Aalborg University

Master Thesis

Content Delivery and Storage in Wireless Networks Using Network Coding

Author:
Juan Cabrera

Supervisor:
Dr. Daniel Lucani

A thesis submitted in fulfilment of the requirements for the degree of Master in Science

in the

Network Coding Focus Group
Electronic Systems

June 2015

Department of Electronic Systems
Fredrik Bajers Vej 7
DK-9220 Aalborg O
http://es.aau.dk

AALBORG UNIVERSITY
STUDENT REPORT

Title:
Content Delivery and Storage in Wireless Networks Using Network Coding

Theme:
Scientific Theme

Project Period:
Spring Semester 2015

Project Group:
15gr1054

Participant(s):
Juan Alberto Cabrera Guerrero

Supervisor(s):
Daniel E. Lucani. R

Copies: 2

Page Numbers: 69

Date of Completion:
June 2015

Abstract:

As an alternative to storing and delivering data from a single cloud server service providers might be interested in solutions that spread the data over multiple devices, particularly peer devices. The dynamics of this problem when introducing wireless mobile devices make the management of content and design of protocols extremely challenging. To address these challenges, this project shall use network coding as a common ground to manage the storage and communication among nodes with a single code structure. When a peer disconnects from the network, a repair of the redundancy lost has to be performed by the other nodes to guarantee that the information is reliable against future disconnections. The contributions include the evaluation of the state-of-the-art techniques for repair in standard distributed storage systems when applied to a limited set of peers that are maintaining a file reliable. Moreover, the project also explore these repair techniques when performed in wireless channels. By taking advantage of the broadcast nature of the channel it is shown that the number of transmissions for repairing can be reduced.

"Let us think the unthinkable, let us do the undoable, let us prepare to grapple with the ineffable itself, and see if we may not eff it after all."

Douglas Adams

Acknowledgements

I want to thank my parents for their unconditional support in every aspect of my life through all these years.

I also wish to express my sincere thanks to Dr. Daniel Lucani, my supervisor. I am extremely grateful for his constant help not only in the academic field but also in a personal dimension. I am thankful for his helpful e-mails and suggestions even before my moving to Denmark.

Contents

List of Figures

Acronyms

DHT Distributed Hash Tables.

FTP File Transfer Protocol.

IoT Internet of Things.

MBR Minimum Bandwith Repair.

MDS Maximum Distance Separable.

MSR Minimum Storage Repair.

P2P Peer to Peer.

PeX Peer Exchange.

RFC Request for Comments.

RLNC Random Linear Network Coding.

SSH Secure Shell.

TTL Time to Live.

WSN Wireless Sensor Network.

To my parents...

Chapter 1

Introduction

Nowadays the standard method for file transfers on the Internet is the client-server approach. A centralized system in which a server sends the files to all the nodes that request them. Service providers have to deal with a range of challenges when they are storing the information in a centralized manner. For instance, if the server storing the data fails, then the information becomes unavailable for the users. When a file becomes popular, the server might get blocked due to an increased number of requests to download, becoming unable to serve all the users or doing so with a degraded quality of service. This degradation includes slower data transmission rates and increased delays between the moment when the user makes the request and when it is served.

To increase reliability of the information at the server side, service providers rely on distributed storage system. By doing such spreading of the files, the system can guarantee that the information is reliable during long periods of time even though the individual nodes are unreliable. In this scenario, even if one node fails losing all the data that it was storing, the information can be recovered from the other nodes. To ensure this reliability, some redundancy is added. The simplest form of redundancy is replication, for example, duplicating the files in mirrors server or in caches close to the edge of the network. Storing the information close to the consumers might help allowing for greater availability of the information and potentially better delay performance since it is closer to the users requesting data. However, besides its high costs, data caching involves a complex problem when deciding what files or what pieces of files to cache. For distributed storage, some providers opt for cloud computing systems. For instance, Google stores information in *cells* or *storage cells* which consists in a shared

pool of machines with different applications and which size might be in the order of thousands of nodes, connected physically close one to another [1].

Nevertheless, speed may still be constrained due to high load and restrictions on the server side. The option of distributing the information in Peer to Peer (P2P) networks instead, allows the systems to aggregate data from multiple locations, even if each one of them is limited in capabilities and resources.

1.1 Peer to Peer Distributed Storage Systems

In practice, having a single distributed cloud storage system is not enough as a solution to guarantee that data is reliably stored, private and readily available with high data rates. A single cloud system is susceptible to failures, like those presented by Amazon during the last years that affected companies like Instagram, Netflix and Pinterest [2]. Besides, managing the data deep into the network in a centralized manner affects the scalability of the solutions when a bigger number of users with a higher demand of content joins the service. For that reason this project focuses on the storage and delivery of content spreading the data in a decentralized manner over multiple heterogeneous devices e.g. mobile phones, laptops, desktops, embedded systems.

The project looks particularly into these devices when they organize themselves in P2P networks with little to no intervention of a central entity. Moreover, in these types of systems, nodes continuously leave and reconnect to the network. In these systems each node is commonly seen as a peer, i.e. an entity equivalent to the others without any privilege. All the peers make a share of their resources available to the rest of the system and are at the same time consumers and providers of information. Since it is not centralized, it is a network with the potential to be scalable. In that sense, in a P2P distributed storage system, a mobile phone, for example, can store some of its files distributedly into all the other phones and computers members of the network. For that, the peer should make available part of its storage capacity for the others to use.

In the last two decades, many solutions that make use of P2P networks for file transferring have appeared. For example, Gnutella, Napster [3], and Kazaa [4] are file sharing protocols that allow the transfer of files directly among users. The aim of these services is to distribute complete files among nodes that constantly joins and leaves the network. The BitTorrent protocol

[5] is one of the most famous protocols nowadays and was designed for this purpose.

