p's core view;
enddo;

Figure 4.6: Algorithms run by any core member when receiving a check block to store at spare members.

Storing Redundant Blocks Once a peer p belonging to a cluster $\mathcal{C}''$ receives a message `put_block()` it performs the algorithm described in Figure 4.6 in order to store the check block at the current cluster. In Section 4.2.2.2 we have mentioned two other vulnerabilities of check blocks when stored at remote clusters. First, check blocks may have vanished due to the churn in the cluster it is stored at. Second, malicious peers inside the cluster can deliberately alter check blocks they are in charge of. Now we present how Datacube tackles these issues.

In order to reduce the influence of spare members departures and consequently to avoid the vanishing of the check blocks, Datacube requires check blocks to be stored at least at $\alpha \geq 2$ spare members. Hence, any peer p at the core set of $\mathcal{C}''$ that receives the message `put_block()` checks if there exists at least α spare members to hold the check block (line 1 of Figure 4.6). The α spare members can be chosen according to any arbitrary but deterministic function. Datacube chooses the α closest spare members to the identifier CB_i to store the check block at, as shown in line 2 of Figure 4.6. Moreover, keeping α replicas of each check block at α spares also avoids the computation cost of creating a new check block each time a spare member leaves. After finding the α spare members, peer p computes a fingerprint of $\langle c_i, x_i \rangle$ in order to ensure the integrity of each check block CB_i stored at spares. This fingerprint $finger_{CB_i}$ is obtained by applying a one-way hash function on the content of the check block, line 3 of Figure 4.6. Hence, core members can check the integrity of any check blocks stored at spare members by using this fingerprint.

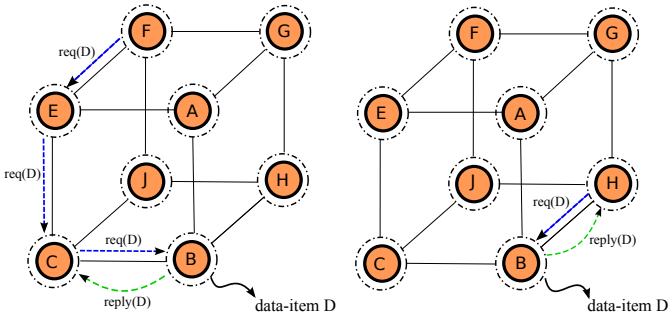
Finally, peer p stores the check block CB_i at α spare members, as shown in lines 4 and 5, by invoking the method `storeblock()`. Line 6 shows that core members also map CB_i to α spares (and vice versa) by using a data structure called block-to-spare mapping. Each entry in this map contains the respective fingerprint $finger_{CB_i}$ of the check block CB_i. This map is used by core members of cluster $\mathcal{C}''$ to keep the availability and integrity of at least α replicas. Whenever some replica q of CB_i among α leaves or becomes corrupted, a maintenance mechanism substitutes q for another spare member. In the worst case, when all α spare members simultaneously leave or collude, the core members of cluster $\mathcal{C}''$ are able to request the cluster $\mathcal{C}'$ that is in charge of storing data-item D to generate a new check block by using the block-to-spare mapping. This is only possible due to the independence property of coded blocks provided by rateless erasure codes. In traditional erasure codes, a whole set of new check blocks should be re-coded in this case. Actually, that is the reason the identifier

$key(D)$ is sent to remote clusters inside the `put_block()`message, to allow them to request new check blocks whenever needed. The new check block will not necessarily be placed in the same cluster, since the placement depends on the identifier of the new check block. Thus, after requesting a new check block cluster C'' can remove this entry from its mapping. Note that, when there are not enough spare members in the cluster C'' to store check blocks at, check blocks are temporarily stored at core members, as shown at line 8 of Figure 4.6.

4.2.3 Detecting Corrupted Clusters

The main goal of the detection mechanism is to identify misbehaving clusters (i.e., byzantine clusters) and tag them as *corrupted*. When a cluster is tagged as corrupted its neighbors have the ability of isolating it from the rest of the system. In Datacube, corruption detection is achieved through data integrity checks constantly performed by clusters in their neighborhood.

The integrity check can be performed in two ways, either passively or actively. The passive check is performed when a cluster benefits from data requests sent to any of its neighbors to compare the replied data with the requested identifier. Figure 4.7(a) shows an example of a passive check of integrity. In this example, a cluster F sends a request for a data-item D stored at cluster B. The request is passed from cluster to cluster until it reaches its destination. In the path to destination (i.e., $F \rightarrow E \rightarrow C \rightarrow B$), a cluster C that points to cluster B may verify the integrity of cluster B upon receipt a request for data-item D. When cluster C receives the reply of the request, it checks the integrity of D by hashing its content and comparing it with the identifier received in the request. The active check is performed periodically without any previous event. To perform this verification a cluster requests a data-item held by some of its neighbors clusters in order to verify the integrity of the replied data. Figure 4.7(b) shows an example of an active check, when a cluster H requests a data-item D stored at cluster B only to verify the integrity of the replied data. In both cases, the result of the check is based on the hash key of the content of the data-item.

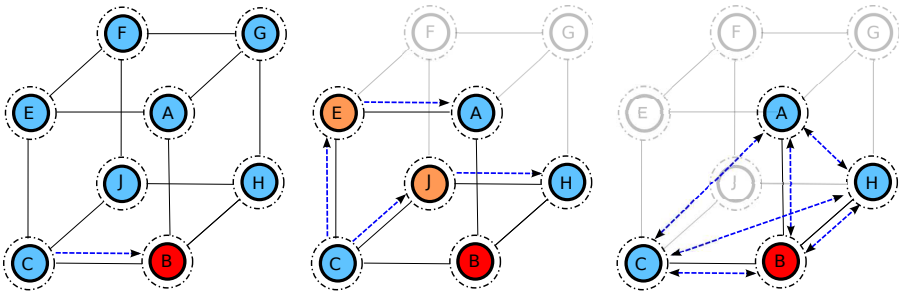


(a) Passive integrity check. (b) Active integrity check.

Figure 4.7: Two ways of detecting corrupted clusters in Datacube.

Now let us explain how clusters are tagged as corrupted. Suppose we have a cluster C that points to cluster B and C detects that cluster B is potentially corrupted, as shown in Figure 4.8(a). In order to avoid malicious clusters from deliberately tagging honest cluster as corrupted, Datacube requires that any cluster C detecting a neighbor cluster B as potentially corrupted to perform the following actions. Cluster C contacts the neighbors of B to perform

a binary Byzantine consensus agreement to decide whether B is corrupted or not. Datacube uses the underlying Peercube overlay to find what are the neighbors of cluster B as shown in Figure 4.8(b). When the neighbors of B are found, cluster C starts a consensus process in the neighborhood as shown in Figure 4.8(c).



(a) B detected corrupted. (b) Search for neighbors. (c) Agreement on B's state.

Figure 4.8: How Datacube tags corrupted clusters.

In order for the detection algorithm to work we need to specify a bound on the number of corrupted clusters that are supported during this phase. From the byzantine agreement protocols we know that f faulty processes can be supported among $n \geq 3f + 1$ processes during the consensus that defines whether a cluster is corrupted or not. If the number of clusters involved in the detection (i.e., target cluster and its neighbors) suffices to guarantee this property of byzantine agreement protocols, the number of corrupted clusters supported during the detection phase follows this bound specified by the byzantine agreement protocol, i.e., f corrupted clusters are supported when $n \geq 3 + 1$ clusters are involved in the consensus process. Thus, as long as this condition is preserved, if the result of agreement decides for a corrupted B, the corrupted cluster B is tagged as corrupted by all the neighbors. Moreover, if a corrupted cluster starts a tagging process to maliciously tag a honest neighbor as corrupted, the result of the agreement must refuse the tagging.

Note that, there is a specific situation where a single consensus execution may not be sufficient to detect a corrupt cluster among the neighbors in a given neighborhood. If a corrupted cluster behave assymetrically with its neighbors the integrity checks performed by honest neighbors may produce different outcomes. In this case, the whole detection process must be re-executed in order to avoid the assymetric behavior of corrupted clusters.

If the number of clusters involved in the detection is not enough to invoke a byzantine agreement among neighbors (i.e., neighborhood size less than 4 clusters) Datacube's clusters use another approach. In this case, any cluster detecting another cluster as corrupted relies on a quorum-based voting among neighbors to decide whether to tag a cluster as corrupted or not. We assume that there will exist only one corrupted clusters in the neighborhood in this situation. Thus, as long as this condition is preserved Datacube ensures that at least two nodes must agree in order to tag another cluster as corrupted.

Finally, after detecting a cluster as corrupted the recovery process is invoked as discussed in the next section.

4.2.4 Recovering Corrupted Clusters

The recovery process is invoked as a result of a positive corruption detection check. Once a cluster is detected as corrupted the first action taken by its neighbors is the isolation of it from the system. Second, the data stored in the corrupted cluster must be recovered. Although all the neighbors actively isolate a corrupted cluster from the system, the task of recovering the data-items stored at B is given to the the closest non-corrupted neighbor among the neighbors of the corrupted cluster.

Suppose that, B is the cluster detected corrupted as shown in the Figure 4.8(c), and Γ is the set of neighbors that detect B as corrupt (e.g., $\Gamma = \{A, C, H\}$), according to the mechanism described in Section 4.2.3. Initially, let us explain how corrupted clusters are isolated from the system. There exist two possibilities. First, let us assume that a cluster $C \in \Gamma$ receives a request r to a data-item D whose final destination is the cluster B, cluster C immediately drops the request r, preventing B from receiving data request r. Second, suppose that a cluster $C \in \Gamma$ receives the request r to a data-item D whose final destination is a cluster different from B, however cluster B is an intermediate cluster in the path to the final destination. In this case, C uses the independent paths of Peercube overlay to forward the request r to the destination by using an alternative path that does not cross cluster B. In this way, neighbor clusters prevent any request message from arriving at a corrupted cluster, isolating it from the system. Moreover, in the first case any cluster $C \in \Gamma$ that is the closest cluster to B (among its neighbors) invokes the recovery process of data-item D. Any other clusters $\mathcal{R} \in \Gamma$, with $\mathcal{R} \neq C$ drops any message to cluster B.

Now, let us explain step-by-step how cluster C recovers data-items stored at its neighbors. Suppose that cluster $C \in \Gamma$ is the closest cluster to cluster B and cluster B owns data-item D. When cluster C receives a request r to a data-item D, it immediately takes in charge the recovery of data-item D in order to reply to the request r with the recovered data-item D.

After isolating the corrupted cluster, the next action cluster C must take is to collect sufficiently many check blocks to ensure the decoding of the data-item D is successful with high probability. Core members of C calculate the minimum number $CB0$ of check blocks to be collected, based on the type of coder and its respective coding parameters. Using the hash-chain mechanism employed by Datacube to create the identifiers of the data-item D, cluster C derives the identifiers of all check blocks by recursively applying the hash function on the identifier $key(D)$ of the data-item. After retrieving all the identifiers, C requests the check blocks from the system by sending a lookup request for each check block. As soon as check blocks arrive in cluster C, it is able to start the decoding process, as we previously explained.

Handling Check Block Lookups Let us call G' the cluster that is holding a check block CB_g requested by cluster C'. Any cluster holding these check blocks are going to behave likewise. When the lookup request arrives at cluster G', the core members of G' first seek at their block-spare mapping to find out who are the α spare members holding a replica of CB_g. Core members of G' request to each α spare member its copy of CB_g. Before replying the check block CB_g to the cluster C', cluster G' checks of integrity of the check block. In order to check the integrity of a check block, core members of cluster G' use the the fingerprint of CB_g previously stored. The integrity of all α copies of CB_g is verified. Datacube allows different versions of the same check block to be stored at each α member. The rationale of doing this

is to avoid correct check blocks to be overwritten by corrupted ones. If there is no problem with the integrity of CB_g, the check block is sent back to C'.

When C' collects sufficiently many check blocks it starts the decoding process of the rateless erasure codes. In Sections 3.4.2.2 and 3.4.3.3 we detailed the process of decoding for both types of rateless erasure codes. In the case of Online codes, once composite blocks are decoded Datacube applies a integrity verification based on the agreed Merkle root registered at the neighbor cluster C' during the coding process. In the case of LT codes, once input blocks are decoded Datacube verifies the consistency of the recovered input blocks using the agreed hash key that corresponds to the data-item identifier, to decide whether other check blocks need to be recovered from the system or not. Hence, during the recovery process Datacube employs 2 different mechanisms on-the-fly to check the integrity of check blocks. If any integrity violation occurs, core members of cluster C' must collect other check blocks to recover the data-item D in order to guarantee the success of the decoding process. Once data-item D is fully decoded core members of C' send it back to the requesting node, and temporarily store data-item D.

When a new cluster N is created (due to a potential split, merge or create operation) and becomes in charge of the data-items of the isolated cluster B, the recovered data-items temporarily stored in cluster C' are transferred to N . Any Further requests for data-items previously owned by, but not recovered by C' are recovered by N, using the procedure explained above.

Enhancing Check Block Selection During the process of collecting check blocks, Datacube has the opportunity to apply different strategies to enhance the decoding process of the rateless codes. These strategies differ from each other according to the priority given to check blocks with specific degrees in detriment of others. To implement these different strategies, Datacube exploits its method of selecting the degree of check blocks by using a PRNG function and the hash-chain approach for the identification of check blocks. By using both functions together, Datacube can easily collect specific check blocks with p degrees. The strategies of collecting check blocks are classified as follows. *a*) *Random strategy*: In this strategy the same priority given to any degree. *b*) *Degree-one First-Random-Afterwards*: This strategy gives priority to all blocks with degree-one. If there is not enough blocks to complete the minimum number $CB0$, random blocks are chosen after those with degree-one. *c*) *Degree-one Only*: In this strategy, all possible check blocks with degree-one are selected. If there is not enough blocks to complete the minimum number $CB0$, check blocks with degree-2 that will be reduced to degree-one are selected. *d*) *Optimal Strategy*: In this strategy, the bipartite graph that links check blocks and input blocks in the case of LT codes and check blocks and composite blocks in the case of Online codes is re-created and only necessary check blocks are collected. The main difference between this strategy and the third one, is that in this case, duplicated blocks with the same degree are avoided. In chapter 5.4 we show the experimental evaluation of each strategy.

4.2.5 Putting-It-All-Together

We can summarize the modus-operandi of the Datacube redundancy scheme. Figure 4.9 presents a brief description of the scheme. Data-items are represented by rectangle boxes

inside the clusters. Note that, rectangle boxes with different colors represent different data-items. For the easiness of representation we assume that only one data-item is stored at cluster C.

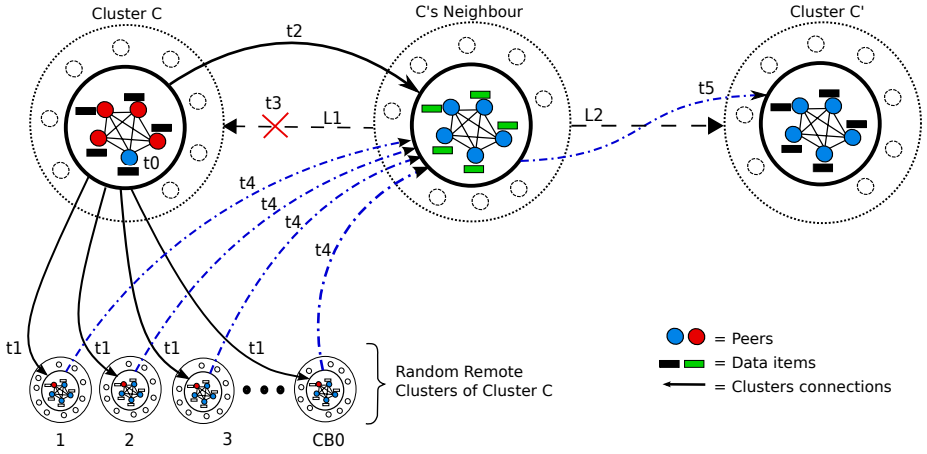


Figure 4.9: Summary of the Redundancy Scheme of Datacube.

Now, suppose that at time t_0, cluster C receives the data-item D to be stored. The data-item is immediately replicated at S_{min} core peers, with S_{min} the number of peers in the core. Following the replication, the data-item is partitioned into k same-sized blocks and coded into $CB0$ check blocks, with $k < CB0$ representing the minimum number of check blocks previously calculated by the coder to guarantee the recovery of the original data with high probability. Recall that, the number of check blocks generated is potentially infinite. Now let us assume that, at time t_1, these check blocks are specifically identified and placed at $CB0$ randomly chosen remote clusters. Next, at time t_2, neighbor clusters of C are informed that the data-item D was just created in the neighbor. At this point, the Datacube redundancy scheme has been set up and any request for the data-item D stored in cluster C are replied by its core members. From time to time, neighbor clusters of cluster C perform periodically integrity tests to verify the reliability of their neighbors. Let us assume that, at time t_3 cluster C is detected as polluted by its neighbors. Then, C's neighbors first isolate the polluted cluster C from the system by avoiding any request messages to be forwarded to it. Neighbors invoke the recovery of the data-item D that cluster C is in charge of by recovering the check blocks stored at remote clusters. At time t_4, the check blocks are collected from those remote clusters are decoded by C's neighbors to rebuild the data-item D which is replied to the requester. At time t_5, data-items that cluster C was in charge of, are eventually placed in a new cluster C' if there are sufficiently enough peers to form the new cluster C'. This briefly illustrates the redundancy design proposed in Datacube.

4.3 Datacube Analysis

This section is devoted to the analysis of the data availability guaranteed by Datacube and the associated storage and bandwidth usage requirements. Note that numerical values of the parameters used in the experiments are drawn from Peercube features [5], in particular

the derivation of the number of independent routes. Finally, we assume a severe adversarial environment with at least 30% of the population is malicious.

4.3.1 Data Availability

In the following, we analyze the availability of data-item D. We assume that D sits at cluster $\mathcal{C}$, and each of the CBi generated check blocks have been spread on clusters $\mathcal{C}_i$, where $CB0 \leq i \leq CB$, with CB the number of generated check blocks needed to reach a given data availability. For simplicity reasons, we assume that each check block sits on a different cluster. By construction, D availability depends on both cluster $\mathcal{C}$ availability where the S_{min} replicas of D sit and on the availability of the clusters $\mathcal{C}_i$ on which check blocks CBi are located. Let μ denote the ratio of malicious nodes in the whole system. The probability p_p that cluster $\mathcal{C}$ is polluted is equal to the probability that its core set is polluted, that is, populated by more than $\lfloor (S_{min}-1)/3 \rfloor$ malicious nodes. In the following we consider the upper bound of p_p which is obtained when the number of clusters in the system is minimal, i.e. equal to $\lceil N/S_{max} \rceil$. Let Y denote the random variable representing the number of malicious nodes in a given core set, and X the one depicting their number in the cluster then

$$p_p = 1 - \sum_{y=0}^{\lfloor (S_{min}-1)/3 \rfloor} \sum_{x=0}^{\lfloor (\mu.N) \rfloor} \Pr\{Y = y | X = x\} \Pr\{X = x\}.$$

Probability $\Pr\{Y = y | X = x\}$ represents the probability that y malicious nodes are inserted in the core knowing that x are in the whole cluster, i.e.,

$$\Pr\{Y = y | X = x\} = \binom{x}{y} \cdot \binom{S_{max} - x}{S_{min} - y} / \binom{S_{max}}{S_{min}},$$

and

$$\Pr\{X = x\} = \binom{\mu N}{x} \cdot (S_{max}/N)^x \cdot (1 - S_{max}/N)^{\mu N - x}.$$

The lower bound p_h on the probability that a request issued from cluster $\mathcal{C}$ successfully reaches cluster $\mathcal{C}_i$ located at h hops from $\mathcal{C}$ and is successfully handled by $\mathcal{C}_i$ is equal to

$$p_h = 1 - (1 - (1 - p_p)^h)^r,$$

where r is the number of independent paths the request can take. It has been proven in [5] that $\log(N/S_{max}) \leq r \leq \log(N/S_{max}) + 3$ and $1 \leq h \leq \log(N/S_{max}) + 5$. We are ready to derive the availability d_a of D.

$$d_a = (1 - p_p) + p_p \sum_{CB0}^{CB} \binom{CB}{CB0} \cdot (p_h)^{CB0} \cdot (1 - p_h)^{CB - CB0}$$

Table 4.1 provides a comparison between the stretch factor of our policy (i.e., the total amount redundancy added to data-items which is equal to CB over $CB0$) and the replication factor imposed by classical full replication required to get data availability at least greater than 0.99, and 0.999. To compute the replication factor obtained with classical full replication, we suppose that each copy of the data-item is replicated on a different cluster, as supposed

for check blocks. Experiments are conducted for different sizes N of the system (i.e., 1000, 5000 and 10000 peers). Let $S_{max} = \lceil \log_2 N \rceil$. We also assume different values of h. When h is maximized the probability of encountering faulty clusters also increases, because the bigger the number of hops is the higher the probability of traversing corrupted cluster will be. For example for $N = 1000$, we have maximal $h = 11$ hops. Finally, we assume that 30% of the nodes in the system are malicious (e.g. for $N = 1000$, $p_h = 0.076$). However, to simulate a targeted attack at cluster $\mathcal{C}$ (the cluster on which D sits), we suppose that $\mathcal{C}$ is populated by $\mu_{target} = 45\%$ of malicious nodes.