In the BitTorrent protocol the files are split into pieces. When a node wants to download a particular file, it contacts some peers that have pieces of it and requests them. This bring the complex problem to the system of deciding in what order to transfer the pieces, specially with less popular files. It might occur that at a certain point in time, all the nodes that have one particular piece are not available making impossible the recovering of the file. It could happen that a small number of nodes have one specific piece causing, as a consequence, a slow download speed [6] for the other peers. Or in the scenario of an on-demand streaming, nodes might be interested in downloading first the pieces that are going to be played soon [7].

The problem of the selection of pieces is also present in P2P networks where the nodes are not interested in getting the whole files but part of them in order to have redundancy in the system for reliability of the information. The case of the Wireless Sensor Networks(WSNs) present such scenario. An example would be the deployment of sensors (peers) in a large scale and in a remote area such that they are not connected to the Internet. The sensors individually are unreliable and have to collect and store data for long amounts of time, until a sink requests from them the recorded information. Taking advantage of distributed storage, the WSN as a whole has the capacity of storing the files even in the case where individual peers would surpass its own storage capacity. Besides, the storing of the data can be reliable against peers failing. Such scenarios are becoming more common with the advent of the Internet of Things (IoT). For instance [8] studies that scenario focusing on data dissemination schemes to duplicate information in WSNs for IoT observation systems to make the storage of the files resilient.

But the problem of where to store the duplicated pieces remains. If all the nodes with the same piece fail, then the whole file is unrecoverable. For that reason such system needs to keep track of what peers are storing what pieces of what files. Such tracking requires resources of time and energy which commonly are sparse in low power WSNs.

1.2 Erassure Coding and Network Coding for Storage

Erasure coding offers a better option for storage efficiency. Dividing a file of size M into k pieces of size M/k, an alternative to storing replicas of the fragments is to produce n coded pieces using an encoder and a MDS code (n, k), and store those n pieces instead. Then, any set of k pieces of size M/k is enough to recover the whole file, which makes the approach optimal in terms of redundancy-reliability tradeoff [9, 10]. This method shows much more reliability for the same amount of redundancy than simple replication [11]. Besides, the system should no longer keep track of where it stores the replicated pieces. Instead it should guarantee only that enough different pieces are available at all time.

One of the most famous MDS codes used for this purposes are the Reed-Solomon codes [12]. Companies such as Wuala, which in its origins offered a peer assisted file caching combined with centralized storage [13], and the now defunct research project OceanStore [14] use this type of codes for the redundancy in their data storage systems.

The use of traditional MDS codes brings new difficulties. When a node fails or a peer disconnects from the network, the system must repair the redundancy lost with the node. With replication, the piece lost is simply copied from other node in the network, without any repair overhead i.e. to repair k bits, only k bits are transmitted over the network. On the other hand, codes like Reed-Solomon first need to decode the whole file to be able to generate new coded pieces. This means that repairing a fragment of size M/k needs a repair bandwidth of at least M i.e. at least the whole file must be transferred over the network every time the system builds new redundancy.

Network coding appears as a solution to this difficulty. With this novel technique it is possible to generate erasure codes that allows repairing by transmitting the information theoretic minimum over the network [9]. The most common variant of network coding is Random Linear Network Coding (RLNC). This variant, when used to code files, takes the original k pieces of a file $x_1, x_2, x_3, ..., x_k$ and creates n linear combinations $p_1, p_2, p_3, ..., p_n$ of the same size called coded packets. Where p_j is

$$p_j = \sum_{i=1}^{k} c_{i,j} \cdot x_i \tag{1.1}$$

In equation 1.1 each $c_{i,j}$ is a coefficient chosen randomly from a finite field, commonly of the form $\mathbf{GF}(2^m)$. These coefficients are then appended to each coded packet. Similarly as in Reed-Solomon codes, any set of k linear independent packets are enough to decode the file. However, a novelty of network coding over Reed-Solomon codes is that it allows the recoding of packets already coded, i.e., it is possible to generate new coded packets without decoding the whole file. In that sense, given a set of coded packets $p_1, p_2, ..., p_l$, a new packet p' can be generated such as

$$p' = \sum_{i=1}^{l} c_i \cdot p_i \tag{1.2}$$

By distributing random combinations of the files among the peers and clouds services instead of just raw data, network coding offers an intrinsic level of security. The data is still private even when "malicious" peers are present in the network or when a cloud service gets compromised by external agents [15, 16]. An eavesdropper would need to compromise the whole system and gather enough coded packets in order to be able to decode and "understand" the data.

At the same time, network coding has proven benefits when used in different communication scenarios. In point-to-point communications, it allows the repair of packets losses in lossy channels. If there is an estimation of the packet error probability, the transmitter can send extra coded packets. Since it is not relevant to know what specific packets got lost, this does not require extra feedback from the receiver. In multicast scenarios over lossy wireless channels, when several nodes are interested in receiving the same data, if the transmitter broadcast uncoded packets, then it will need to retransmit every single one of the lost packets. Due to the uncorrelated losses, many of these retransmissions will be useful only for a few nodes. If coded packets are sent instead, the information contained in the retransmitted packets might benefit with high probability all the nodes that suffered from losses.

By using network coding in a distributed storage system to manage the storage and communications with a single code structure, this project takes advantages of the benefits of that technology in the field of storage [17] and wireless communications [18]. Which brings reliable multicast and increased data transmission rates in lossy channels.

1.3 Our Contributions

The goal of this project is to provide mathematical models to analyze the problem of storage and repair of redundancy in P2P networks. It studies the use of network coding as a common code to manage the storage and cooperation between peers at the edge of wireless networks. Specially when said peers constantly leave and rejoin the network. State of the Art in the field has considered the benefits of network coding for repairing redundancy [9] into new nodes, or into new clouds [19]. However, in this project we go beyond these controlled scenarios and focus on evaluating cases where there are no new available nodes or cloud services and the system must repair redundancy in the available nodes to prepare itself for the potential disconnections of more peers. At the same time, this project studies the case when the losses of peers or cloud services are temporary. When the node returns, the system must perform the removal of redundancy built during repair sessions.

Furthermore, we designed a protocol and a software application that was tested in real devices. This implementation is capable of maintaining files reliable when they are stored into a limited set of unreliable nodes. Even with disconnections of peers, the system can guarantee, after a repair session, that the data will be available against the potential disconnection of another node.

From the results of this project, we are preparing a paper to be submitted in July 1st 2015 for IEEE Globecom Workshops.

Chapter 2

State of the Art in Peer to Peer Networks

In a P2P network, opposed to a centralized network where nodes request resources to a server, each user is a client and a server for the other nodes. Each user or *peer* has the same status within the network. The different nodes share tasks like file storage, video streaming, etc. in such systems. For this to be possible, each peer makes part of its resources e.g. disk storage, processing power, network bandwidth directly available for the others nodes of the network without the need of a centralized coordination or any other entity [20].

2.1 Routing and Resources Discovery

The P2P networks are built as overlay networks over the underlying physical network. The data is still transferred through the underlying TCP/IP network, but at the application layer peers can communicate directly with each other. This is done through logical overlay links. Each one of these links correspond to a specific path through the underlying physical layer (which may involve several hops). Overlays at the same time are used for indexing and peer discovery.

According to how the nodes are linked within the overlay network and how the resources are indexed and discovered, the peer to peer networks may be classified in structured and unstructured networks.

2.1.1 Unstructured Peer to Peer Networks

An unstructured P2P network presents no particular overlay structure by design. Each time a node joins the network, it randomly forms connections with the others peers. This makes the network robust against *churn*, i.e., nodes constantly joining and leaving the system.

A problem with unstructured networks is that the lack of organization makes it complicated to find a file when a node requests it. If a peer performs a search query, the other peers must flood it into the network. This flooding is specially expensive in terms of network resources because it is done on an overlay network on top of the underlying physical network. There is a tremendous bandwidth consumption in flooding-based approaches [21]. A forwarded search query from a peer to another in reality is performing several hops in the physical network. The mismatch between the P2P network and the physical network produces zigzag routes [21] as described in [22]. Because of the high cost involved in flooded messages, the system typically limits their lifetime to a few hops. As a consequence there are no guarantees that the search query would be successful even if the file is present in the network.

2.1.2 Structured Peer to Peer Networks

In a structured peer to peer network, the nodes connect to each other in a specific topology. Since the network maintains a structured and organized topology, all the peers can cooperatively maintain routing information relevant for reaching all the other nodes in the overlay network [23]. When any peer performs a search query in the system for a specific file, due to the structured organization of the overlay, the protocol can guarantee that the search query will reach all the nodes. As a consequence, rare files stored only in a few nodes in the system can be found by the other peers.

Structured networks use distributed indexing structures such as Distributed Hash Tables (DHT) to find files. In these methods, a key is assigned to each file in the network. The keys are chosen from a finite keyspace, for example, the space of all the 160-bit long strings. These keys are generated with hash functions like the SHA-1 function [24]. Hash functions have the property of producing very different outputs with small changes in the input. Figure 2.1 illustrates how different data on the different peers in a network is indexed by giving it a hash key using a hash function.

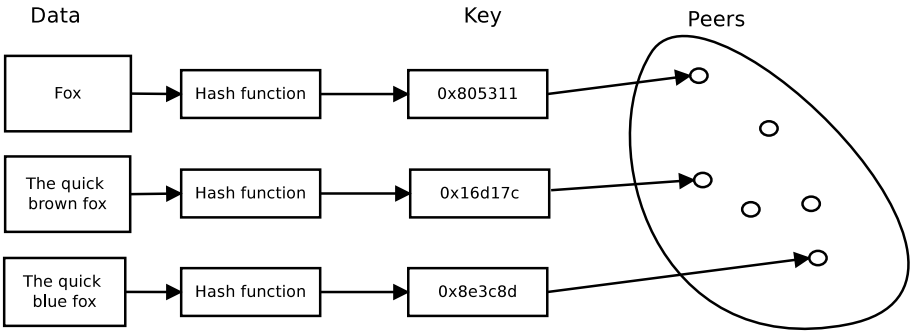

FIGURE 2.1: Distributed hash tables. Small variations in the input of a hash function results in big variations of the output

When a node wants to retrieve a particular file from the network, it sends a message $get(k)$ where k is, for example, the SHA-1 key. Then, a search is performed in the network and the nodes route the message using various methods, e.g., routing trees [25], finger tables [26]. Depending of the protocol used, this search may have a complexity as low as $O(\log(N))$ [21].

Maintaining routing information becomes a problem when nodes are constantly leaving and rejoining the network. Every time that such event occurs, the peers must update the routing tables.

2.2 File Sharing in Peer to Peer Unstructured Networks

In the client-server approach, like the File Transfer Protocol (FTP), there is a clear distinction on which node is serving the files and which node is consuming them. For that reason, finding where a file is stored is not that complicated, it is a matter of finding the server. On the other hand, the architecture of a peer to peer network differs from the classic approach since in these networks each node is an equal entity that functions as a server and a client at the same time. The files in these types of networks are distributed among several nodes. For that reason, a challenge of P2P systems is how to find a file that is stored somewhere in the network. This challenge is particularly difficult in unstructured networks and different solutions have been proposed in the last decades. If we want to distribute content in a P2P network, then it is important to study the state of the art solutions proposed for finding data in these systems.

2.2.1 Centralized Approach: Napster

In the centralized approach a server has information about all the files in the system. It contains information about what files holds and share each peer as well as meta-data information like the file size and file name.

In a centralized approach when a new peer joins the system it reports to the server what files it posses and share. Then, when a user needs to find a specific file in the network, it requests it to the server. If the file exists, the server sends a list of all the peers that have the file. Then the user contacts the returned peers with a request to download the required file. Figure 2.2 illustrates this search process.