Results of these experiments, given in Table 4.1, show the benefit of our approach over full replication. It is shown that for reaching different levels of availability, the required stretch factor increases relatively smoothly, while the growing of the replication factor is more sensitive. For instance, for $N = 1,000$, $h = 11$ and $CB0 = 50$, the stretch factor is equal to $13,4$ for an availability of .99 and reaches a stretch factor of $13,5$ for an availability equal to 0.999% using a single path. Whereas in the same conditions, the replication factor increases from 44 to 65 which clearly becomes a problem for large data-items. The same trend is obtained for increasing values of N.

N	hops number	number of paths	Stretch factor		Replication factor	
			0.99*	0.999*	0.99*	0.999*
1.000	1	1	1.42	1.50	3	7
	5	1	3.66	3.96	11	16
	11	1	13.4	13.5	44	65
	1	9	1.02	1.02	1	1
	5	9	1.98	1.10	2	2
	11	9	1.98	2.12	5	8
5.000	1	1	1.42	1.50	3	5
	7	1	5.66	6.18	18	27
	14	1	25.00	27.5	83	125
	1	12	1.02	1.02	1	1
	7	12	1.12	1.16	2	3
	14	12	2.60	2.80	7	11
10.000	1	1	1.42	1.5	3	5
	8	1	7.02	7.68	22	34
	15	1	30.90	34.00	103	155
	1	13	1.02	1.02	1	1
	8	13	1.18	1.22	2	3
	15	13	2.88	3.12	8	12

Table 4.1: Comparison of the stretch factor in Datacube and the replication factor obtained in classical full replication as a function of the required availability and the number of nodes N in the system. In these experiments, the ratio of malicious nodes in $\mathcal{C}$ is equal to 45% while it is equal to 30% in the remaining of the system. Numbers followed by a * represent different values of data availability.

Figure 4.10 confirms the scalability of our approach. For all these experiments we have $CB0 = 50$, $\mu = 30$ and $\mu_{target} = 40\%$. For $N = 2^{16}$, rather than a replication factor of 194, we achieve the same availability with a 3-fold savings by relying on hybrid replication. Finally,

one can notice the impressive benefit when using independent routes on both approaches. For instance, for $r = 6$ the replication factor decreases to 33 while in our solution the stretch factor drops to 10.

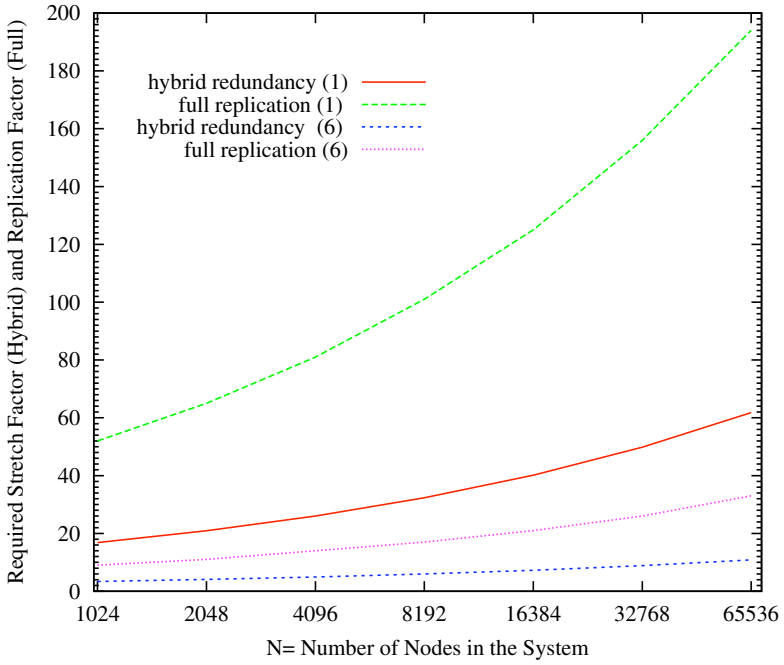


Figure 4.10: This graph shows the required stretch factor for our solution (hybrid redundancy) and replication factor for the full replication approach as a function of the number of nodes N in the system. The required availability is $d_a = 0.99$. The number shown in brackets (i.e., 1 and 6) represents the number of redundant routes.

4.3.2 Storage Overhead

We now compute the storage overhead implied by Datacube. We recall that each data-item D is initially replicated at S_{min} core members and K input blocks generated from D are coded and spread to other clusters at α spare members. The storage overhead DS for D is given by:

$$DS = \lceil (kS_{min} + CB\alpha) \rceil - k. \tag{4.3}$$

We now estimate how this storage overhead is distributed among nodes in Datacube. First, remark that data-items identifiers are randomly assigned for both data-items and check blocks. Thus, we can interpret the placement of both data-items and check blocks as the throwing of balls into several urns. We denote by Z_c (respectively Z_s) the random variable representing the total number of data-items (respectively check blocks) stored at each core (respectively spare) member. Let f be the total number of data-items in the system, CB be the number of check blocks generated for each data-item D to get a target data availability, and $N = N_c + N_s$ be the current number of nodes in Datacube, N_c (respectively N_s) is the number of core (respectively spare) members. The probability that the number of data-items

(respectively check blocks) at any core (respectively spare) is upper bounded by z is given by:

$$P(Z_{s,c} \leq z) = \sum_{i=0}^{z} \binom{fDS_{s,c}}{i} \cdot \left(\frac{1}{N_{s,c}}\right)^{i} \cdot \left(1 - \frac{1}{N_{s,c}}\right)^{fDS_{s,c}-i},$$

where the notation $X_{s,c}$ stands for X_s when dealing with check blocks and X_c for data-items. In particular, $DS_s = CB\alpha$ and $DS_c = kS_{min}$.

Figure 4.11 compares the number of fragments per node (core and spare member) in Datacube with the one needed in case of full replication and pure rateless erasure coding to guarantee an availability of 0.99. Recall that data-items are made of k fragments.

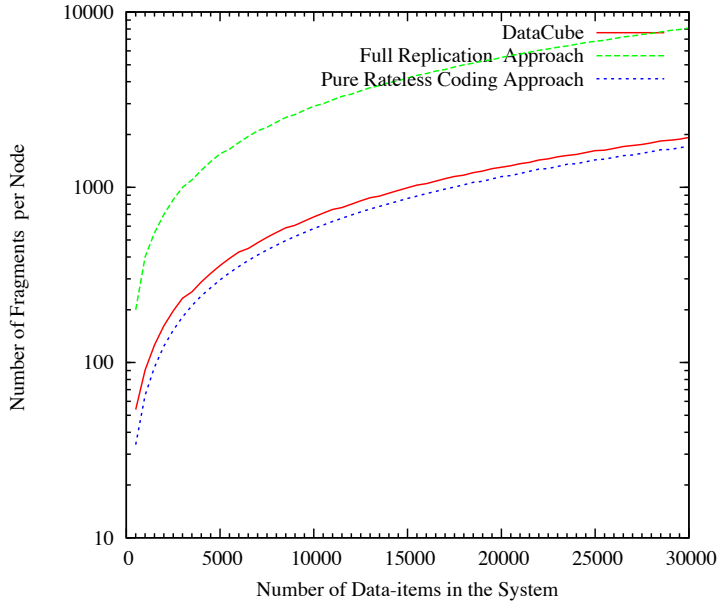


Figure 4.11: This graph shows the number of stored fragments per node in Datacube, in a full replication approach, and pure rateless erasure coding one as a function of the size of the system **N**. The replication factor used for the full replication approach is equal to 33 (see Figure 4.10) while the stretch factor for pure coding is equal to 10.

In these experiments, $N = 1,000$, $CB0 = 50$, the stretch factor is equal to 10 (see Figure 4.10), and $\alpha = 2$. Lessons learned from these experiments are two-fold: first, our approach guarantees a 3.5-fold savings WRT full replication, and second is very close to pure rateless erasure coding approach which clearly shows the low impact on storage overhead of the full data-items stored at S_{min} core members in Datacube.

4.3.3 Bandwidth Usage

We finally derive the total bandwidth needed per node for maintaining Datacube redundancy mechanism in presence of churn. Each time a node p leaves the system data-items or checked blocks p cached are copied over to new nodes (to keep the redundancy guarantees), while each time a new node p joins the system p needs to download all the data that match

to it. If we assume that nodes join the system at rate λ and leave it at rate ϕ, and that $\phi = \lambda$ (to keep the system size constant in average), the usage bandwidth per core node is, in expectation, equal to twice the size of fragments a node houses times λ. Note that, the rate at which core members leave the system is

$$1/\frac{S_{min}}{\log_2(N) - S_{min}} \tag{4.4}$$

less than the one spare members do. Figure 4.12, derived from results obtained in Figure 4.11, shows the required bandwidth per node needed for maintaining up to 1000 Tera Bytes of unique data in a system of $N = 10^6$ nodes (this corresponds to a very large video archive).

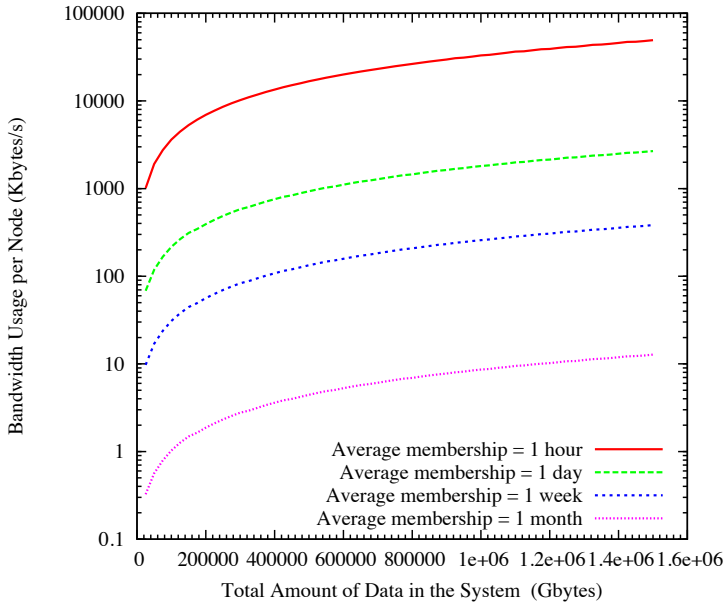


Figure 4.12: This graph shows the required bandwidth per node as a function of the churn for maintaining up to 1000 Tera Bytes of unique data at an availability at least equal to 0.99.

Clearly at a daily turnover rate, the required redundancy policy is too demanding to be supported by nodes, however, at a monthly turnover rate, continuous contribution of each node shrinks to less than $20K$bytes per second which is clearly compatible with classic ADSL rates.

4.4 Conclusion

In this chapter we have presented the details of Datacube's architecture. We have shown how the hybrid redundancy scheme employed by Datacube is used to achieve durable access to data, despite the existence of adversaries and churn. Datacube presents significant advantages when compared to other peer-to-peer storage architectures, such as those presented in the Section 2.4.5.

First, different from CFS and Total Recall that are based on ring structures, the hypercube structure used by Datacube increases the robustness of the architecture. The ring-based architecture of CFS and the tree-based architecture of Total Recall keep pointers to successor and predecessor neighbors at every node, in order to maintain the overlay architecture. In these approaches, repairing mechanisms are used to keep the pointers up to date. These repairing mechanisms are strongly affected by the churn because they have to be executed whenever a peer joins or leaves the overlay. The clustered overlay used by Datacube reduces the impact of churn on rearing mechanisms by organizing nodes into clusters and classifying nodes into two types. The different roles attributed to each type of node allow Datacube to keep only limited number of nodes (i.e., core members) responsible for maintaining the overlay structure. Therefore, Datacube limits the impact of churn on the overlay's architecture.

Second, in order to reduce the impact of churn on data availability, as we discussed in the Section 2.3.2, CFS and Total Recall use the Chord' successors list to apply replication on data-items and Oceanstore applies massive replication. The approach used by Oceanstore is clearly very expensive in terms of storage overhead, while the negative point of using Chord's successors lists used by CFS and Total Recall is that, in Chord each node replicates the data it is in charge of at $log_2 N$ adjacent nodes. Thus, if $\log_2 N + 1$ nodes form a collusion it is sufficient to permanently damage data objects. Datacube tolerates collusion within a given neighborhood and requires low storage overhead.

Regarding byzantine nodes, in Datacube any operation is performed according to the result of consensus agreement performed among core members at each cluster. Among those architectures presented in the Section 2.4.5, Oceanstore is the only architecture that deals with byzantine nodes. Oceanstore relies on an inner set of servers (called primary replicas) that run byzantine protocols in order to decide on every store (write) operation. After agreeing and storing the data at primary replicas, these servers store secondary replicas and archive replicas at other nodes. Moreover, only primary replicas perform byzantine protocols, consequently, secondary and archival replicas are more vulnerable to attacks.

Finally, in the analysis we have shown the benefits of hybrid replication over full replication in terms of data availability, storage overhead and bandwidth usage. In terms of data availability, we have compared our redundancy scheme and the replication factor imposed by classical full replication required to provide redundancy. The obtained results have shown the benefit of our approach over full replication in different sizes of systems. When evaluating the scalability, we have obtained a 3-fold savings by relying on Datacube's hybrid scheme compared to the results obtained for full replication. Regarding storage overhead we have obtained a 3.5-fold savings when comparing Datacube's hybrid scheme with full replication. In terms of bandwidth, our experiments have shown that by using Datacube, the impact on the bandwidth at each node decreases as the rate peers join and leave the system increases.

Part III

Evaluation

Chapter 5

Evaluation

Contents

5.1 Introduction

This chapter presents the evaluation of Datacube's redundancy scheme through a series of practical experiments. The main objectives of our experiments are the following. First, we want to know how significant is the impact of coding parameters on the coding process, for both Online and LT codes. How these parameters can influence on the generation of check blocks. Second, we want to understand how the coding parameters can impact on the performance of the recovery process. Finally, we want to know how interesting would be to use different policies to recover check blocks. How these policies could affect the performance of the decoding process.

5.2 Experimental Setup

5.2.1 Platform

The experiments were conducted over a Linux-based cluster of 100 dedicated nodes. In this cluster the configuration of each node is heterogeneous and it is distributed as follows: Forty eight nodes with 2 Intel Xeon processors (4 cores each) at 2,67GHz and 48 gigabytes of RAM memory. Ten nodes with 2 Intel Xeon processors (2 cores each) at 3,40GHz and 4 gigabytes of RAM memory. Ten nodes with 2 Intel Xeon processors (2 cores each) at 2,33GHz and 4 gigabytes of RAM memory. Ten nodes with 2 Intel Xeon processors (4 cores each) at 1,60GHz and 8 gigabytes of RAM memory. Thirty nodes with 2 Intel Xeon processors (4 cores each) at 2,33GHz and 8 gigabytes of RAM memory. All nodes run Fedora Linux and they are managed by the Sun Grid Engine.

5.2.2 Experiments Overview

We have implemented a prototype using Java language to evaluate the Datacube redundancy scheme. We have implemented both Online and LT coders in order to evaluate which of them would be more suitable for Datacube. Each coding schema was evaluated with a large range of input parameters (i.e. number of input blocks, file size, and other specific coding parameters). For Online codes, we used coding parameters ϵ and q varying from 0.01 to 0.9 and from 0.1 to 3, respectively. For LT codes, parameters δ and C were respectively selected from 0.01 to 0.9 and from 0.1 to 3. All the experiments were extensively performed. Each plot generated to represent an experiment actually represents the average of at least 50 experiments. In order to evaluate the coding performance of both coders in Datacube, we simulate a targeted attack on a specific cluster $\mathcal{A}$. The attack prevents any access to the data stored at the attacked cluster $\mathcal{A}$. By doing this, whenever we request a data item $\mathcal{D}$ stored at $\mathcal{A}$ we force $\mathcal{A}$'s neighbors to recover sufficiently enough check blocks from the system to rebuild the data item $\mathcal{D}$ by using the decoding process. The main objective in these experiments is to stress the use of rateless coders, as they are fundamental components of Datacube redundancy scheme.

5.3 Results

In this section we present our experiments and discuss the obtained results. We start by presenting some expected behaviors of the coding processes. First, we present the degree distributions used by each coder to generate check blocks. Next, we show the influence of coding parameters on the number of check blocks that need to be generated for the decoding process to be successful.

5.3.1 Degree Distribution of Check Blocks

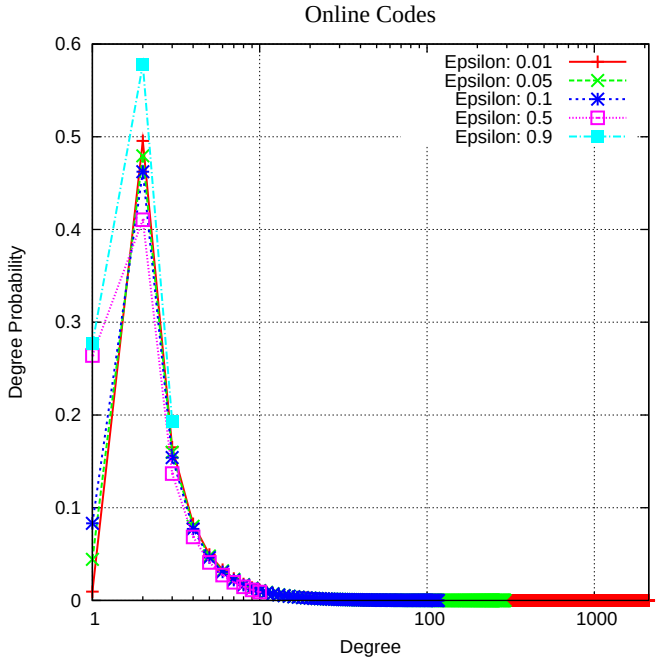
During the coding process, the choice of the degree of each check block is a critical point to guarantee an efficient recovery. The design of a good degree distribution is a challenging task because it must provide some specific features. A degree distribution must guarantee

that all input block are covered, i.e., every input block must be directly or not connected to at least one check block. This is required to guarantee a successful recovery of the original data. Online codes use composite blocks to ensure input block coverage, while LT codes rely one a robust degree distribution to ensure that. Moreover, the existence of at least one check block with degree one is required to start the decoding process, while check blocks with low degrees are crucial to keep the decoding process running. For instance, the lack of check blocks with low-degrees is the major weakness of an earlier version of the LT distribution. The current version of LT is expected to provide a constant number of check blocks with degree-one at each decoding round. On the other hand, providing too many check blocks with low degree may lead the decoder to face too much redundancy of check blocks during the recovery, which affects the performance of the decoding process and consequently the recovery of corrupted clusters. Depending on the coder used, distinguished coding parameters are used to configure the coder. Each coding parameter contributes to define the probability with which each degree will be chosen to generate a check block. These coding parameters (i.e., δ, ϵ, q, C) are described in Section 3.4.2.1 and Section 3.4.3.2.