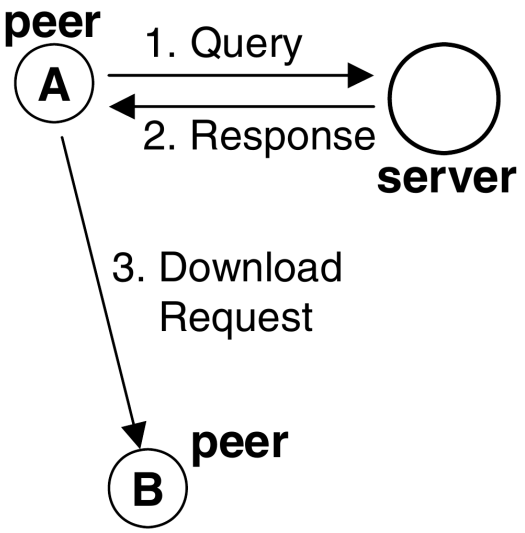

FIGURE 2.2: Search process in Napster. Image source: [23].

A centralized approach is easy to implement. The network administrator only has one server to take care of and maintain. However, it is not a scalable approach. When the number of peers increases, the server or centralized entity needs more storage capacity, bandwidth and processing power in order to process and serve all the search queries of the peers. A centralized approach is not robust to attacks or failures since there is one central point of failure that can bring the system down. If the server fails, then the system is disabled.

2.2.2 Distributed Approach: Gnutella

The Gnutella protocol does not require a central server, which leads to a decentralized, dynamic and self organized network. When a new peer joins the system, it broadcasts a PING message to all its neighbors. Each

neighbor responds with a PONG message containing its IP address, port and information of the files that it shares.

When a peer needs to find a specific file, it broadcasts a search query which propagates through the network based in flooding. Each peer that receives the search query checks if it posses the file. If the node have the whole file or part of it, then it sends a response back through the same path used by the flooded message.

To prevent a big number of search query messages in the network, these contain a Time to Live (TTL) which is a value that decrements by one at each node. However, choosing this value is not an easy task. If it is too big, then the query message might "live" too long time in the network affecting the performance of the system. If it is too small then it might lead to unsuccessful search queries with high probability even if the searched file exists somewhere in the network. In Figure 2.3 it can be seen the search mechanism used in this protocol.

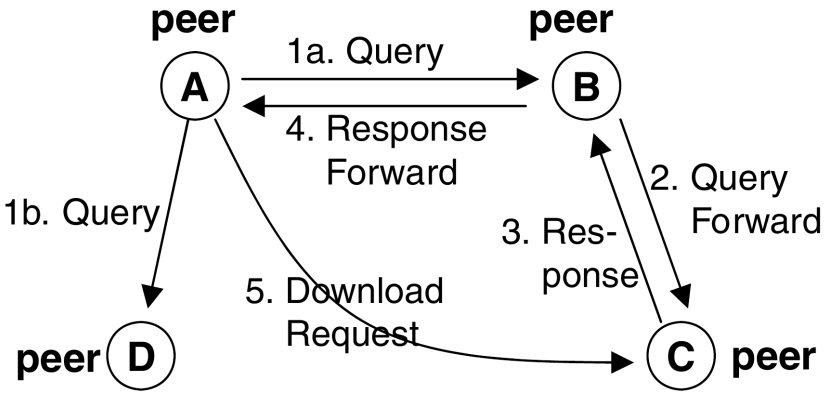

FIGURE 2.3: Search process in Gnutella. Image source: [23].

In a distributed approach each peer connects and communicates with a few nodes in its vicinity. This allows the system to support an unlimited number of peers as long as search efficiency is not a constraint. These systems are also robust to dynamics of the peers. If a peer leaves or joins the network, then it and its neighbors can connect to other peers easily by the exchange of PING and PONG messages. Nevertheless, a big disadvantage of this approach is that the search mechanism is inefficient and might result in unsuccessful outcomes even if a copy of the file exists somewhere in the system.

2.2.3 Hybrid Approach: Kazaa

Hybrid approaches try to combine the advantages of the centralized and distributed protocols. In an hybrid approach, some of the nodes, specifically those with more resources in terms of bandwidth and processing power are promoted to *super nodes*. These super nodes contain information of the files of some of the peers that are connected to it as well as information for contacting other super nodes. They can be seen as local entities similar to servers of a centralized P2P network.

In the Kazaa protocol, if a node not promoted to super node, wants to search for a file, it sends a search query to a super node. The super node then works as the server in a centralized approach and responds with a list of peers containing the file. If it does not have that list, then it performs a search query by flooding, similar to the search queries performed in a distributed approach, but only among other super nodes. When the regular node receives the list of peers that have the file, then it starts the download from them. Figure 2.4 illustrates the search mechanism used in the Kazaa protocol.

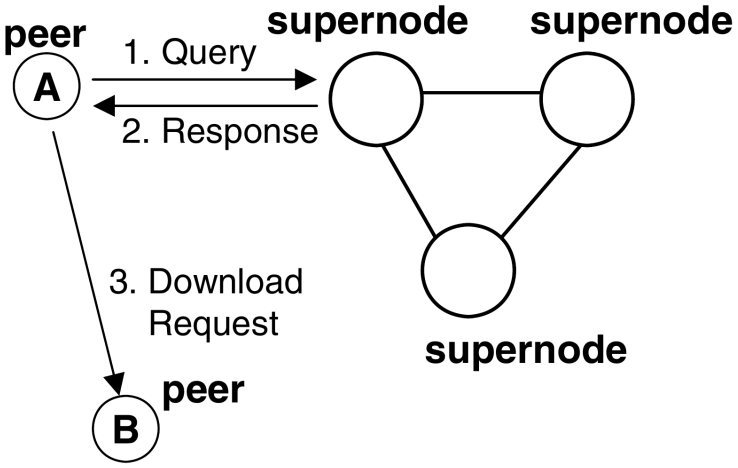

FIGURE 2.4: Search process in Kazaa. Image source: [23].

An hybrid approach has faster and more successful search queries. Since only the super nodes flood messages into the network, this approach is cheaper in term of network resources. It also solves the problem of having a central point of failure present in centralized protocols. One super node failing or departing does not disable the whole system. If a super node fails, then the regular nodes can be connected to other super nodes.

2.2.4 Other Approaches: BitTorrent

In the BitTorrent protocol, in order to share a file, the peer creates a torrent file which contains meta-data information about the file as the size, and hashing information. In the protocol, the file is split into small pieces of fixed size. Each piece has a checksum which is also stored in the torrent file. The torrent file also contains the URL of a "tracker", which keeps a record of all the peers that have the file (complete or not).

When a peer wants to download a specific file, it first downloads the torrent file. This allows it to connect to the tracker. The tracker server then returns a list of the peers that posses the whole file or that are downloading it. This information allows the peer to start the communication with the other nodes sending download requests. Figure 2.5 shows the search mechanism used in the BitTorrent protocol.

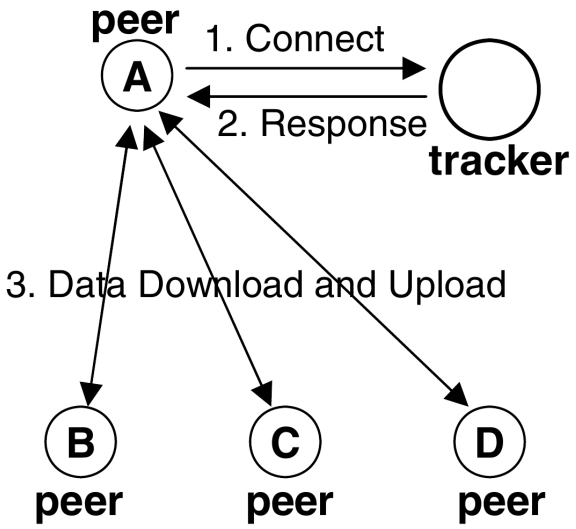

FIGURE 2.5: Search process in BitTorrent. Image source: [23].

The tracker server is a central entity. This is a problem of these systems when faced to "lawyer based" attacks. For example, the famous site The Pirate Bay stopped in 2009 serving as a tracker amid legal troubles [27] since many of the files stored and shared between peers were copy righted material. This disgusted right-holder companies which filed a civil law-suit. This problem has brought research on the area of decentralization of the tracker during the last decade. For example, the use of Peer Exchange (PeX) protocol augments the BitTorrent protocol allowing peers participating in the share of a specific file to directly update other peers about what other files are being shared by those participating nodes. Another example is the proposed in [28] which implemented a mechanism of peer discovery

alternative to the central tracker approach. The researchers uses the Tribler protocol, based on the BitTorrent protocol, and its creators claims that "The only way to take it down is to take the internet down"[29].

2.3 Personal Storage in Peer to Peer Networks

Solutions for personal storage in P2P networks have been recently proposed. For example, [30] proposes Storj, a protocol to store in a decentralized manner personal information. By removing the need of a centralized agent like in the traditional cloud storage services, the protocol promises to be strong against security threats. Man-in-the-middle atacks, malware and application hacks that expose private information are not efficient if the data is distributed among different nodes.

This protocol distribute the information of particular users in decentralized P2P networks. This brings several challenges to the system. The previously mentioned problem in file sharing P2P systems of finding where the files are located in the network is still present in Storj. Moreover, since this is a system not intended for file sharing, it is important to keep the data encrypted in order to protect sensitive and private information of the users against malicious nodes.

The distribution of information among peers brings the problem of having to use processing power resources at the user side. This is specially problematic if the client node is a mobile terminal. For instance, the company Wuala in its origins used to store information in peers close to the consumer of the data. However they stopped doing this and decided to change to the traditional client server approach, among other reasons, because they wanted to reduce the resources usage at the client side in terms of processing power and data transfers [13].

Chapter 3

Network Coding in Peer to Peer Distributed Storage Systems

If a peer wants to distribute a file in a network, and at the same time to guarantee that the information is reliable when some of the nodes depart or fail, then it needs to decide how to upload the information. For instance, the simplest approach would be to add redundancy in the system by storing replicas of the file. At the same time, the network must decide the actions that are going to take place when any of the nodes disconnects.

To distribute a file, for example, into four nodes we could split it into four pieces and store one on each node. As long as the node that uploaded the file is connected in the network the file can be recovered independently of the state of the other nodes. However, the system must be prepared for the departure of the uploader. The preparation involves making sure that in the other nodes, there is at least a copy of each one of the four pieces of the file as shown in Figure 3.1.

But if the uploader departs, then the system must take actions to guarantee that if another node disconnects then the file will be recoverable. One way to do it is by making a copy of the piece that each node is storing into another node as seen in Figure 3.2.

The problem of maintaining reliable the information in a distributed storage system has been studied in the past demonstrating that there are better alternatives than simply storing copies of the file in order to maintain reliability of the information. In the following sections we discuss the state of the art results for the actions that distributed storage networks perform when

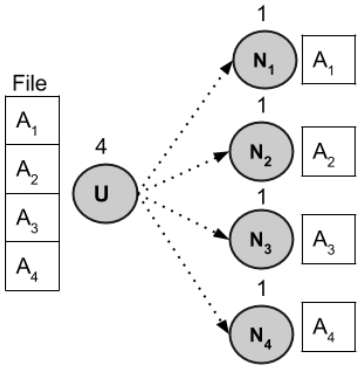

FIGURE 3.1: A node splits a file into four pieces and stores them into four different nodes.

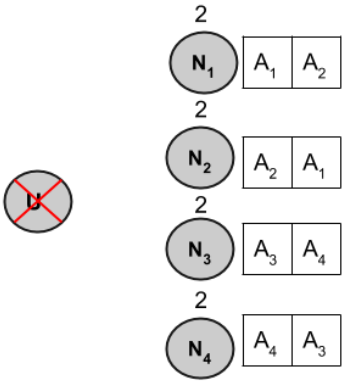

FIGURE 3.2: When the uploader disconnects, the nodes must take actions and copy the pieces they are storing into other nodes.

nodes fail. Subsequently we extend these results to the scenarios where the distribution of information is performed into P2P networks. These type of networks have the particularity of being dynamic, meaning that the number of available nodes for the storing of the information changes in time. Besides the disconnections of the peers do not necessarily means that the data that they store is lost. Instead the information is unavailable temporarily.