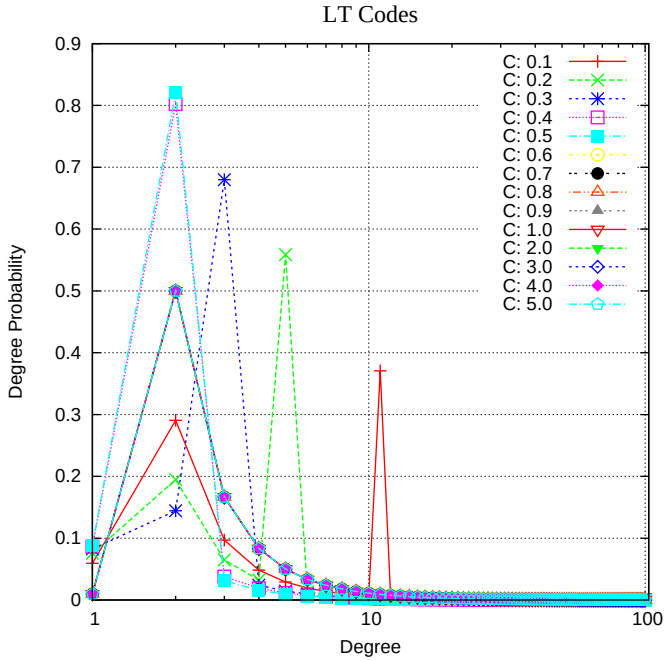
Figure 5.1(a) shows the distribution of Online codes. Interesting behaviors can be observed in this distribution. First, for values of $\epsilon \leq 0.1$, the probability $p(x)$ of selecting degree one is very small (i.e., $p(1) \leq 0.09$), while for $\epsilon \leq 0.5$, the expected number check blocks with degree-one is greater than 26%. The second point is that, the range of degrees a check block can be built from (i.e., F values) drastically augments with decreasing values of ϵ. For instance, for $\epsilon = 0.9$, $F = 3$ and for $\epsilon = 0.01$, $F = 2114$.

Figure 5.1(b) presents the degree distribution of the LT coder. The first observation in this distribution is the spike included at the probability of a degree $d = k/S$, with k the number of input blocks and $S = c\ln(k/\delta)\sqrt{k}$ a derived value that defines the expected number of degree-one check blocks throughout the decoding process. The spike at degree $d = k/S$ is the mechanism that ensures all input blocks coverage in LT codes. We can also see the impact of the parameter C on the choice of degrees. Increasing C values slightly augments the probability of degree-one check blocks (e.g., from 0.059 to 0.088 when value of C increases from 0.1 to 0.5). An interesting point to observe is that the probability of degree-one abruptly drops when C overcomes a specific value. For instance, at Figure 5.1(b) the probability $p(1)$ of selecting a degree-one when $C > 0.5$ drops to 0.010. The reason of this behavior is that the combination of these specific values of C and those of k and δ lead Robust Distribution of LT to tend to zero, which makes the Robust Soliton distribution of LT codes behaving exactly as its Ideal distribution. The value of C for which this phenomenon occurs will be referred to in the following as the cut-off value of C. Figure 5.2 shows how the cut-off value of parameter C evolves as a function of parameter δ for LT codes using $k = 100$. The decreasing on the probability of degree-one check blocks when value of C reaches its cut-off value (e.g., $C = 0.6$ for $k = 100$ and $\delta = 0.01$) is observed in Figure 5.1(b) when the probability of building a check block with degree one drops to 0.009. We show in the following how the cut-off value impacts on LT recovery performance.

Now, let us check the influence of the parameter δ on the degree distribution. Figure 5.3(a) shows the degree distribution of LT for different values of δ. The first observation is that the probability $p(1)$ of selecting a degree-one check block negligibly changes for any value of δ. This small variation of probability $p(1)$ is preserved even if we change the value of C, as we verify in Figure 5.3(b) where have plotted the degree distribution varying δ with a higher value of $C = 0.5$. For degrees greater than 1 the probability $p(d)$ of choosing a



(a) The probability $p(d)$ of choosing a degree d for a check block. Online codes with 100 input blocks and parameter $q = 3$.



(b) The probability $p(d)$ of choosing a degree d for a check block. LT codes using 100 input blocks and parameter $\delta = 0.01$.

Figure 5.1: Degree Distributions of Online and LT codes.

given degree d (with $d > 1$) increases with increasing values of δ. For instance, Figure 5.3(a) shows that for degree $d = 2$ the probability varying from 0.28 when $\delta = 0.01$ to 0.45 when

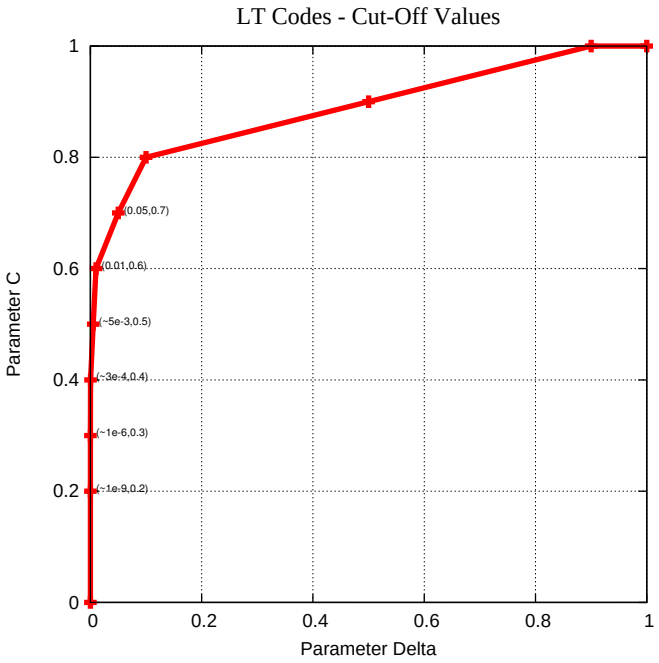


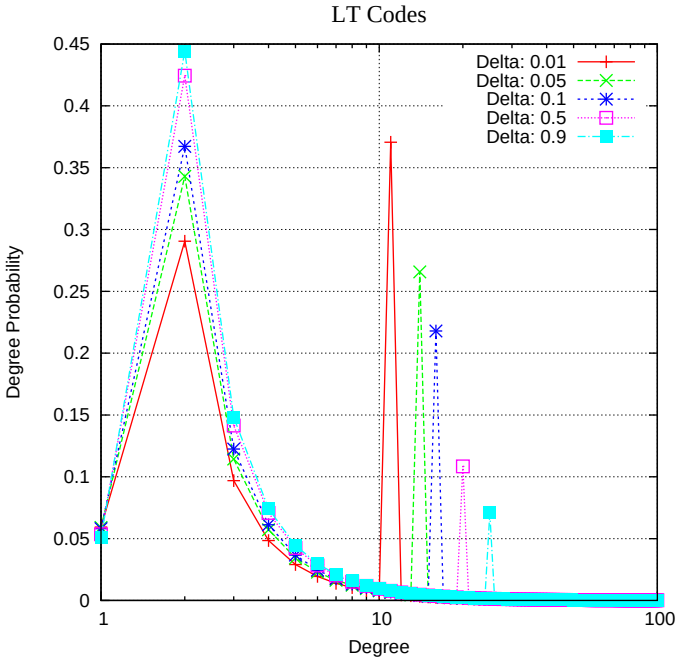
Figure 5.2: Cut-off values of parameter C as a function of parameter δ (rounded to the first decimal digit).

$\delta = 0.9$. Another interesting point to be observed is that the spike at the degree $d = k/S$ has its probability decreased and it moves to higher degrees with increasing values of δ, as shown in the Figure 5.3(a). It happens because increasing values of δ increases the relation k/S, moving the spike to higher degrees while its probability is reduced. On the other hand, by increasing the value of C it decreases the relation k/S moving the spike to lower degrees. Comparing Figures 5.3(a) and 5.3(b) we clearly see the influence of the spike on degree 2, by increasing the value of C from 0.1 to 0.5 we can see the spikes moving to lower degrees with higher probabilities making the probability of choosing degree 2 to increase from about 0.28 to 0.8. This leads the coder to create a significant number of check blocks with low degrees potentially redundant among themselves.

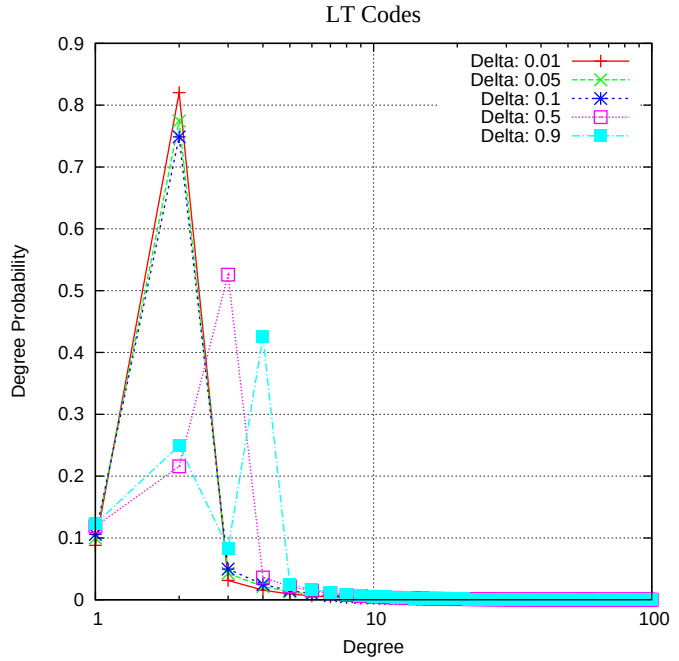
It will be shown later that these features have a great impact on the performance of Datacube coding and decoding processes.

5.3.2 Minimum Number of Check Blocks

In this section we inspect how coding parameters of both Online and LT codes affect the minimum number of check blocks needed to successfully recover the original data. The theoretical minimum number of check blocks $CB0$ is the number of check blocks that is required to be recovered in order to guarantee the decoding of the original data with high probability. For both Online and LT codes, the required number of check blocks depends on the distribution of degrees used to code the data. Hence, for each combination of coding parameters a specific number $CB0$ of check blocks is required. Figure 5.4 shows how both coders behave and their required $CB0$.



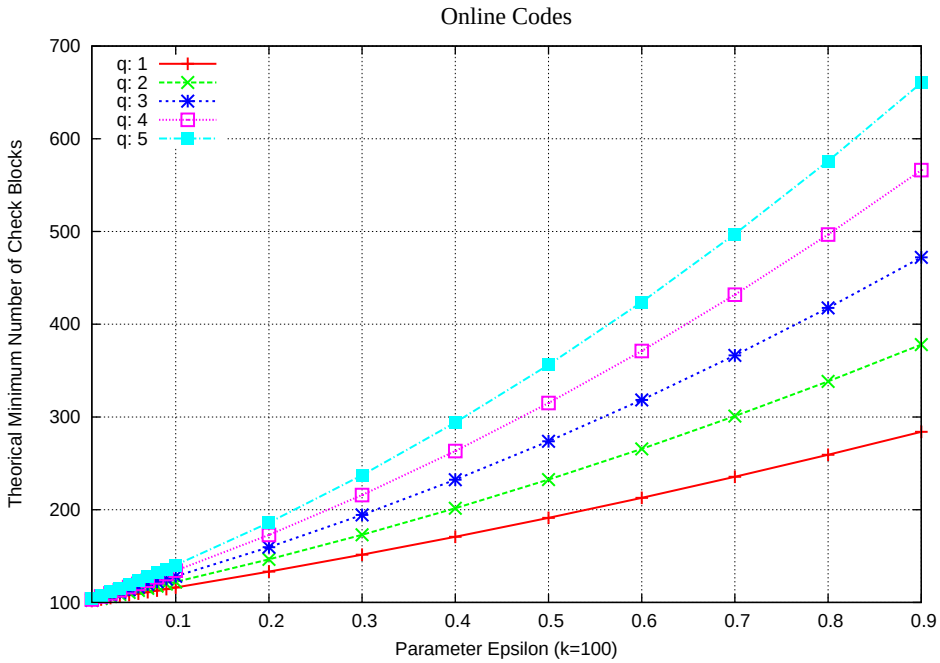
(a) LT codes with $k = 100$ and $C = 0.1$



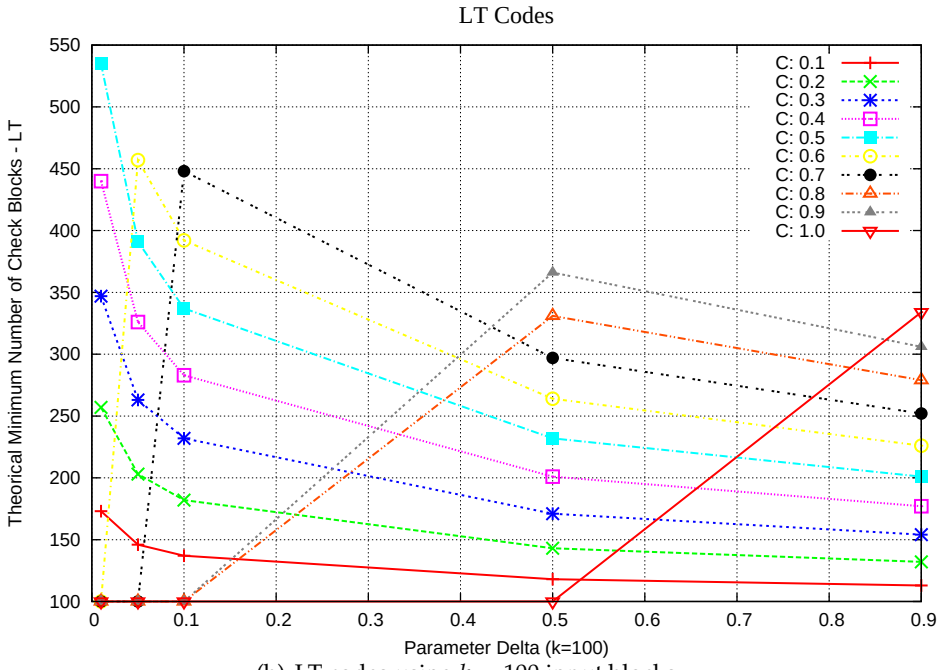
(b) LT code with $k = 100$ and $C = 0.5$

Figure 5.3: The influence of parameter δ on the distribution of degrees with LT codes.

Figure 5.4(a) shows the behavior of Online codes. The first interesting observation is that for low values of ϵ the number $CB0$ of check block required is slightly influenced by parameter q. For instance, when $\epsilon = 0.01$ the number $CB0$ of check blocks required varies about 2% when q varies from 1 to 5. With increasing values of ϵ the influence of q also



(a) Online codes using $k = 100$ input blocks.



(b) LT codes using $k = 100$ input blocks.

Figure 5.4: Theoretical minimum number of check blocks as a function of parameter δ.

increases. For instance when $\epsilon = 0.1$ and $k = 100$ the required $CB0$ augments by a small factor of 1.2 when q varies from 1 to 5. However, significant influence of q takes place when $\epsilon > 0.1$. Figure 5.4(a) shows that for $\epsilon = 0.9$ and varying q from 1 to 5, the minimum number of check blocks required to be generated is increased by a factor of 2.32 reaching 660 check

blocks. Thus, the required number $CB0$ of check blocks increases with increasing values of parameters ϵ and q.

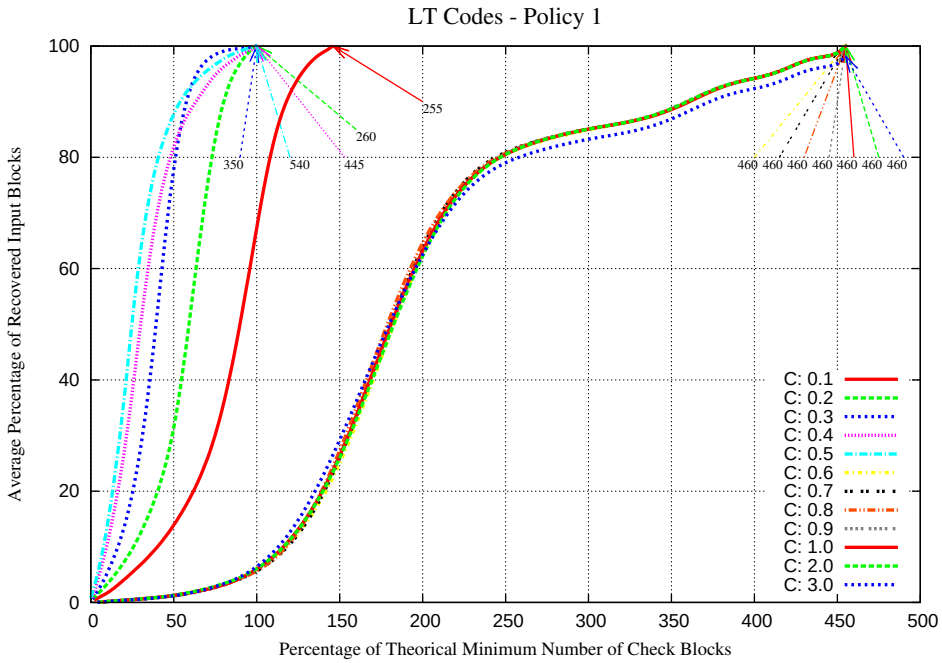
Figure 5.4(b) shows the impact of parameters C and δ on the theoretical minimum number of check blocks $CB0$ that need to be recovered with LT codes. In this figure we considered the number of input blocks $k = 100$. For any value of C lower than its cut-off point, the required number $CB0$ decreases with increasing values of parameter δ. For instance, for $\delta = 0.01$ and $C = 0.3$ the value of $CB0$ is almost 350. If the value of δ increases from 0.01 to 0.9, the value $CB0$ drops to almost 150. On the other hand, for a given value of δ, the required number $CB0$ increases with increasing values of C, if C is lower than its corresponding cut-off point. For instance, with $\delta = 0.1$ the number $CB0$ increases with increasing values of C until $C = 0.7$, which is the cut-off point for $\delta = 0.1$. Over the cut-off point, the number $CB0$ presents a particular behavior and requires the same number of input blocks. Figure 5.4(b) shows the behavior of $CB0$ of many values of C and δ.