We are interested particularly in the case of wireless P2P networks. We will show that by taking advantage of the broadcast nature of wireless channels, it is possible to reduce the number of packets transmitted in the network when any of the nodes disconnects and the information that it was storing has to be repaired.

3.1 Coding for Distributed Storage

One problem with the replication of parts of the file in the network is that the only way to guarantee that the file is recoverable is having at least a duplicate of the file in the system. This means that for storing a file of four pieces, four redundant pieces must be added to the system. In Figure 3.3 it can be seen that if only 50% of the file is stored as redundancy then the system cannot guarantee that the file will be reliable when any of the nodes departs. The system guarantees reliability only if the node that departs is storing a piece replicated somewhere else in the network.

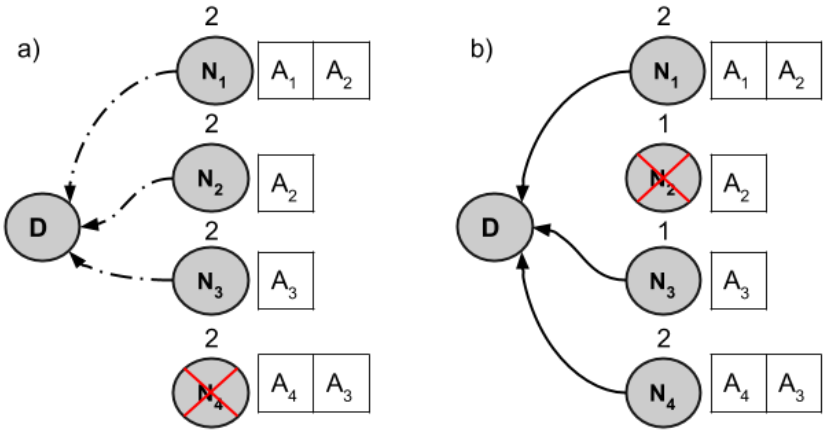

FIGURE 3.3: Example shows that 50% of redundancy cannot guarantee 100% reliability when any node disconnects. In case a) the file is not recoverable, in case b) the file can be recovered.

However there is an alternative to the storage of exact replicas of the pieces in the network. For instance, by using MDS codes such as Reed-Solomon codes it is possible to encode the four pieces of the file into coded packets that can be distributed into the nodes. The advantage of this method is that any set of four coded packets are enough to reconstruct the file.

Lets examine the example shown in Figure 3.4. In this example, the same file of four pieces A_1, A_2, A_3 and A_4, is encoded into six coded packets $R_1, R_2, ..., R_6$, using an $(6, 4)$ MDS code, which are stored into the nodes. By doing this and opposed to the example of Figure 3.3, a redundancy of 50% of the size of the file is enough to guarantee that the information is recoverable no matter what node leaves the network.

The problem of distributed storage in P2P networks goes beyond than just encoding a file and distributing the coded packets among the nodes. If our interest is only the tradeoff redundancy-reliability, then we can consider the problem as solved; MDS codes are optimal in this tradeoff [10] because

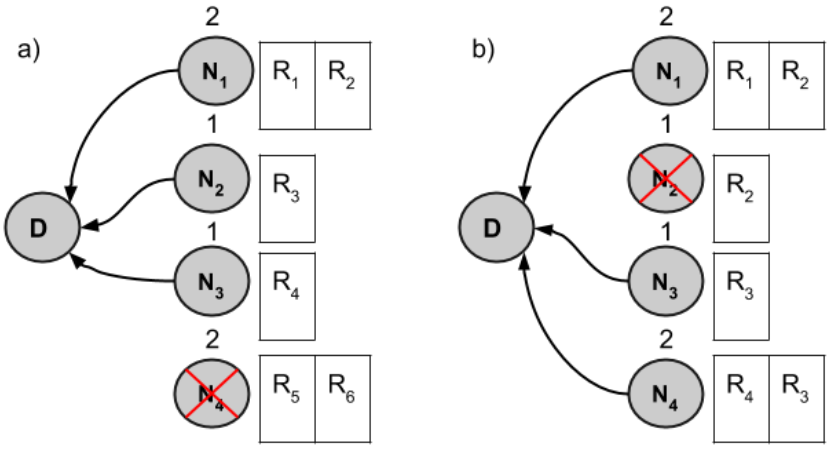

FIGURE 3.4: Example shows that when using coding 50% of redundancy can guarantee 100% reliability when any node disconnects. In both cases the file can be recovered.

a file split into k pieces and encoded with a (n, k) MDS code contain the minimum information required to recover the original data. However, when a node fails or disconnects, then the system must take actions to maintain the same level of redundancy-reliability. For instance, if one of the nodes fails or departs, the other nodes must reconstruct in the system the redundancy lost with the node. These actions require network resources i.e. the maintenance requires bandwidth to be performed. Distributed storage systems are built in shared networks where sometimes the time the costs in terms of bandwidth are more expensive that the costs in terms of storage. In that sense, the optimal tradeoff of redundancy-reliability is not enough. We might be interested in sacrificing some of the storage resources to be able to perform maintenance in the system using less bandwidth.

Most of the researchers study the reconstruction of the redundancy in newcomers nodes (i.e. when a node fails, some redundancy is built into a new node in a process called repair) and focus in the costs in terms of bandwidth and storage associated with the repair. When MDS codes are used and the system performs a repair, it needs to reconstruct the whole file in a node, and subsequently generate new coded packets that will be stored in a newcomer node.