5.3.3 Performance of Recovery Process

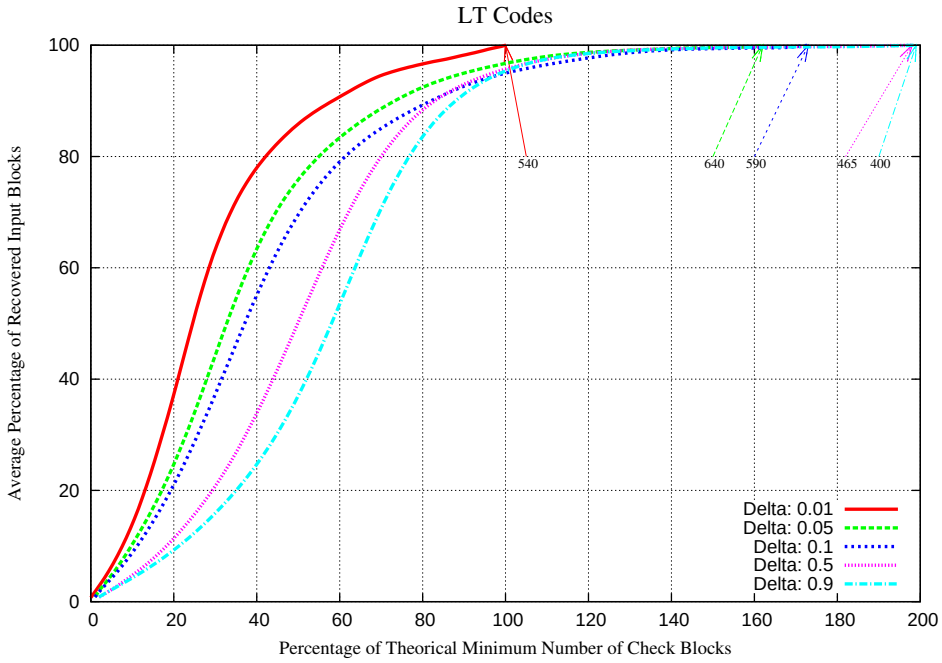
In this section we evaluate the practical performance of Datacube recovery process. To evaluate this feature we simulate an attack over a specific cluster $\mathcal{C}$ and analyze the number of check blocks that is needed to be recovered in order to successfully decode a specific data D stored at $\mathcal{C}$. We perform the same evaluations using both coding techniques, Online and LT codes. In all the figures in this section, curves are depicted as a function of the fraction of the predicted minimal value $CB0$. That is, an abscissa equal to 150 means 1.5 times $CB0$ check blocks. Arrows point to the number of check blocks that are needed to recover exactly the k input blocks, that is 100% of the data D. We first analyze the results for LT and Online codes when check blocks are randomly collected, the default procedure of rateless codes (called policy 1 in Datacube). Later on, we evaluate the results for other policies.

Figure 5.5(a) and Figure 5.5(b) show the impact of parameters C and δ on LT codes when policy 1 is run.

In Figure 5.5(a), the first observation is that by randomly collecting check blocks, for increasing values of C with $0.2 \leq C \leq 0.5$ the practical results matched the theoretical prediction of the required number $CB0$. We also observe that the number of check blocks needed to recover the original input blocks increases with increasing values of C as predicted. For $C = 0.1$ LT codes differed from the theoretical prediction in up to 48%. To recover $k = 100$ input blocks with $C = 0.1$, the theoretical $CB0$ is 173, while in average 255 check blocks are needed to ensure the recovery of data D. By observing the degree distribution for $\delta = 0.01$ and $k = 100$ we note that for $C = 0.1$ the spike is over degree 11, while for increasing values of C the spike moves to lower degrees (i.e., between degrees 2 and 5). Also, the probability of the spike is the highest probability among all the other possible degrees (i.e., about 38%). In this case, the decoder is likely to have many high-degree check blocks to decode and the absence of lower degrees makes the decoder to need to recover more check blocks than predicted, which does not happen for values of C greater that 0.1. When C reaches its cut-off value, from $C = 0.6$ onwards, the Robust Soliton distribution behaves as the Ideal distribution and the behavior of LT becomes independent from C values. From the degree distribution shown in Figure 5.3(a) it is easy to see that degree 1 has a very small probability of being chosen (i.e., $p(1) = 0.009$). This highly probable absence of check blocks with degree 1 strongly contributes for the decoding process to halt. Thus, the decoder needs to



(a) Parameters k=100 and $\delta = 0.01$.



(b) Parameters k=100 and $C = 0.5$.

Figure 5.5: Performance of LT Codes with different coding parameters.

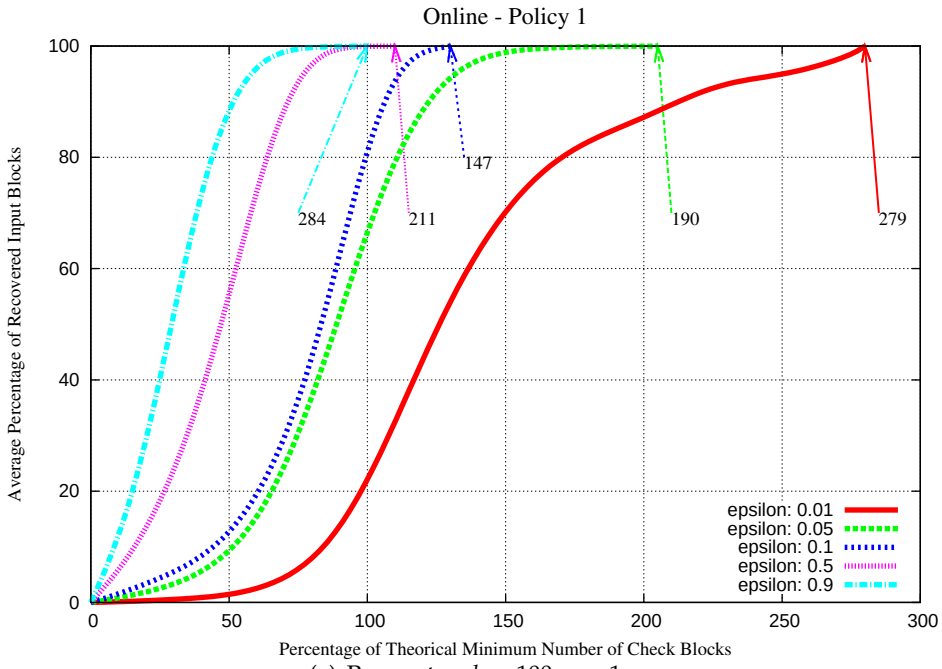
recover many other check blocks to achieve a successful decoding of the original data. For these values of C over the cut-off, in average 460 check blocks are necessary to successfully recover $k = 100$ input blocks.

Figure 5.5(b) shows the performance of LT codes for different values of δ. Different from

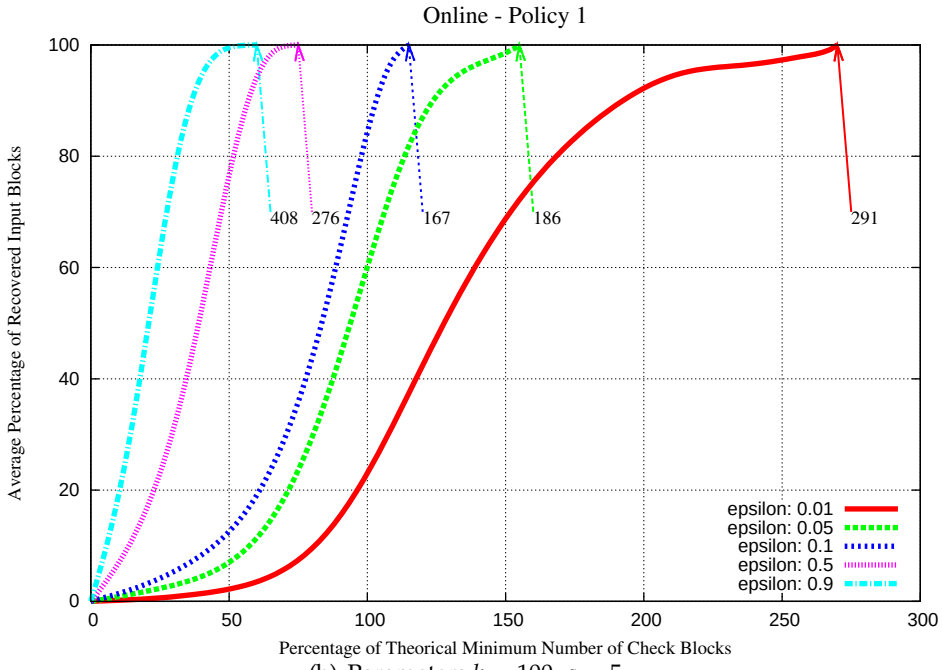
the last experiment, we use $C = 0.5$ in this experiment to check what happens when the spike is closer to lower degrees. From the obtained results we see that the average number of check blocks required to restore 100% of the input blocks matches the theoretical prediction when $\delta = 0.01$. However, it considerably augments for values of $\delta > 0.01$, reaching almost 200% when $\delta = 0.9$. Checking the distribution of degrees in Figure 5.3(b), we can see that the majority of check blocks is built from degrees 2, 3 and 4. With an increased value of C the spike is in lower degrees which increases significantly the probabilities of choosing these degrees 2, 3 and 4. Considering that for increasing values of δ the number $CB0$ of check blocks significantly decreases, from 540 to 200 when δ increases from 0.01 to 0.9. Increasing values of δ increase the probability of having redundant degrees of lower degrees (i.e., 2,3 and 4) due to their high probabilities of being chosen. Then, the decoder need to collect more check blocks to recover 100% of the input blocks as δ augments, diverging from the expected values.

Now let us analyze results obtained for Online codes. Figures 5.6(a) and 5.6(b) present results for different values of ϵ and with $q = 1$ and $q = 5$, respectively. Comparing with Figure 5.5(a), a preliminary observation drawn from Figure 5.6(a) is that, varying ϵ value in Online codes leads in average to a lower range of $CB0$ values than what is obtained when varying the parameter C in LT codes. In our results Online codes $CB0$ varies only from 147 to 284 while in LT $CB0$ varies from 255 to 540. By checking the Figure 5.4 we see that the difference is expected to be larger in LT than with Online codes, as verified in this experiment. In both cases, the expected variation in the number of check blocks to recover is lower than the theoretical variation expected (i.e., 190 for Online and 440 for LT).

Surprisingly experimental results have shown that in Online codes the required number of check blocks to be collected for a successful recovery does not vary proportionally to the parameter ϵ. Actually, the values of ϵ on the extremum of the range (i.e., 0.01 and 0.9) presented the most significant values of $CB0$, while it decreased to reach its minimum value at $\epsilon = 0.1$. Specifically, for $\epsilon = 0.9$, the number of auxiliary blocks is large but the range of possible degrees is very small (i.e., equal to 3) due to the current configuration of the degree distribution. Thus a large number of redundant check blocks are a priori collected, as we previously mentioned regarding to LT, which demands for more check blocks to successfully recover all input blocks. Now for $\epsilon = 0.01$, the number of auxiliary blocks is small but they form a complete bipartite graph with their associated input blocks, which clearly makes them useless. Moreover the probability of having degree-one check blocks is very small and the probability of having degree-two is large (i.e., 50%). However as the space degree F is very large too, the likelihood of having some very high degree check blocks is not null, and thus a large number of check blocks need also to be collected. Finally, for $\epsilon = 0.1$, the distribution of check blocks degrees is likely to be balanced, in the sense that the proportion of degree-one is 9% which is relatively high, but not too high to prevent redundancy. The proportion of degree-two is 47% which combined with degree-one allows to cover many input blocks. Finally, degree-three and degree-four check blocks are also useful, respectively 15% and 4%,. This clearly make this value of ϵ optimal, which is confirmed by the fact that in the practical evaluation it has required the lower number check blocks to successfully recover 100% of input blocks. The same argument applies for larger values of q as shown in Figure 5.6(b).



(a) Parameters $k = 100$, $q = 1$.



(b) Parameters $k = 100$, $q = 5$.

Figure 5.6: Performance of Online Codes with different coding parameters.

5.3.3.1 Performance of Datacube Policies

Now, let us show the results obtained by changing the policy used to recover check blocks from other clusters. In the previous experiments we randomly selected check blocks from the

system, that is the default procedure proposed by theoretical rateless codes, which we call policy one in Datacube. In Datacube other policies are proposed in order to achieve better performance, as presented in Section 4.2.4. Let us verified how these policies can contribute with the performance of the recovery process.

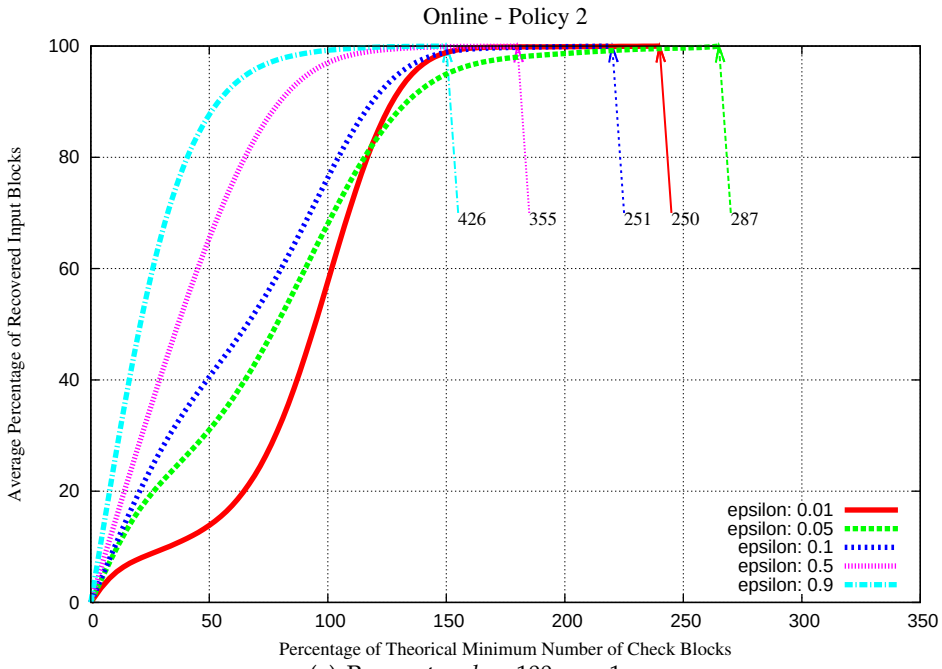
Figures 5.7(a) and 5.7(b) show the impact of policies 2 and 3 on the performance of Online Codes. The first clear observation noticed in both Figures 5.7(a) and 5.7(b) is that for both cases (i.e., policy 2 and 3) the decoder starts the decoding process before, if compared to policy 1. This is observed for all values of ϵ but more significantly for lower values. Specifically, with $\epsilon = 0.01$ the decoder needs to recover almost 50% of $CB0$ to start decoding the first input blocks in policy 1. In policy 2 with 50% of $CB0$ 15% of input blocks are already decoded while with policy 3 once 50% of $CB0$ check blocks are collected about 20% of the original message is decoded. The reason for such a behavior is that degree-one check blocks (and degree-two for policy 3) are first recovered and they ensure that, as soon as these check blocks arrive they are able to decode a significant fraction of the input blocks. On the other hand, the priority given to check blocks built from low degrees overwhelms the decoder with many redundant degrees which requires more check blocks to be recovered to successfully decode 100% of input blocks. It explains why the number of check blocks needed to decode 100% of input blocks in policy 2 and 3 is in average higher than in policy 1. The only value of ϵ that has its average of check blocks decreased in policies 2 and 3 is when $\epsilon = 0.01$. In this case, the number of check blocks with degree-one is very low since the probability of building a check block with degree one is extremely low (i.e., $p(1) = 0.009$). Thus, the number of recovered check blocks can be reduced up to 10% for $\epsilon = 0.01$ if we use police 2 instead of default policy 1.

Now, let us analyze Figure 5.8 that shows what happens when policy 4 is used. In policy 4, each check block recovered is the outcome of the bipartite graph rebuilt at recovery time. The policy 4 guarantees the full recovery of input blocks in a linear number of check blocks. As expected, this optimal policy behaves very well and each single recovered check block is useful for the decoding process. For instance, for $\epsilon = 0.01$ and $q = 1$, 102 check blocks are enough to successfully recover $k = 100$ input blocks, which corresponds exactly to $CB0$. For $\epsilon = 0.9$ only 40% of $CB0$ is needed to recover the entire original data.

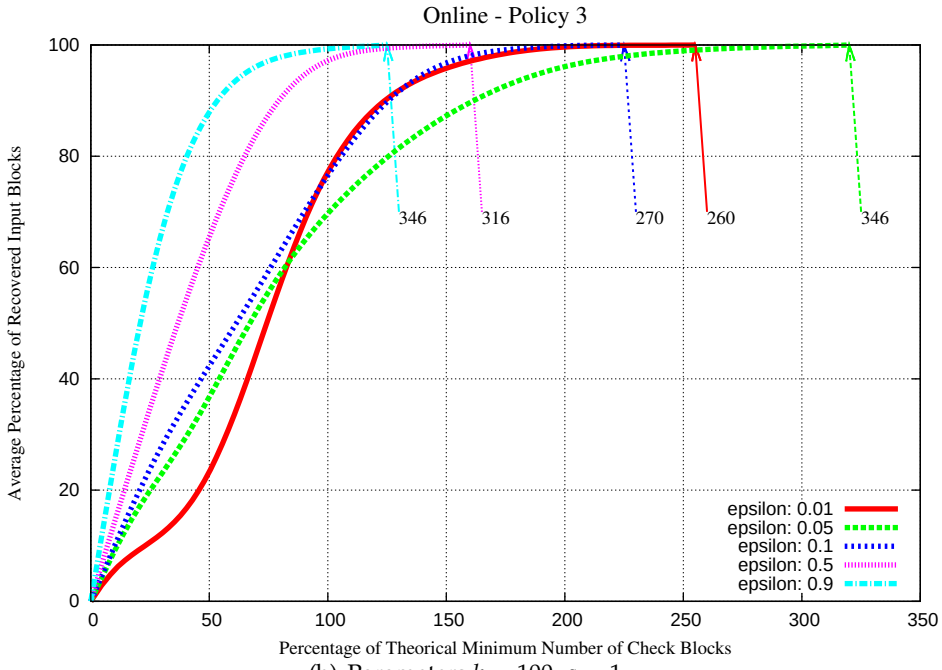
Thus, we observed that Online codes recovers the input blocks in average with less check blocks than with LT codes. However, our experiments show that the impact of both parameters ϵ and q is not linear, as the expected behavior presented in Figure 5.4. Moreover, whatever policy we use to collect check blocks, the obtained results for Online codes have shown that greater values of ϵ (i.e., 0.5 and 0.9) present the closest performance to the expected percentage of the theoretical number of $CB0$ needed to recover the original data. However, results with $\epsilon = 0.1$ has presented the best performance in terms of minimal number of check blocks to be collected in order to recover the original data.

Now let us analyze the impact of policies 2 and 3 on LT codes. Figures 5.9(a) and 5.9(b) present the obtained results.

Similar to Online codes, we observe that there is an augmentation in the number of check blocks required to recover 100% of the input blocks for polices 2 and 3, compared to the results obtained for policy 1. Another intriguing point observed is that policies 2 and 3 present similar behaviors. Specifically for all values of C lower than its cut-off value the probability of having a check block built with degree one is very low (i.e., from 0.059 to 0.088). When C reaches its cut-off value, this probability drops to 0.01. Hence, by giving



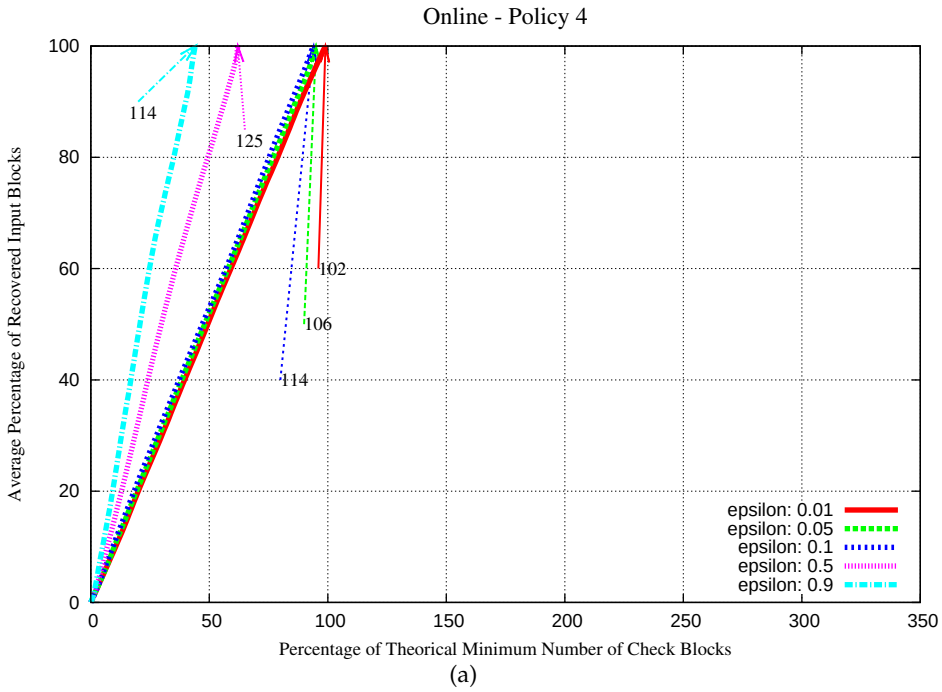
(a) Parameters $k = 100$, $q = 1$.



(b) Parameters $k = 100$, $q = 1$.

Figure 5.7: Performance of Online Codes with Datacube Policies.

priority to degree-one check blocks with both polices 2 and 3, does not seem to be effective due to the scarcity of degree-one check blocks. Second, for low values of C there is a high probability of having many check blocks built with high degrees as the spike moves to high degrees. For instance, with $k = 100$, $\delta = 0.01$ and $C = 0.1$ the probability of having a

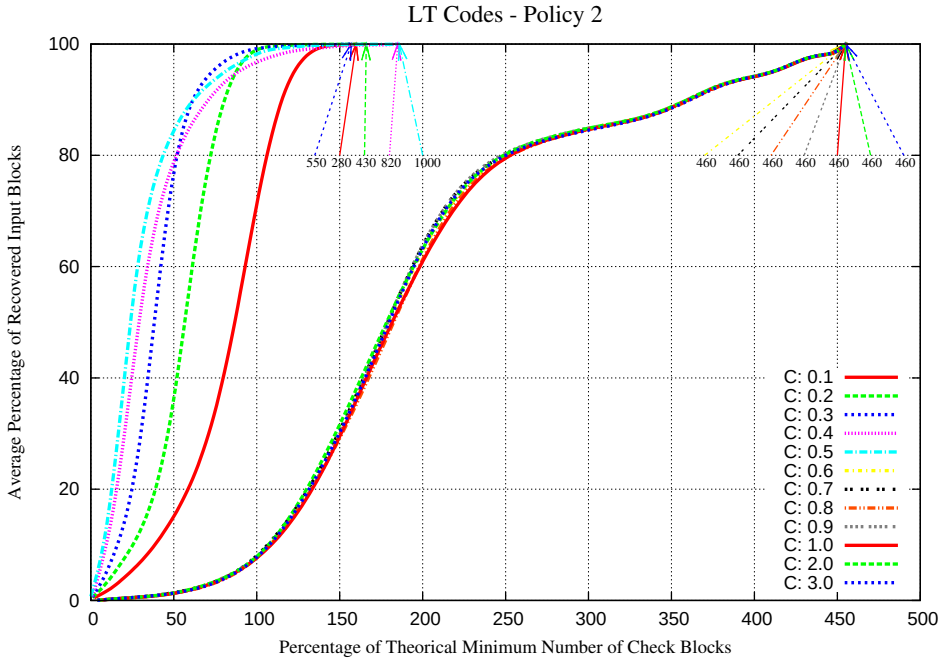


(a)

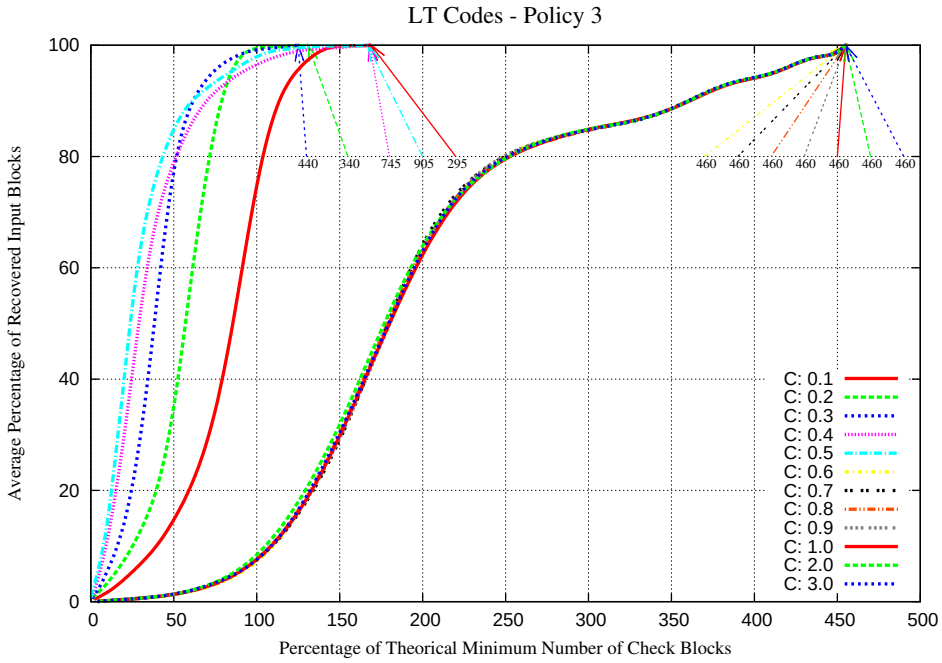
Figure 5.8: Performance of Online Codes with Datacube Policy 4. Parameters $k = 100$, $q = 1$.

check block built from degree 11 is 37.06%. Hence, the scarcity of degree-one check blocks and the abundance of check blocks with high degrees possibly leads the decoder to halt. As the value of C increases, the dependency of check blocks with high degree decreases as the spike move to lower degrees. However, the probability of lower degrees become extremely high, reaching in some cases 80%. This leads peers to recover a large number of redundant check blocks that are not useful for the decoding. Then, performance of policies 2 and 3 is significantly affected by the choice of δ and C because by given the priority only for degrees one (respectively degree two for policy 3) makes the decoder to collect either to many redundant low degrees or the decoding process to halt due to the lack of degree one. Therefore, we conclude that in this situation randomly collecting check blocks to recover the original data is a better choice when compared to policies 2 and 3.

Now, let us check the performance of policy 4 presented in Figure 5.10. Similar to Online codes, policy 4 in LT codes presents an excellent performance when compared to the other policies. There are some interesting points to be emphasized. First, for all values of C under the cut-value 100% of the original data is successfully decoded with no more than $CB0$ check blocks. Second, for higher values of C (e.g., $C = 0.4$ and $C = 0.5$) input blocks are recovered in a linear number of check blocks while for lower values (i.e, $C \leq 0.1$) up to 98% of input blocks are recovered in a linear number of check blocks and then few more check blocks are needed in order to complete the other 2%. From the distribution we know that at lower values of C the spike moves to the higher degrees, decreasing the probability of choosing lower degrees. The abundance of these check blocks with high degrees (for low values of C) seems to prevent the existence of a sufficiently enough number of lower degrees to keep the decoding process running and thus few other check blocks are required to decode the remaining input blocks. Finally, for values of C greater than the cut-off a similar problem



(a) Parameters k=100 and $\delta = 0.01$.



(b) Parameters k=100 and $\delta = 0.01$.

Figure 5.9: Performance of LT Codes with Datacube Policies.

seems to take place. For these values, the probability of building a check block of degree-one is much smaller. In the cut-off values there is no spike and the coverage of input blocks can not be ensured. The distribution of degrees provides a very large number of redundant degree-2 check blocks check blocks (e.g. degree 2 with probability of 0.5). Despite these

factors, recovery is successfully performed in average with 262 check blocks, reducing the required number of check blocks in 43.47% if compared with the results obtained when using policies 1, 2 and 3.

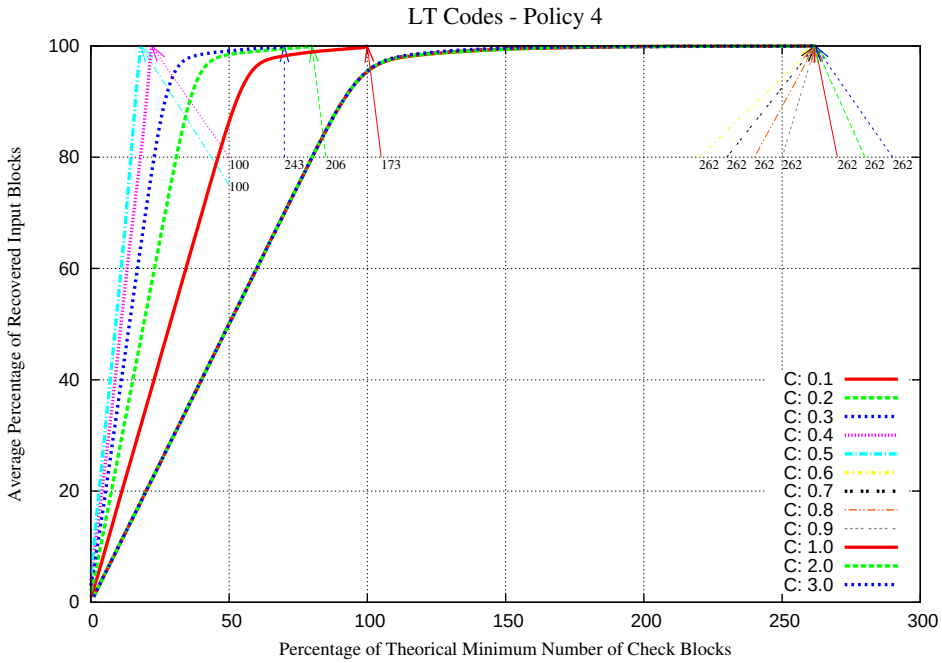


Figure 5.10: Performance of LT Codes with Datacube Policies. With Parameters k=100 and $\delta = 0.01$.

Thus, the experimental results obtained using LT codes are closer to the expected theoretical behavior in lower values of δ (i.e, 0.01) than in higher values as presented in Figure 5.5. The cut-off value on C has a significant impact on the performance of LT. In our experiments we observed better performances in larger values of C close to the cut-off value (e.g., $C = 0.5$).

5.3.4 Computational Cost of XOR Operations

In this experiment we evaluate the computational costs of both LT and Online decoding process. The computational cost is quantified by the number of XOR operations required to execute in order to successfully recover the input data. Note that, the unit of XOR used by the coder and decoder matches the length of an input block. We evaluate the number of XOR operations in both LT and Online coding as a function of their respective parameters (C, δ) and (ϵ,q). Results are presented in Figures 5.11(a) and 5.11(b).

The main observation drawn from Figure 5.11(a) is that the number of XOR operations quickly drops down with increasing values of C and after some specific value it stabilizes. Specifically, for small values of C, the degrees with higher probabilities are those defined by the spike of each C value in the distribution. The spike has usually high degree for small values of C and its degree decreases with increasing values of C. Hence, low values of C are likely to perform a significant number of XOR operations due to the degrees of spikes, as we verify in the Figure 5.11(a). For instance, for $\delta = 0.01$ and $C = 0.1$ the probability

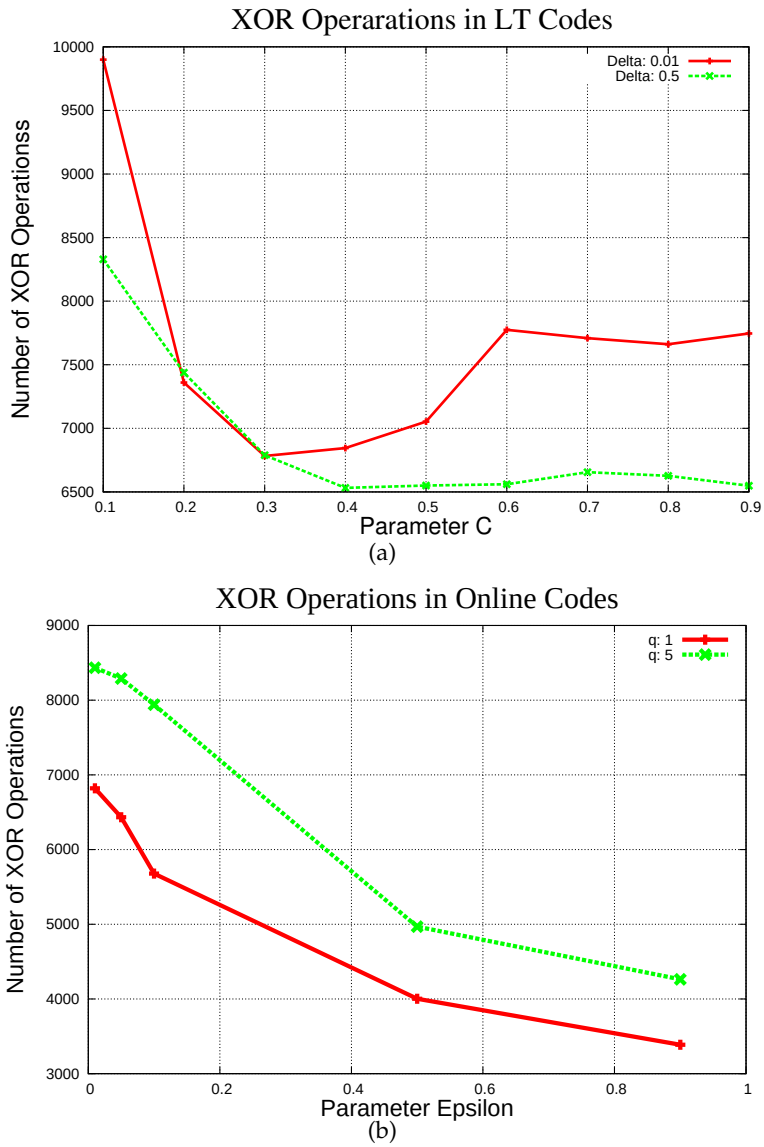


Figure 5.11: Number of XOR operations performed by LT and Online codes.

of selecting the spike with degree-11 is around 38%, while for degree-one and degree-3 the same probability is only 10%. As values of C increase the spike degree quickly decreases becoming a low-degree spike. It reduces its influence on the number of XOR operations performed by the decoder. Although the minimum number of check blocks increases with the increasing of C, most of check blocks are likely to have degrees 2 and 3, due to the distribution. For instance, for $C = 0.4$ or $C = 0.5$, the probability of selecting a degree-2 reaches 80%. When value of reaches its cut-off, the number of XOR operations become stable. This stabilization can be attributed to the fact that the number $CB0$ of check blocks required is constant and the degree distribution becomes an Ideal distribution.

Regarding to Online codes, Figure 5.11(b) presents the evaluation of the computational cost as a function of ϵ and q. Clearly, ϵ has a strong impact on the number of XOR operations

that need to be performed. Specifically for increasing values of ϵ, the number of XOR operations drastically decreases, reaching an average number of XOR operations much lower than the average of LT codes. This behavior seems to result from the combination of two phenomena. For ϵ from 0.01 to 0.1, the number of check blocks needed to successfully recover the input data decreases. In the meantime, F equally decreases with an increasing probability of generating degree-one and degree-two check blocks. Thus, both phenomena cumulate and give rise to the diminution of the number of XOR operations as observed in Figure 5.11(b). Remind that, check blocks with degree-one do not trigger any XOR operations as their data content are only copied. Now, ϵ from 0.1 to 0.9, the number of check blocks increases as presented in Figure 5.4, and the range F of possible degrees drastically decreases together with a high probability of generating degree-one and degree-two check blocks. Thus despite the fact that a large number of check blocks need to be decoded, most of them have a degree-one or degree-two, and thus most of them do not trigger XOR operations, which explains the negative gradients of the curves in Figure 5.11(b).

5.4 Conclusion

In this chapter we have studied how significant is the impact of coding parameters on the performance of rateless codes. We mention below some of the main lessons learned from these experiments.

First, we have observed the significant impact of the coding parameters on the distribution of degrees of the generated check blocks. For instance, on Online codes the probability of building check blocks of degree one increases with increasing ϵ. The range of possible degrees drastically increases with decreasing values ϵ. For LT codes, there exists a cut-off point in the probability for certain values of parameter C. The cut-off point results from the combination of specific values of C and k and δ that leads the Robust Soliton distribution to behave exactly as Ideal distribution. Thus, the cut-off value has an important impact on the probabilities of choosing specific degrees. There is also a spike in the probability of degree $d = k/S$. For increasing the value of C the spike moving to lower degrees given a higher probability of choosing degree d, while increasing values of δ move the spike to higher degrees with a lower probability of d.

Second, we have learned how the choice of the coding parameters can affect the performance of the decoding process. This is of a great importance to understand how good (or bad) Datacube's redundancy scheme can operate. For instance, in Online codes, the required number $CB0$ increases with increasing values of parameters ϵ and q. However, for low values of ϵ the required number of check blocks $CB0$ is slightly influenced by parameter q. In LT codes, for any value of C lower than its cut-off point, the required number $CB0$ decreases with increasing values of parameter δ, while for a given value of δ, the required number $CB0$ increases with increasing values of C, if C is lower than its cut-off point. Over the cut-off point, the number $CB0$ is equal to the number of input blocks. In LT codes, we have noted that the performance for different values of C follows pretty much the expected behavior while for different values of δ diverges from the expected behavior. Another, general observation is that over the cut-off value of C performance is very poor. The parameter C must be chosen such that the cut-off is avoided.

Finally, we have studied different approaches that can be used during the recovery pro-

cess. We have observed that giving priority to degree-one check blocks with both polices 2 and 3 does not seem to be effective due to the absence of degree-one check blocks. The performance of policies 2 and 3 is significantly affected by the choice of δ and C, while randomly collecting check blocks performs better. As expected, policy 4 presents an excellent performance when compared to the other policies, recovering input blocks in a linear number of check blocks. Using Online codes, the impact of ϵ in terms of check blocks needed to recover is lower than in LT, meaning that in average Online codes recover the original data with lower number of check blocks. Varying the parameter ϵ in Online does not provide the expected variation on the number $CB0$. The number of recovered check blocks is reduced up to 10% for lower values of ϵ if we use police 2 instead of policy 1. On the other hand, policy 4 guarantees the full recovery of input blocks in a linear number of check blocks.

Part IV

Conclusions

Chapter 6

Concluding Remarks and Perspectives

Contents

6.1 General Conclusions

The demand for storage resources has notably increased over the past few years. In addition to that, current predictions [43] show that this phenomenon must persist and be intensified in the near future. Properties of peer-to-peer architectures makes it well positioned to provide scalable storage. However, peer-to-peer architectures aiming at providing persistent storage face two main challenges, the fundamental freedom the participating members have to join and leave system at any time and the unavoidable presence of Byzantine members.

In this context, the main goal of this thesis was to propose Datacube, a peer-to-peer storage architecture designed to provide persistent data storage. As a first contribution we have presented the design of Datacube. Datacube leverages the power of clustering and benefits from the properties of the recently proposed rateless codes to efficiently provide persistent data storage despite churn and the presence of colluding malicious nodes. We have shown that, the hybrid redundancy scheme of Datacube offers redundancy at peer and cluster levels, allowing the recovery of the whole set of data stored at clusters. To the best of our knowledge, Datacube is the first storage architecture that provides data persistency while facing both churn and malicious targeted attacks.

Datacube's design has advocated for the use the rateless codes in our hybrid redundancy scheme instead of using traditional erasure codes. Although traditional erasure codes present excellent features (e.g., its optimality on the number of blocks needed to recover the

original data), some fundamental properties of rateless erasure codes make them more suitable for dynamic peer-to-peer systems. Indeed, in traditional erasure codes, encoding and decoding process times scale quadratically with the number of input blocks, which makes them computationally too expensive for large files, while rateless codes offer code and decode with linear time complexity. On the other hand, in rateless erasure codes the loss-rate does not need to be estimated beforehand, as required by traditional erasure codes. This makes it easier to the storage architecture to adjust the redundancy level according to the dynamics of the system. Another advantage of rateless erasure codes over traditional erasure codes is that check blocks are independently generated, which makes it possible to substitute a single lost check block just by coding another one. In traditional erasure codes, a complete set of new check blocks would be required instead. Thus, we strongly believe that the new class of rateless erasure codes are the best option on peer-to-peer environments.