This means that if each node stores a fraction M/k of a file, to repair the redundancy lost with a disconnected node, then the process would require the download of $(k-1) \cdot M/k$ bytes into a node so it can reconstruct the file. This node would then generate a packet of size M/k bytes that it would transfer to the newcomer node. In conclusion, to repair M/k bytes the system must transfer $(k-1) \cdot M/k + M/k = M$ bytes i.e. the whole file

must be transfered over the network. This example is shown in Figure 3.5 with a system which distributes a file into six nodes using a $(6, 4)$ MDS code.

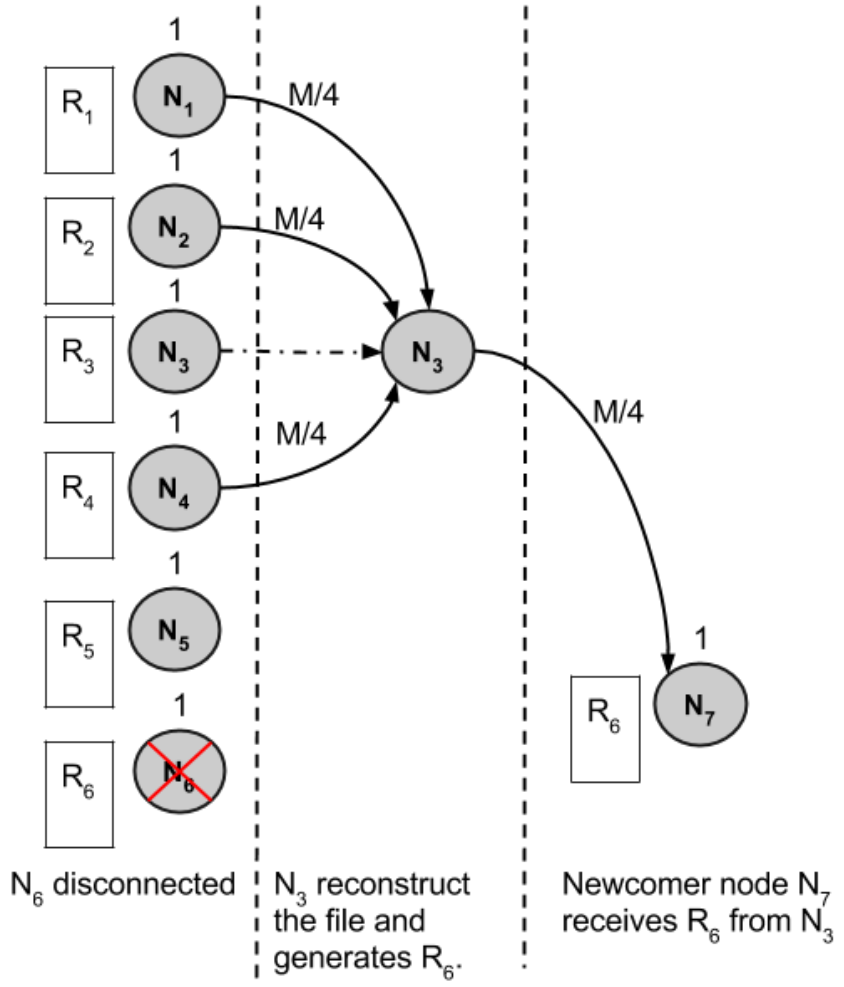

FIGURE 3.5: Example of the repair process when using a $(6, 4)$ MDS code. The whole file must be transfered over the network.

This tradeoff in this type of repairs has been recently studied in [9, 10, 31, 32] among others. Concluding that by using network coding it is possible to generate codes capable of reducing the bandwidth required for the repair. They [9] found that , it is possible to find the optimal curve describing the tradeoff storage-bandwidth, and that such curve can be achieved using network coding. The optimal curve has two special points of interest: The MSR codes with a similar behavior to the MDS codes in terms of bandwidth required for a repair; and the MBR codes, which sacrificing storage resources are optimal in bandwidth usage, since they are able to make the repair of a missing node by transferring over the network the minimum theoretical information.

However, sometimes the systems might not be able to bring a newcomer node to the network, and the repair of the redundancy must be performed in the nodes already present in the system. In this project we are interested in expanding the concept of the repair. We are interested in studying what are the actions that the system must perform when the disconnections of the nodes are not only due to failures but also due to departures i.e. when the nodes that disconnected do not lose all the information that they were storing but instead they reconnect after some time with the same data that they had stored at the moment of the disconnection. In that case the system must perform a repair of different nature, not with the objective of reconstructing the redundancy lost, but instead intended to remove the extra redundancy now that a node reconnected bringing back information to the system. We are also interested in evaluating some of the results of [9] in the case where the repair is not performed in a newcomer node but instead it is built in the remaining nodes, thus preparing the system for another disconnection.

The type of repair that we are going to consider for the rest of the project is the so called functional repair. In the bibliography there are described three types of repairs [10]: If the nodes have to repair exact copies of the packets stored in the failing node, as in the example of Figure 3.5, then it is called an *exact repair*. If the restriction of exact repair is relaxed, and we only need to guarantee that the code maintains MDS properties, i.e. that only the minimum number of packets are needed to recover the information. Then we are talking about a *functional repair*. If on the other hand, the system have to perform the storage with a systematic code, meaning that at all time exists in the system one copy of uncoded data, then the repairs of the systematic parts of the code are done with an exact repair approach while the non systematic parts are repaired with a functional repair approach. In this case we are talking about an *exact repair of the systematic part*.