After presenting the design of Datacube, we have presented our analysis to evaluate its efficiency. In this analysis we have compared the hybrid redundancy scheme of Datacube with traditional replication in terms of data availability, storage overhead and bandwidth usage. In terms of data availability, we compared the stretch factor of Datacube and the replication factor imposed by classical full replication required to get specific data availabilities (i.e., 0.99, and 0.999). The obtained results have shown the benefits of our approach over full replication in different sizes of systems, as presented in Chapter 4. We have also verified the scalability of Datacube's redundancy scheme. The obtained results have shown a 3-fold savings by relying on Datacube's hybrid scheme compared to the results obtained for full replication. Note that, this evaluation was made using a system with 2^{16} peers. Regarding storage overhead we have obtained a 3.5-fold savings when comparing Datacube's hybrid scheme with full replication. In terms of bandwidth, our experiments have shown that by using Datacube, the longer the peer stays connected the lower is the impact of the redundancy scheme on the bandwidth at each node. The obtained results have shown that, in a system with 10^6 peers the continuous contribution of each node shrinks to merely 20 kilobytes/second at each peer when the turnover rate of join and leave operations is in average 1 month.

As highlighted in Section 4.4, Datacube presents significant advantages when compared to other peer-to-peer storage architectures. Different from CFS and Total Recall that are based on ring structures, the hypercube structure used by Datacube increases the robustness of the architecture. The repairing mechanism of the ring-based architecture of CFS and the tree-based architecture of Total Recall are strongly affected by the churn. The clustered overlay of Datacube reduces the impact of churn on repairing mechanisms by giving different roles to each type of nodes in the cluster. It allows Datacube to significantly limit the impact of churn on the architecture. Regarding to the impact of churn on data availability, CFS and Total Recall rely only on Chord' successors list. By applying a promiscuous replication strategy, Oceanstore presents a undesirable large storage overhead because data objects can be stored anywhere at anytime. There's no bound in the number of replicas created on Oceanstore. For instance, any node holding a specific replica can store other copies of this replicas at neighbor nodes in order to reply requests more efficiently. From the performance point of view it is an interesting feature, however it significantly increases the cost in terms of storage overhead.

The negative point of using Chord's successors lists as used by CFS and Total Recall is its vulnerability to collusion attacks. As shown in Chapter 4.1 the design of Datacube ensures

robustness to churn and collusion with low storage overhead. Regarding byzantine nodes, Oceanstore relies on a set of well defined servers that run byzantine protocols and decide on every write operation. These servers store only primary replicas, while secondary replicas and archive replicas are stored into unsecured nodes. As only primary replicas perform byzantine protocols, secondary and archival replicas are more vulnerable to attacks. In Datacube, all operations are performed in the clusters are the result of a consensus agreement performed among core members. Compared to Oceanstore this approach improves the load balancing of write operations on the system and guarantees that data can be restored even if clusters are attacked.

Additionally, we have implemented both LT and Online rateless coders in order to evaluate the recovery and computational performances of our redundancy scheme. This evaluation has provided good insights into the properties of the coders and their respective parameters. By performing these experiments we could verify the importance of the choice of the coding parameters and how these parameters can impact in the final performance of the recovery process. In our results, Online codes recovered input blocks in average with less check blocks than with LT codes. Different from expected behavior, the impact of both parameters ϵ and q in Online codes was not linear. Moreover, whatever policy we used to collect check blocks, the obtained results for Online codes have shown that higher values of parameter ϵ presented the closest performance to the expected of the theoretical number of $CB0$ needed to recover the original data. Online codes has presented the best performance in terms of minimal number of check blocks to be collected with $\epsilon = 0.1$. With LT codes we have obtained results closer to the expected theoretical behavior with lower values of parameter δ than with higher values. Another important remark was that, the cut-off value of parameter C has a significant impact on the performance of LT codes. Our results have shown the better performance for LT codes in larger values of C close to the cut-off point. The main achievement of these results is to emphasize the importance of the choice of the correct values of coding parameters taking on account the impact they can produce on the recovery process. These values should be considered by developers and engineers when developing applications using rateless codes.

6.2 Open Points and Perspectives

Providing persistent and scalable storage architectures is definitely a subject of an increasing concern. Datacube presents a reliable and realistic approach to provide persistent and scalable data storage. However, there are important points that still need to be better investigated and improved. We mention few of them in the following.

First, our work has dedicated a large effort to investigate the tunning of coding parameters of Datacube hybrid redundancy scheme. This brings valuable results to be applied in practice, however the experiments were performed using a fixed number of input blocks. The size of files varied from kilobytes to megabytes but the number of input blocs were preserved. One interesting extension for our evaluation would consist in evaluate the behavior of the coders with larger number of input blocks to check if the performance changes or not.

Second, at this stage Datacube's model considers only read operations. Once a data file is created subsequent access to the same file are read-only. When a file is updated, the updated file is considered as a new data-item and Datacube does not track the sequence of updates.

Another interesting extension would be to study and evaluate a new mechanism to handle data updates in Datacube. We should answer the following questions. Could we use clusters to store old versions and store only the most recent version on core members? Would the hybrid redundancy scheme based on rateless codes remain a good option with read and write operations? How important would be the impact of this mechanism on the bandwidth consumption?

Another point concerns the cluster's recovery process. At this stage, Datacube randomly spreads the check-blocks over system in order to balance the load and storage overhead among the clusters. Questions that still remain opened are, could we use other data placement strategies to improve the recovery process? Would it be better to store all check blocks at x closest neighbors instead of spreading over the system?

Another point concerns data privacy. Datacube is an architecture that was designed to be an open system and used by anyone connected to the Internet aiming at sharing few of their resources in order to collaborate to (and benefit from) a bigger storage facility. At this stage, Datacube does not take neither data privacy nor access control into account. In this context, personal data is potentially vulnerable and users would not few motivated to use Datacube. One fundamental improvement to Datacube is the inclusion of a mechanism to guarantee that stored data-items are not publicly exposed. Moreover, access control mechanisms could also be added to allow data owners to delegate access to other users.

Finally, in this work our experiments were focused on key points of the design of Datacube. For Datacube to be a ready-to-use software, much more implementation and evaluation are still needed. However, the established ideas are promising.

Bibliography

[1] Gnutella. http://en.wikipedia.org/wiki/Gnutella2.

[2] K. Aberer, L.O. Alima, A. Ghodsi, S. Girdzijauskas, S. Haridi, and M. Hauswirth. The essence of P2P: A reference architecture for overlay networks. In *Proceedings of the 5th IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 11–20, 2005.

[3] M.K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proceedings of the 35th International IEEE Conference on Dependable Systems and Networks (DSN)*, pages 336–345, 2005.

[4] E. Alchieri, A. Bessani, J. da Silva Fraga, and F. Greve. Byzantine consensus with unknown participants. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS)*, pages 22–40, 2008.

[5] E. Anceaume, F. Brasiliero, R. Ludinard, and A. Ravoaja. PeerCube: an hypercube-based P2P overlay robust against collusion and churn. In *Proceedings of the 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 15–24, 2008.

[6] I. Ari, B. Hong, E.L. Miller, S.A. Brandt, and D.D.E. Long. Modeling, analysis and simulation of flash crowds on the Internet. Technical Report UCSC-CRL-03-15, Storage Systems Research Center Jack Baskin School of Engineering University of California, Santa Cruz Santa Cruz, CA, 2004.

[7] B. Awerbuch and C. Scheideler. Towards a scalable and robust DHT. In *Proceedings of the 18th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 318–327, 2006.

[8] O. Baldellon, A. Mostéfaoui, and M. Raynal. A necessary and sufficient synchrony condition for solving byzantine consensus in symmetric networks. In *Proceedings of the 12th International Conference on Distributed Computing and Networking (ICDCN)*, pages 215–226, 2011.

[9] R. Baldoni, J.M. Hélary, M. Raynal, and L. Tangui. Consensus in Byzantine Asynchronous Systems. *Journal of Discrete Algorithms*, pages 185–210, 2003.

[10] M. Ben-Or. Fast asynchronous byzantine agreement. In *Proceedings of the 4th Annual ACM Symposium on Principles Of Distributed Computing (PODC)*, pages 149–151, 1985.

[11] E. R. Berlekamp. *Algebraic Coding Theory*. McGraw-Hill, New York, NY, USA, 1968.

[12] J.P. Berrut and L.N. Trefethen. Barycentric Lagrange Interpolation. *SIAM Journal on Applied Mathematics*, pages 501–517, 2004.

[13] R. Bhagwan, S. Savage, and G. Voelker. Replication strategies for highly available peer-to-peer storage. In *Proceedings of the 3rd International Workshop on Future Directions in Distributing Computing (FuDiCo)*, pages 153–158, 2003.

[14] R. Bhagwan, S. Savage, and G.M. Voelker. Understanding availability. In *Proceedings of the 2nd International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 256–267, 2003.

[15] R. Bhagwan, K. Tati, Y.C. Cheng, S. Savage, and G.M. Voelker. Total recall: System support for automated availability management. In *Proceedings of the 1st USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 25–25, 2004.

[16] A.B. Bondi. Characteristics of scalability and their impact on performance. In *Proceedings of the 2nd ACM International Workshop on Software and Performance (WOSP)*, pages 195–203, 2000.

[17] G. Bracha. An asynchronous [(n-1)/3]-resilient consensus protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles Of Distributed Computing (PODC)*, pages 154–162, 1984.

[18] Y. Chawathe, S. Ratnasamy, L. Breslau, N. Lanham, and S. Shenker. Making gnutella-like P2P systems scalable. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 407–418, 2003.

[19] G.H. Chen, F. Wu, H.X. Li, and T.Q. Qiu. Redundancy schemes for high availability in DHTs. In *Proceedings of the 3rd IEEE International Symposium on Parallel and Distributed Processing and Applications (ISPA)*, pages 990–1000, 2005.

[20] R. Chien. Cyclic Decoding Procedures for Bose-Chaudhuri-Hocquenghem Codes. *IEEE Transactions on Information Theory*, pages 357–363, 1964.

[21] C.K.P. Clarke. Reed-solomon error correction. Technical Report WHP031, BBC Research and Development, 2002.

[22] M. Correia, N.F. Neves, and P. Veríssimo. From Consensus to Atomic Broadcast: Time-Free Byzantine-Resistant Protocols without Signatures. *The Computer Journal*, pages 82–96, 2006.

[23] M. Correia, G.S. Veronese, and L.C. Lung. Asynchronous byzantine consensus with 2f+1 processes. In *Proceedings of the 25th ACM Annual Symposium on Applied Computing (SAC)*, pages 475–480, 2010.

[24] M. Correia, G.S. Veronese, N.F. Neves, and P. Verissimo. Byzantine Consensus in Asynchronous Message-Passing Systems: A Survey. *International Journal of Critical Computer-Based Systems*, pages 141–161, 2011.

[25] F. Dabek, M.F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, pages 202–215, 2001.

[26] S. Decker, M. Schlosser, M. Sintek, and W. Nejdl. Hypercup: Hypercubes, ontologies and efficient search on P2P networks. In *Proceedings of the 1st International Conference on Agents and Peer-to-Peer Computing (AP2PC)*, pages 112–124, 2003.

[27] Flavio DePaoli and Leonardo Mariani. Dependability in Peer-to-Peer Systems. *IEEE Journal of Internet Computing*, pages 54–61, 2004.

[28] A.G. Dimakis, V. Prabhakaran, and K. Ramchandran. Decentralized Erasure Codes for Distributed Networked Storage. *IEEE/ACM Transactions on Networking*, pages 2809–2816, 2006.

[29] P. Druschel and A. Rowstron. Past: A large-scale, persistent peer-to-peer storage utility. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS)*, pages 75–80, 2001.

[30] M. Gradinariu M. Roy E. Anceaume, X. Defago. Toward a theory of self-organization. In *Proceedings of the 9th international conference on Principles of Distributed Systems (OPODIS)*, pages 191–205, 2005.

[31] P. Elias. Coding for Noisy Channels. pages 37–46, 1955.

[32] S.V. Fedorenko. A Simple Algorithm for Decoding Reed-Solomon Codes and its Relation to the Welch-Berlekamp Algorithm. *IEEE Transactions on Information Theory*, pages 1196–1198, 2005.

[33] W. Fenner et al. RFC 2236, IGMP - Internet Group Management Protocol, 1997.

[34] A. Fiat, J. Saia, and M. Young. Making chord robust to byzantine attacks. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA)*, pages 803–814, 2005.

[35] M.J. Fischer, N.A. Lynch, and M.S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the ACM*, pages 374–382, 1985.

[36] K. Fu, M.F. Kaashoek, and D. Mazieres. Fast and secure distributed read-only file system. *ACM Transactions on Computer System*, pages 1–24, 2002.

[37] R. G. Gallager. *Low Density Parity-Check Codes*. PhD thesis, Massachusetts Institute of Technology (MIT), Cambridge, MA, 1963.

[38] S. Gao. A New Algorithm for Decoding Reed-Solomon Codes. *Journal of Communications, Information and Network Security*, pages 55–68, 2003.

[39] P. Godfrey, S. Shenker, and I. Stoica. Minimizing churn in distributed systems. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 147–158, 2006.

[40] R.W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, pages 147–160, 1950.

[41] M.D. Hill. What is Scalability? *ACM SIGARCH Computer Architecture News*, pages 18–21, 1990.

[42] R.A. Horn and C.R. Johnson. *Topics in Matrix Analysis*. Cambridge University Press, 1994.

[43] David Reinsel John F. Gantz. The digital universe decade, are you ready? Technical Report Online, International Data Corporation (IDC) & EMC Corporation, 2011.

[44] V. Kalogeraki, D. Gunopulos, and D. Zeinalipour-Yazti. A local search mechanism for peer-to-peer networks. In *Proceedings of the 11th International Conference on Information and Knowledge Management (CIKM)*, pages 300–307, 2002.

[45] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web. In *Proceedings of the 29th annual ACM symposium on Theory of computing (STOC)*, pages 654–663, 1997.

[46] R. Koetter and A. Vardy. Algebraic soft decoding of reed-solomon codes, October 14 2003. US Patent 6,634,007.

[47] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, C. Wells, et al. OceanStore: an Architecture for Global-Scale Persistent Storage. *ACM SIGARCH Computer Architecture News*, pages 190–201, 2000.

[48] L. Lamport. Constructing digital signatures from a one-way function. Technical Report CSL-98, SRI International, 1979.

[49] L. Lamport. Password Authentication with Insecure Communication. *Journal of ACM Communications*, pages 770–772, 1981.

[50] L. Lamport, R. Shostak, and M. Pease. The Byzantine Generals Problem. *ACM Transactions on Programming Languages and Systems*, pages 382–401, 1982.

[51] S. Lang. Algebraic Groups over Finite Fields. *American Journal of Mathematics*, pages 555–563, 1956.

[52] J.C. Laprie. Dependable computing and fault tolerance: Concepts and terminology. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (ISFTC)*, pages 2–13, 1995.

[53] D.H. Lehmer. Euclid's algorithm for large numbers. *American Mathematical Monthly*, pages 227–233, 1938.

[54] H. LI and G. Chen. Data Persistence in Structured P2P Networks with Redundancy Schemes. In *Proceedings of the 6th IEEE International Conference on Grid and Cooperative Computing (GCC)*, pages 542–549, 2007.

[55] R. Lidl and H. Niederreiter. *Finite Fields and their Applications*. Cambridge University Press, 1996.

[56] E.K. Lua, J. Crowcroft, M. Pias, and R. Sharma. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Surveys & Tutorials*, pages 72–93, 2005.

[57] M. Luby. LT codes. In *Proceedings of the 43rd IEEE International Symposium on Foundations of Computer Science (FOCS)*, pages 271–286, 2002.

[58] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker. Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th International Conference on Supercomputing (ICS)*, pages 84–95, 2002.

[59] L.T.D.P.W. Madhu and S.K. Sudan. Coding theory: Tutorial & Survey. *SIAM Journal on Computing*, pages 1863–1920, 2001.

[60] D. Mandelbaum. On Decoding of Reed-Solomon Codes. *IEEE Transactions on Information Theory*, pages 707–712, 2002.

[61] P. Maymounkov. Online codes. Technical Report TR2002-833, New York University, 2002.

[62] P. Maymounkov and D. Mazieres. Kademlia: A peer-to-peer information system based on the xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[63] D. Mazieres. *Self-certifying File System*. PhD thesis, Massachusetts Institute of Technology (MIT), 2000.

[64] R. Merkle. Protocols for public key cryptosystems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 122–134, 1980.

[65] Napster. http://www.napster.com, 2001.

[66] D.A. Patterson, G. Gibson, and R.H. Katz. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[67] J.S. Plank and M.G. Thomason. A practical analysis of low-density parity-check erasure codes for wide-area storage applications. In *Proceedings of the 34th IEEE International Conference on Dependable Systems and Networks (DSN)*, pages 115–130, 2004.

[68] C.G. Plaxton, R. Rajaraman, and A.W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pages 241–280, 1997.

[69] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM Computer Communication Review*, pages 161–172, 2001.

[70] I.S. Reed and G. Solomon. Polynomial Codes Over Certain Finite Fields. *SIAM Journal of Applied Mathematics*, pages 300–304, 1960.

[71] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: The oceanstore prototype. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[72] H. Ribeiro and E. Anceaume. A comparative study of rateless codes for P2P persistent storage. In *Proceedings of the 12th International Conference on Stabilization, Safety, and Security of Distributed Systems(SSS)*, pages 489–503, 2010.

[73] H.B. Ribeiro and E. Anceaume. Datacube: a P2P persistent storage architecture based on hybrid redundancy schema. In *Proceedings of the 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing (PDP)*, pages 302–306, 2010.

[74] H.B. Ribeiro and E. Anceaume. Exploiting rateless coding in structured overlays to achieve data persistence. In *Proceedings of the 24th IEEE International Conference on Advanced Information Networking and Applications (AINA)*, pages 1165–1172, 2010.

[75] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems (IPTPS)*, 2005.

[76] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. *ACM SIGOPS Operating System Review*, pages 188–201, 2001.

[77] J.S. Russell. Report on waves. In *Proceedings of the 14th Meeting of the British Association for the Advancement of Science (BAAS)*, pages 311–390, 1844.

[78] Matthew S. Ryan and Graham R. Nudd. The viterbi algorithm. Technical Report FCS-RR-238, University of Warwick, Coventry, UK, 1993.

[79] Andrew S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[80] Y. Saad and M. Schultz. Topological properties of hypercubes. *IEEE Transactions on Computers*, pages 867–872, 1988.

[81] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Proceedings of the 1st IEEE International Conference on Peer-to-Peer Computing (P2P)*, pages 101–102, 2002.

[82] C.E. Shannon. A Mathematical Theory of Communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, pages 3–55, 2001.

[83] C. Shikey. *What is P2P...And What Isn't?* O'Reilly Network, 2000.

[84] A. Shokrollahi. Raptor codes. *IEEE/ACM Transactions on Networking*, pages 2551–2567, 2006.

[85] R.C. Singleton. Maximum distance q-nary codes. *IEEE Transactions on Information Theory*, pages 116–118, 1964.

[86] S.H. Standard. FIPS Pub 180-1. *National Institute of Standards and Technology*, 1995.

[87] R. Steinmetz and K. Wehrle. Peer-to-Peer-Networking &-Computing. *Journal Informatik-Spektrum*, pages 51–54, 2004.

[88] I. Stoica, D. Liben-Nowell, R. Morris, D. Karger, F. Dabek, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 149–160, 2001.

[89] M. Sudan. Coding Theory: Tutorial and Survey. *IEEE Foundations of Computer Science*, page 36, 2001.

[90] R. Tanner. A Recursive Approach to Low Complexity Codes. *IEEE Transactions on Information Theory*, pages 533–547, 1981.

[91] F. Vanhaverbeke, F. Simoens, M. Moeneclaey, and D. De Vleeschauwer. Binary erasure codes for packet transmission subject to correlated erasures. In *Proceedings of the 7th Pacific Rim conference on Advances in Multimedia Information Processing (PCM)*, pages 48–55, 2006.

[92] H. Weatherspoon, B.G. Chun, C.W. So, and J. Kubiatowicz. Long-term data maintenance in wide-area storage systems: A quantitative approach. Technical Report CSD-05-1404, University of California, Berkeley (Computer Science Division), 2005.

[93] H. Weatherspoon and J. Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, pages 328–337, 2002.

[94] M.H. Weik. *Communications Standard Dictionary*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1998.

[95] Z. Zhang and Q. Lian. Reperasure: Replication protocol using erasure-code in peer-to-peer storage network. In *Proceedings of the 21st IEEE Symposium on Reliable Distributed Systems (SRDS)*, pages 330–339, 2002.

[96] B.Y. Zhao, J. Kubiatowicz, and A.D. Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report FCSD-01-1141, University of California at Berkeley, 2001.

[97] X. Zhu, D. Zhang, W. Li, and K. Huang. A prediction-based fair replication algorithm in structured P2P systems. In *Proceedings of the 4th International Conference on Autonomic and Trusted Computing (ATC)*, pages 499–508, 2007.

Appendix A

Résumé Étendu en Français : Conception d'une Plate-forme "Pair-à-Pair" de Stockage de Données

Contents

A.1 Introduction

L'explosion exponentielle de la bande passante, de la capacité de stockage et des ressources de calculs offertes par les nœuds des systèmes pair-à-pair explique l'apparition récente de plate-formes de stockage pair-à-pair. Ces plate-formes (*e.g.*, CFS [25], Reperasure [95], Total Recall [15], OceanStore [47]) cherchent à exploiter les ressources offertes par les systèmes pair-à-pair pour fournir un service de stockage performant et sûr en termes de disponibilité, fiabilité et d'intégrité sans recourir à des serveurs centralisés. Garantir de telles propriétés dans un environnement pair-à-pair nécessite de prendre en compte à la fois un fort

dynamisme des nœuds (connexion et deconnexion à tout moment avec une durée de connexion pouvant être relativement courte) mais également des comportements malveillants. Dans le premier cas, il est assez naturel de considérer que ces évéments sont uniformément répartis spatialement et temporellement alors que dans le second cas, il n'est pas concevable de supposer que des attaques sont aléatoires. Au contraire, elles sont souvent coordonnées pour augmenter leur impact sur le système. La proposition de mécanismes permettant la construction d'une plate-forme de stockage de données appropriée à ce contexte d'exécution est l'objet de cette thèse.

Plus précisemment, au cours de cette thèse nous avons proposé une plate-forme de stockage pair-à-pair, appelée Datacube, fondée sur les principes suivants. Datacube organise ses nœuds de stockage autour d'une topologie structurée de type hypercube. Chaque sommet de l'hypercube est un groupe (*cluster*) de nœuds auto-organisés en *noyau* et en *spares* permettant l'exécution de protocoles de maintenance, et de routage robustes (protocoles d'accord résistants aux comportements byzantins). Ces protocoles exécutés sur les nœuds du noyau du cluster concerné permettent de s'adapter à la dynamicité de l'environnement.

Nous avons proposé une technique de redondance hybride reposant sur une combinaison de réplication et de codage à effacement sans rendement (*i.e.*, *codage fontaine*) en tirant profit des propriétés de structuration de cette architecture clusterisée. En l'absence d'attaques massives du système de stockage, Datacube s'appuie sur la réplication des données au sein du noyau du cluster ce qui assure une latence optimale de récupération des données. Par contre sur détection de manipulations massives du système, leur recouvrement est effectué à partir des blocs de données codées qui sont stockés sur les spares des autres clusters de la plate-forme. Un mécanisme de vérification d'intégrité basé sur l'empreinte des blocs assure que seuls les blocs non corrompus sont collectés. Cette redondance hybride garantit donc l'intégrité et la disponibilité des données à tout instant.

Nous avons implémenté deux algorithmes de codage fontaine [57, 61] pour évaluer les performances des opérations de codage et de décodage en nombre d'opérations élémentaires, en nombre de blocs à collecter, et en coût de maintenance pour assurer le taux de disponibilité souhaité. Ces expérimentations ont été faites dans des environnements agressifs, *i.e.*, en présence d'une forte densité de nœuds malveillants et de fréquence élevée d'entrées/sorties des nœuds. Nous avons également comparé ces deux algorithmes de codage fontaine avec les résultats théoriques de quasi-optimalité existants en faisant varier les différents paramètres de ces protocoles. Ceci nous a permis de mettre en évidence le mauvais comportement d'un des deux algorithmes dans certaines conditions d'exécution.

Tous ces travaux ont donné lieu à des publications dans des conférences internationales [73, 74, 72].

La suite de ce résumé présente brièvement les principes et les performances de Datacube et est organisée comme suit. Nous présentons au paragraphe A.2 les définitions et la terminologie nécessaires à la compréhension du reste du résumé. Le paragraphe A.3 présente les principes du codage fontaine. Au paragraphe A.4 nous présentons les architectures logiques des réseaux pair-à-pair, suivie par la présentation de Datacube au paragraphe A.5.

A.2 Stockage Persistent Pair-à-Pair : Définitions, Terminologie

Dans ce paragraphe, nous présentons les notions essentielles sur lesquelles notre travail de thèse repose.

A.2.1 Définitions et Terminologie

Les propriétés qu'un système de stockage doit assurer sont les suivantes :

Disponibilité : La disponibilité (*Availability*) est l'aptitude d'une entité à être prête à l'utilisation

Integrité : L'intègrité (*Integrity*) d'une entité est sa non altération par une entité extérieure

Maintenabilité : La maintenabilité (*Maintenability*) est l'aptitude d'une entité à être maintenue ou remise en état de fonctionnement.

Le terme entité est suffisamment générique pour représenter une donnée, un nœud, ou la plate-forme dans son ensemble, ce qui permet de décliner ces propriétés à différents niveaux de granularité. Par contre l'obtention de ces propriétés nécessite des techniques adaptées à chacun des niveaux de granularité considérés. Par exemple, et comme nous le verrons dans la suite de ce document, les techniques se prémunissant des attaques contre l'intégrité d'une donnée, où généralement l'attaquant fait en sorte que l'altération de la donnée ne soit pas visible de l'utilisateur (la fausse donnée apparait comme authentique), reposent sur l'utilisation d'une empreinte de la donnée, ou encore sur la transmission d'un MAC (*Message Authentication Code*) de la donnée. Par contre l'intégrité de la plate-forme repose sur des mécanismes de maintenance appropriés à sa structure, comme par exemple la cohérence des tables de routage des nœuds, afin de maintenir la connectivité du graphe d'interconnexion des nœuds.

La disponibilité des données est quant à elle garantie via l'utilisation de mécanismes de gestion de redondance. Il en existe deux grandes familles : la réplication (*e.g.*, [69, 29]) et le codage [47]. Soit d une donnée quelconque,

Réplication : La réplication de la donnée d consiste à créer $\ell > 1$ copies identiques de d et de les placer sur ℓ nœuds physiquement distincts. La disponibilité de d repose sur l'accessibilité d'au moins un nœud. Soit ε, la probabilité que d soit disponible (ε est typiquement de l'ordre de $0.99, 0.999, 0.9999$). Nous avons

- $\Pr\{d \text{ est disponible}\} = \varepsilon = 1 - (1-a)^{\ell}$, où a est la probabilité espérée qu'un nœud soit accessible
- Facteur de réplication = $\ell = \frac{\log(1-\varepsilon)}{\log(1-a)}$

Codage : Le codage de données consiste à introduire de la redondance dans la donnée elle-même. La donnée d est divisée en $k \geq 1$ blocs élémentaires de même taille et le processus de codage consiste à les combiner pour générer $n > k$ blocs codés incorporant suffisant de redondance pour qu'une fraction quelconque de k blocs parmi n soit suffisante pour que le processus de décodage puisse regénérer d. La probabilité ε que d soit disponible est exprimée comme suit

- Pr{d est disponible} = $\varepsilon = \sum_{i=k}^{\ell k} \binom{\ell k}{i} a^i (1-a)^{\ell k - i}$
- Rendement du codage = $\ell = \frac{n}{k} = \left(\frac{\sigma \sqrt{\frac{a(1-a)}{k}} + \sqrt{\frac{\sigma^2 a(1-a)}{k} + 4a}}{2a} \right)^2$

  ou σ représente l'écart type d'une loi normale (approximation d'une loi binomiale pour ℓk grand). (On a $\sigma = 3.7$ pour $\varepsilon = 0.9999$.)

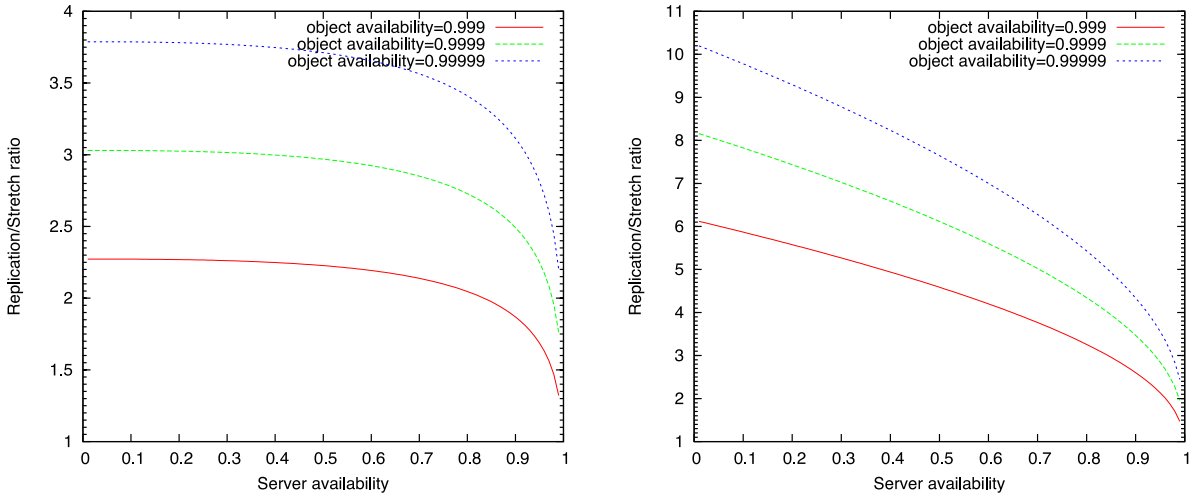


Figure A.1: Ratio entre le facteur de réplication et le rendement du codage en fonction de la disponibilité des nœuds et pour différents niveaux désirés de disponibilité de la donnée. À gauche, $k = 10$ et à droite $k = 1000$.

Le Figure A.1 représente le rapport entre le facteur de réplication et le rendement du codage en fonction de la disponibilité des nœuds. Elle montre d'une part l'avantage du codage sur la réplication lorsque les nœuds exhibent une faible disponibilité (ce qui correspond parfaitement aux caractéristiques des systèmes pair-à-pair). Elle montre également l'influence de la taille des données sur le rendement du codage. Plus cette taille est grande, plus il est intéressant de recourir au codage pour diminuer la quantité de redondance nécessaire pour garantir une disponibilité souhaitée.

Par rapport à la réplication, ces avantages sont cependant obtenus au détriment d'une perte de simplicité d'implémentation et d'une augmentation de la latence d'accès aux données. Ces constats sont à l'origine de notre proposition de redondance hybride combinant les avantages des deux techniques tout en limitant leurs points faibles.

Correct/Malveillant Un nœud est dit correct lorsqu'il suit les spécifications du protocole du système. Un nœud est dit malveillant lorsqu'il tente de perturber le bon fonctionnement du système. Pour ce faire, il peut utiliser toutes les ressources dont il dispose. Par exemple, il peut lancer des attaques contre l'intégrité des données ou du système en modifiant le contenu des données qu'il stocke localement ou en les remplaçant par des données indésirables, ou bien manipuler les tables de routage en substituant les nœuds honnêtes par d'autres nœuds malveillants. A l'extrême, et afin de causer le maximum de dommages, les nœuds malveillants peuvent coordonner leurs actions et former des coalitions.

A.3 Codes à Effacement Fontaine

Une transmission sur Internet peut être modélisée par un canal à effacement binaire sur lequel les symboles (*i.e.* les données) transmis sont soit reçus soit perdus. Pour que la transmission soit réussie il est nécessaire soit de retransmettre les symboles perdus soit d'introduire de la redondance dans l'information pour recouvrer les symboles perdus. Avec la première approche il est difficile d'atteindre la capacité du canal alors que la seconde approche, via des codages à effacement appropriés, le permet. Le codage à effacement consiste à transformer les k symboles d'information en un ensemble de n symboles codés. Le rapport $R = n/k$, appelé rendement du code ou taux de codage, spécifie la fraction des symboles codés nécessaires pour reconstruire l'information initiale. En fonction de la valeur de ce rapport, les codes à effacement sont classifiés en deux grandes familles : les codes à effacement à taux fixe (*fixed-rate coding*) [15, 47] et les codes à effacement sans rendement (*rateless coding*) [57, 61, 84]. Dans le premier cas, il existe des algorithmes de codage optimaux, les codes MDS (pour *Maximum Distance Separable*), qui permettent le décodage de l'information initiale dès que k symboles codés ont été reçus. De tels codes atteignent la capacité $1 - p$ du canal à effacement où p représente le taux de perte du canal. Les codes Reed-Solomon sont des codes MDS. Etant donné le coût de codage et de décodage des codes MDS qui peut devenir prohibitif lorsque k est grand, les codes LPDC (pour *Low Density Parity Check*) ont été proposés. Ces codes sont quasi-optimaux en nombre de symboles codés nécessaire pour reconstruire l'information initiale, *i.e.*, $n = (1 + \varepsilon)k$ pour tout $\varepsilon > 0$. Ces codes sont construits à partir d'une matrice de parité de faible densité ce qui leur permet d'exhiber une complexité linéaire à l'encodage et au décodage. Les codes Tornado en sont un exemple. Malheureusement l'utilisation des codes à taux fixe reposent sur la connaissance du taux de perte du canal pour déterminer le nombre de symboles à générer, ce qui n'est intrinsèquement pas adapté aux transmissions multipoints car cela nécessite de se caler sur le taux le plus élévé.

Les codes à effacement sans rendement, ou codes fontaine, ont été proposés récemment pour palier ce problème en garantissant des transmissions bi et multipoints efficaces sur des canaux à effacement. Contrairement aux codes à taux fixe, les codes fontaine ne reposent pas sur la connaissance du taux de perte du canal. Au contraire, les symboles de parité générés par combinaison linéaires des blocs initiaux peuvent être émis en continu ce qui explique le terme de *code fontaine*. De plus, le codage fontaine génère des blocs de parité indépendants les uns des autres ce qui leur confère un grand intérêt dans des environnement très dynamiques où les nœuds ne peuvent pas entamer de procédure de réconciliation pour éliminer les redondances des blocs de parité. Enfin, il a été montré que les codes fontaine atteignent asymptotiquement la capacité du canal. Il existe deux grandes classes de codes fontaine. Le représentant de la première classe étant le code LT proposé par Luby [57] et ceux de la seconde classe étant les codes online et raptor respectivement proposés par Maymounkov [61] et Shokrollahi [84]. La seconde classe diffère de la première par la présence d'une phase de pré-codage. Les principes de base de codage et décodage des codes fontaine sont les suivants.

A.3.1 Processus de Codage

Le processus de codage consiste à partitionner la donnée en k blocs de même taille. La génération des blocs codés est effectuée *(i)* en choisissant un degré d_i à partir d'une distribution particulière décrite ci-dessous, *(ii)* en choisissant aléatoirement d_i blocs (ces d_i blocs sont appelés les blocs voisins de d_i) parmi les k blocs initiaux, et *(iii)* en combinant les d_i blocs en un bloc codé c_i en les additionnant bit-à-bit via une opération OU EXCLUSIF (XOR).

Le degré ainsi que les blocs voisins de d_i sont nécessaires au processus de décodage. Il y a différentes solutions pour communiquer cette information. Par exemple, une fonction déterministe connue à la fois du codeur et du décodeur peut être utilisée, ou bien ces informations peuvent être directement mis dans l'entête du bloc codé. Dans notre solution, nous avons choisi la seconde solution pour limiter le nombre d'informations à conserver chez les nœuds. Le bloc codé cb_i est donc représenté par le couple $\langle c_i, x_i \rangle$, où x_i est l'ensemble des d_i voisins et c_i est le résultat de l'opération XOR sur l'ensemble des d_i voisins. Le processus de codage peut très simplement être modélisé par un graphe bipartite de Tanner [90], où le premier ensemble de sommets du graphe représente les blocs $k_1, k_2, \ldots$ initiaux et le second ensemble de sommets représente les blocs codés $cb_1, cb_2, \ldots$.

A.3.2 Processus de Décodage

Le processus de décodage consiste à reconstruire le graphe de Tanner sur les blocs codés récupérés lors du recouvrement de la donnée. Sur réception d'un bloc codé, le décodeur execute itérativement les étapes suivantes : *(i)* Trouver un bloc codé cb_i de degré un, *(ii)* affecter le bloc codé c_i de $cb_i\langle c_i, x_i \rangle$ au bloc initial k_i. (*i.e.*, le voisin k_i du bloc codé cb_i est une copie exacte de c_i), *(iii)* effacer les liens entre cb_i and k_i, et *(iv)* effectuer un XOR entre k_i et tous les blocs codés cb_r qui ont k_i comme voisin (*i.e.*, $cb_r = cb_r \oplus k_i$), et effacer les liens entre cb_r et k_i.

Ces 4 étapes sont répétées jusqu'à ce que tous les blocs initiaux aient été recouvrés. Il est clair que la réussite du processus de décodage repose sur la distribution de degrés utilisée lors du codage. Le codage LT repose sur une extension de la distribution de Soliton [77]. La motivation d'utiliser une telle distribution est qu'à chaque itération du processus de décodage, le nombre moyen de blocs codés de degré un est précisemment égal à un. Ainsi, si $\rho(d)$ décrit la probabilité de générer un bloc codé de degré d ($1 \leq d \leq k$), nous avons

$$\rho(d) = \begin{cases} \frac{1}{k} & \text{if } d = 1 \\ \frac{1}{d(d-1)} & \text{if } d = 2, \ldots, k \end{cases} \tag{A.1}$$

Malheureusement, la distribution de Soliton $\rho(.)$ ne se comporte bien qu'en moyenne, et donc en pratique de faibles variations sur la probabilité d'obtenir un bloc de degré un à chaque itération rend caduque le processus de décodage. L'extension de cette distribution par l'ajout d'une seconde fonction $\mu(d)$ pallie ce problème. Succintement, un plus grand nombre S de blocs codés de degré un sont générés ce qui permet de garantir le succès du processus de décodage avec probabilité $1 - \delta$. Spécifiquement, la fonction $\mu(d)$ est basée sur trois paramètres, k, δ et C, où k représente le nombre de blocs initiaux, δ la probabilité d'échec du décodage et C un entier positif affectant le nombre S de blocs de degré un. La distribution

de Soliton robuste est la valeur normalisée de $\rho(d) + \tau(d)$ avec $\tau(d)$ définie comme suit:

$$\tau(d) = \begin{cases} \frac{S}{k} \cdot \frac{1}{d} & \text{if } d = 1, \ldots, (\frac{k}{S}) - 1 \\ \frac{S \cdot \ln(\frac{S}{\delta})}{k} & \text{if } d = \frac{k}{S} \\ 0 & \text{if } d > \frac{k}{S} \end{cases} \tag{A.2}$$

où S, le nombre moyen de blocs codés de degré un, est donné par $S = C \ln(k/\delta)\sqrt{k}$. Il a été montré [57] que le nombre minimum moyen CB_0 de blocs codés nécessaires pour recouvrer la donnée initiale avec probabilité $1 - \delta$, avec δ petit est égal à

$$CB_0 = k \cdot \sum_{d=1}^{k} \rho(d) + \tau(d). \tag{A.3}$$

Le chapitre 3 de notre manuscrit de thèse décrit plus avant les principes du code LT ainsi que ceux du code online. Le chapitre 5 quant à lui compare leurs performances via des simulations dans Datacube. Ces simulations sont effectuées en fonction des paramètres des protocoles de codage et de décodage. Nous montrons ainsi les conditions d'utilisation optimales pour chacune de ces deux familles.

A.4 Architectures "Pair-à-Pair"

Les systèmes Pair-à-pair organisent leurs éléments (nœuds et données) au-dessus d'un réseau logique sous-jacent (*overlay*) construit au dessus d'une infrastructure physique. Les nœuds de cet overlay, communément appelés *pairs*, communiquent entre eux en utilisant les liens logiques de l'overlay via les primitives de communication fournies pas le réseau sous-jacent. La structure de l'overlay est caractérisée par l'organisation des pairs à l'intérieur de celui-ci. On distingue classiquement deux familles d'overlays. Lorsque les pairs s'organisent de manière aléatoire, on parle d'overlay non structuré ou mesh. Ce type d'overlay s'appuie sur un graphe aléatoire. De façon similaire aux pairs, les données sont placées aléatoirement sur ce graphe. Aussi, pour trouver une donnée ou un pair, il faut utiliser des techniques d'inondation maîtrisée, ou des marches aléatoires. À l'inverse, les overlays dits structurés s'organisent selon une topologie basée sur des graphes réguliers de type anneau [88], tore [69], ou hypercube [62] et d'une fonction de hachage (MD5, SHA-1, ...). Les nœuds entrant dans le systéme sont ainsi placés en fonction d'un identifiant qui leur est attribué. La fonction de hachage permet de placer équitablement et pseudo-aléatoirement les nœuds au sein du système, engendrant des topologies équilibrées. Ainsi chaque pair souhaitant entrer dans l'overlay, se voit attribuer un identifiant unique qui détermine sa position dans celui-ci, permettant de le contacter de manière efficace (classiquement en un nombre polylogarithmique de sauts en fonction de la taille de l'overlay). Le chapitre 2 de la thèse présente en détail les topologies pair-à-pair adaptées au stockage de données proposées dans la littérature ou utilisées à des fins commerciales. Dans ce court résumé, nous nous attardons sur la description des principes retenus pour Datacube.

A.5 Datacube : une Architecture de Stockage Distribuée Pair-à-Pair

Datacube s'appuie sur le réseau logique sous-jacent structuré Peercube [5]. Peercube est un overlay implémentant une stratégie de routage robuste. Cette stratégie se base sur un routage par chemins canoniques combinée avec un routage indépendant. L'implémentation des chemins canoniques est réalisée grâce à une technique de *clustering* des nœuds, alors que le routage indépendant découle des propriétés de l'hypercube. Au niveau local les nœuds, identifiés de manière unique, sont regroupés en clusters. Ces clusters sont identifiés par le prefixe commun des identités des noeuds le composant. La longueur de ce prefixe commun détermine la dimension de l'hypercube et varie en fonction du nombre de nœuds dans le système. Un cluster contient un *noyau* de pairs fortement connectés dont le rôle est de gérer les nœuds de ce cluster ainsi que la connectivité du cluster avec les autres de son voisinage. Les autres nœuds de ce cluster, appelés *spares*, ne maintiennent de liens qu'avec un sous-ensemble des nœuds du noyau. Tous les nœuds d'un cluster maintiennent les mêmes données, offrant ainsi un niveau de réplication élevé. La robustesse de la maintenance de la topologie face à des attaques lors des arrivées et départs des nœuds est garantie par le renouvellement aléatoire du noyau. Ces renouvellements sont faits au travers de protocoles de consensus tolérants aux comportements Byzantins au sein du noyau. Au niveau de la topologie globale, les clusters sont interconnectés suivant une topologie en hypercube. Outre sa scalabilité, cette topologie permet de router les messages sur des chemins indépendants, permettant ainsi de garantir la robustesse du routage.

Cependant, des attaques ciblées peuvent rapidement biaiser la répartition uniforme des nœuds et engendrer l'écroulement du système. En effet, lorsque les attaques sont orchestrées et centrées sur une même région de l'overlay, l'adversaire multiplie ses ressources pour polluer des clusters ciblés. Une stratégie d'attaque possible pour parvenir à une telle situation est de maximiser le nombre de nœuds malveillants dans le système et d'encercler progressivement les nœuds corrects pour aboutir à leur isolement. Une fois l'isolement atteint, l'adversaire peut librement empêcher les nœuds corrects de rentrer dans les clusters corrompus, et ainsi manipuler les données qu'il contient à loisir, violant leur intégrité et/ou leur accessibilité.

Dans Datacube nous proposons de faire face à ce problème en recourant à une gestion de la redondance hybride consistant à combiner la réplication et le codage des données. Brièvement, chaque donnée d est répliquée sur l'ensemble des nœuds du noyau du cluster $\mathcal{D}$ dont elle est le plus proche. La donnée d est également codée et chacun des blocs codés issus de l'algorithme de codage reçoit un identifiant unique (via une fonction de hashage). Les blocs codés sont disséminés sur les spares des clusters $\mathcal{D}_i, \mathcal{D}_j, \ldots$ les plus proches des identifiants des blocs. Les empreintes de ces blocs sont conservées par les nœuds du noyau de $\mathcal{D}_i, \mathcal{D}_j, \ldots$. Ainsi, en l'absence de corruption de $\mathcal{D}$, d est récupérée directement à partir des nœuds du noyau de $\mathcal{D}$ alors qu'en présence de pollution de son cluster, d est recouvrée en collectant un sous-ensemble des blocs codés sur les spares de $\mathcal{D}_i, \mathcal{D}_j, \ldots$

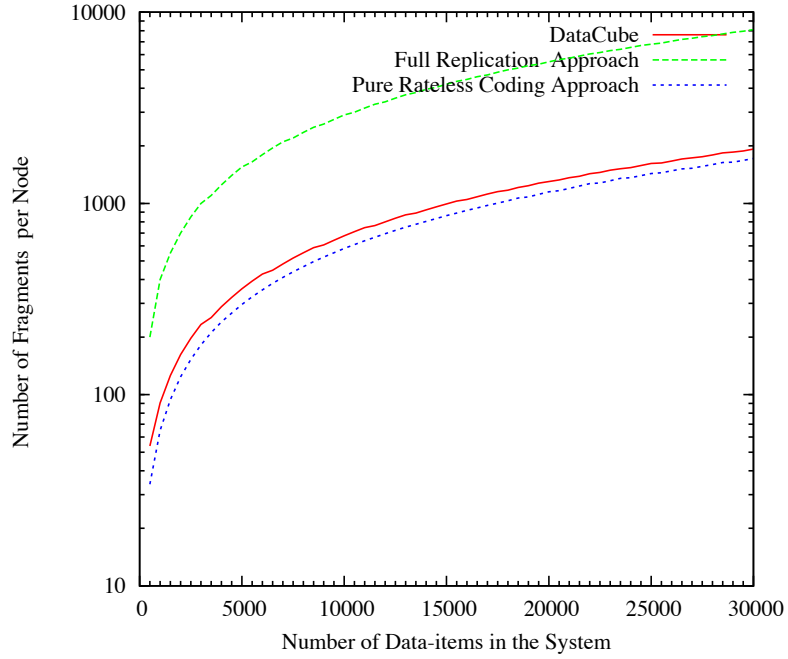
Quelques Résultats Nous terminons ce résumé par un extrait des résultats obtenus lors de cette thèse. Toutes les figures ci-après ont été obtenues dans un système dont la population varie entre 1000 et 10000 nœuds. L'environnement est fortement agressif. L'adversaire a

réussi à polluer au moins un cluster (*i.e.*, 45% de sa population est constituée de nœuds malveillants), et est présent à 30% dans le reste du système (*i.e.*, tous les autres clusters ont une population polluée à 30%). Chaque requête (*e.g.*, demande de stockage, lecture) est acheminée vers sa destination sur r chemins indépendants avec r satisfaisant $\log(N/S_{max}) \leq r \leq \log(N/S_{max}) + 3$. Le nombre de sauts moyen h sur chaque chemin satisfait la contrainte suivante $1 \leq h \leq \log(N/S_{max}) + 5$. Le tableau 1 donne les facteurs de réplication et les rendements du codage (On-line) en fonction de la disponibilité souhaitée des nœuds et ceci pour différentes valeurs de r et h. Les valeurs confirment nettement l'intérêt du codage par rapport à la réplication, et également l'impact du routage en terme de chemins redondants et du nombre de sauts sur la redondance à apporter au système pour garantir la disponibilité des données.

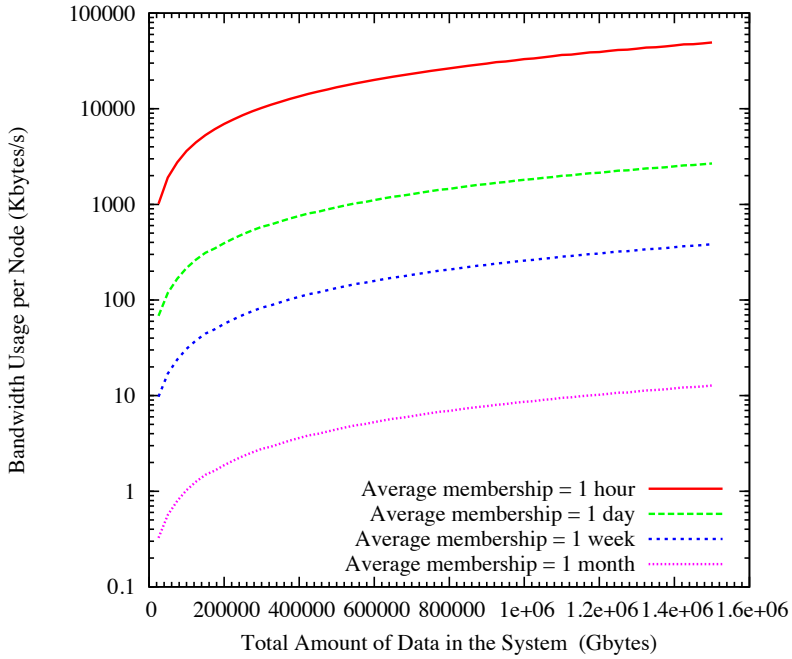
N	nombre de sauts	nombre de chemins	rendement codage		facteur de réplication	
			0.99	0.999	0.99	0.999
1,000	1	1	1.42	1.50	3	7
	5	1	3.66	3.96	11	16
	11	1	13.4	13.5	44	65
	1	9	1.02	1.02	1	1
	5	9	1.98	1.10	2	2
	11	9	1.98	2.12	5	8
5.000	1	1	1.42	1.50	3	5
	7	1	5.66	6.18	18	27
	14	1	25.00	27.5	83	125
	1	12	1.02	1.02	1	1
	7	12	1.12	1.16	2	3
	14	12	2.60	2.80	7	11
10.000	1	1	1.42	1.5	3	5
	8	1	7.02	7.68	22	34
	15	1	30.90	34.00	103	155
	1	13	1.02	1.02	1	1
	8	13	1.18	1.22	2	3
	15	13	2.88	3.12	8	12

Table A.1: Facteurs de réplication et rendements du codage (On-line) en fonction de la disponibilité souhaitée des nœuds et pour des valeurs extrêmes de r et h

La figure A.2(a) compare, pour une disponibilité des données de 0.999 et pour une population de 10000 nœuds, les performances en terme de stockage de l'approche de gestion de la redondance proposée dans Datacube avec les approches réplication et codage. Le facteur de réplication utilisé est de 34 et le rendement du codage est égal à 7.68 (*cf.* tableau 1). Les résultats démontrent clairement le gain du codage vis-à-vis de la réplication, ainsi que le faible impact de la réplication dans Datacube. La figure A.2(b) quant à elle montre l'impact en terme de bande passante pour chaque nœud pour maintenir le niveau de redondance souhaité en fonction du churn du système. Lorsqu'un nœud quitte le système, il doit envoyer les données et les blocs codés qu'il détient à ses nœuds voisins, et lorsqu'un nœud rejoint le système, ce nœud doit télécharger les données et blocs codés qui lui reviennent (par construction de l'overlay). Dans les expériences montrées sur cette figure, la quantité globale de données à maintenir (pour une disponibilité de 0.99) peut atteindre 1000 TeraBytes (ce qui correspond à une très grosse archive vidéo). Clairement, la quantité moyenne d'informations émise et reçue par nœud dépend de la fréquence avec laquelle les nœuds entrent et sortent du système. Pour des entrées/sorties journalières, la quantité d'informations qu'il est nécessaire de s'échanger est trop importante pour être gérable au niveau du nœud. Par contre si les nœuds joignent et quittent le système mensuellement, alors la quantité de



(a) Comparaison des performances de Datacube en terme de stockage par nœud en fonction de la quantité globale de données



(b) Quantité de données (en KBytes/s) à transférer par nœud en fonction de la quantité totale de données dans Datacube (GBytes)

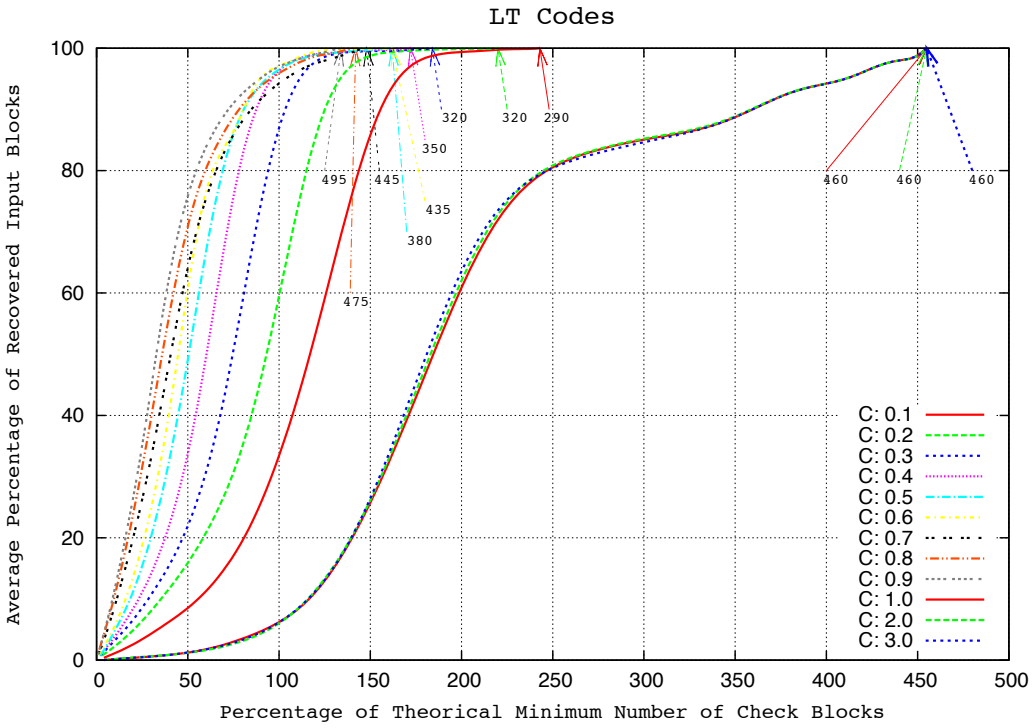
Figure A.2: Quelques résultats de Datacube

données à transférer est en moyenne de 20KBytes/s ce qui est complètement compatible avec des connexions de type ADLS.

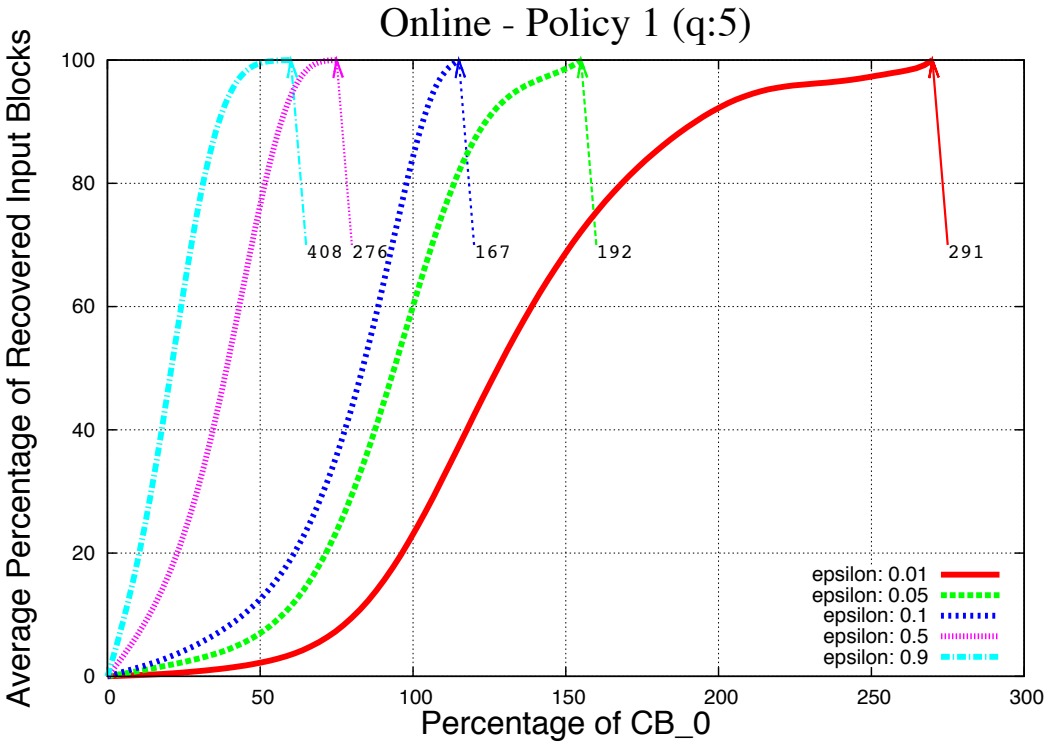
Les figures A.3(a) et A.3(b) montrent respectivement le comportement des codes LT et Online par rapport aux prédictions théoriques sur le nombre de blocs codés à collecter pour recouvrer avec succès les données initiales. Une remarque générale est la grande variabilité des comportements des codes par rapport aux prédictions théoriques en fonction de la valeur de leurs paramètres. Dans le cas de LT, on observe un écroulement des performances du décodage pour des valeurs de $C \geq 0.6$. Ceci correspond à la limite d'utilisation de la distribution soliton robuste (*cf.* paragraphe A.3). Dans le cas de Online, l'écroulement se produit pour de faibles valeurs de ϵ, paramètre influant la quantité de blocs générés lors de la phase de pré-codage. A l'inverse, les prédictions théoriques sont pessimistes pour certaines plages de valeurs des paramètres, et ceci pour les deux codes. D'autres résultats montrant les comportements de chacun des deux codes sont fournis dans le chapitre 5 de notre manuscrit.

A.6 Conclusions et Perspectives

Ces travaux de thèse nous ont permis de mettre en évidence que la construction d'une plateforme de stockage pair-à-pair basée sur une gestion de la redondance hybride permet de garantir la disponibilité et l'intégrité des données à tout instant dans un environnement hostile et fortement dynamique. Ce travail ouvre des perspectives de recherche nombreuses. En particulier, il serait intéressant d'étendre Datacube pour permettre à l'utilisateur de modifier ses données (extension aux opérations décriture) en tirant profit des propriétés du codage. En effet, dans l'état actuel de Datacube, modifier le contenu d'une donnée nécessite de stocker de nouveau cette donnée modifiée, ce qui revient à la considérer comme une nouvelle donnée. Si au contraire, il est possible de générer uniquement les blocs codés correspondant à la partie modifiée de la donnée, alors le coût de mise-à-jour en sera considérablement diminué.



(a) Fraction des blocs de données recouvrées par rapport aux prédictions théoriques pour un code LT en fonction du paramètre C. Les valeurs des abscisses et ordonnées sont en pourcentage.



(b) Fraction des blocs de données recouvrées par rapport aux prédictions théoriques pour un code Online en fonction du paramètre ϵ. Les valeurs des abscisses et ordonnées sont en pourcentage.

Figure A.3: Quelques résultats de Datacube