3.2 Network Coding and the Functional Repair in Newcomer Nodes

The key in [9] to describe the tradeoff storage-bandwidth problem of distributed storage systems is the definition of the *information flow graph*. Dimakis et al. define this as an acyclic graph which consists of three different kinds of nodes. An uploader U, storage nodes N^i_{in}, N^i_{out}, and a downloader. The uploader is the node that possesses the original file, the storage node i is represented by an storage input node N^i_{in} and a storage output node

N^i_{out} connected with a directed edge $N^i_{in} \to N^i_{out}$ with capacity equal to the amount of data stored at the node i. The nodes of the graph might be at any point in time *active* or *inactive* since the graph evolves in time. When the uploader wants to distribute a file into the storage nodes, it connects to the storage input nodes $U \to N^i_{in}$ with edges of infinite capacity. When the uploader disconnects, its vertex becomes inactive. A node j can only download information from active vertices. If a node j downloads information from a node i, then it is represented as a directed edge $N^i_{out} \to N^j_j$ with capacity equal to the amount of data downloaded. A downloader is a node that intends to reconstruct the data stored in the system. It connects with any set of active nodes through directed edges $N^i_{out} \to D$ of infinite capacity.

The main idea in [9] is that given a system of n active storage nodes, storing α bits each, where a repair is done such as a newcomer downloads β bites from each d surviving nodes for a total repair bandwidth of $\gamma = d\beta$, then exists a family of information flow graphs for each set of parameters $(n, k, d, \alpha, \gamma)$ (they focus on the case where any k nodes can recover the whole file and the newcomer node downloads the same amount of information from each node). This family is denoted as $G(n, k, d, \alpha, \gamma)$ and it is said that a tuple $(n, k, d, \alpha, \gamma)$ is feasible if a code with storage α and repair bandwidth γ exists. Subsequently they prove that exists a threshold function $\alpha^*(d, \gamma, k, n, M)$ such that for any $\alpha \geq \alpha^*$, the tuple $(n, k, d, \alpha, \gamma)$ is feasible. And also they find that there is an optimal tradeoff (α, γ).

There are two special points in their optimal tradeoff curve. The minimum storage point is

$$(\alpha_{MSR}, \gamma_{MSR}) = \left(\frac{M}{k}, \frac{Md}{k(d-k+1)}\right) \tag{3.1}$$

It is important to note that when $k = d$ i.e. when the number of nodes that the newcomer contact is equal to the minimum required to recover the file, then $\gamma = M$. This means that it is optimal to download the whole file. This behavior, when $k = d$ is similar to the MDS codes in terms of redundancy-storage tradeoff and bandwidth-storage tradeoff. However if we let the newcomer contact more nodes such that $d > k$, then there are codes that outperforms the MDS codes with the same storage requirements in terms of repair bandwidth.

Similarly, the other special point in their optimal tradeoff curve is the minimum bandwidth point.

$$(\alpha_{MBR}, \gamma_{MBR}) = \left(\frac{2Md}{2kd - k^2 + k}, \frac{2Md}{2kd - k^2 + k} \right) \tag{3.2}$$

In this point $\alpha_{MBR} = \gamma_{MBR}$ which means that to repair a node it is needed to transfer exactly the same amount of bits that each node stores. This behavior is similar in terms of bandwidth to the replication approach.

The main idea for reaching their results is to analyze the information flow graphs as multicasting problems. Then the results offered by linear network coding show that in multicast networks with a single source and multiple receivers it is possible to achieve the optimum min-cut max flow bound [33, 34]. Linear newtwork coding achieves this multicast capacity [35]. This means that if the min-cut of the information flow graphs is at least of the same size of the file M, then it is possible to recover the information.

To illustrate this idea, lets consider the example shown in the information flow graph in Figure 3.6 of a file of size $M = 12$Mb that we want to store in three nodes. If we use equation 3.1, with $k = d = 2$ then we get the tuple $(\alpha, \gamma) = (6, 12)$. This means that we need to store 6 coded packets of size 1Mb each into each one of the nodes. According to these results, if a node disconnects, a newcomer needs to contact the two remaining nodes and download 12Mb in total to repair the redundancy. As shown in the figure, when a downloader contact two nodes and requests the file, the minimum cut of this graph is equal to the size of the file. According to [33, 34] then it is possible to use linear network coding to achieve this bound and recover the file.

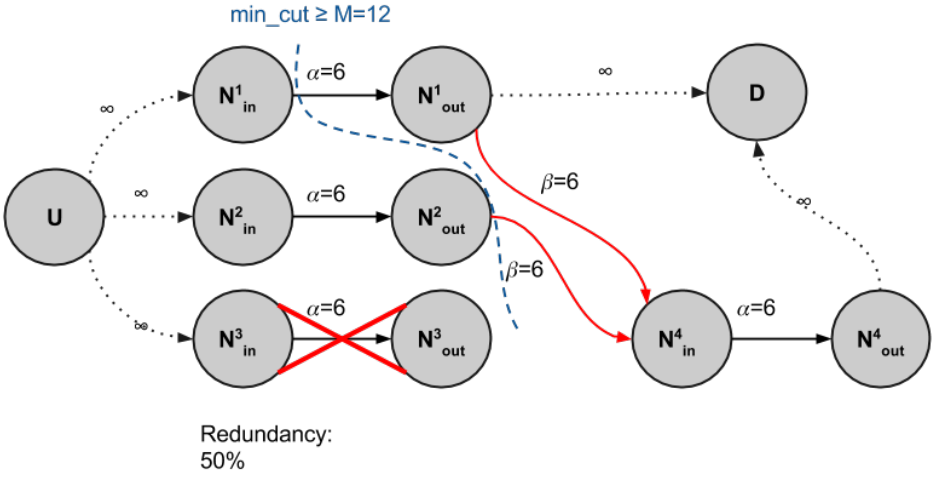

FIGURE 3.6: Example showing the information flow graph for MSR codes and a file of 12Mb stored in 3 nodes. All sizes are in Mb.

In a similar way, if we use equation 3.2, with $k = d = 2$ then we get the tuple $(\alpha, \gamma) = (8, 8)$. In this case, a newcomer node needs to download 8Mb in total from the surviving nodes. It can be seen that the minimum cut of this information flow graph shown in Figure 3.7 is also equal to the size of the file. This shows that with linear network coding, by storing more redundancy in the system it is possible to repair the information lost with the disconnection of a node transferring the information theoretic minimum [9].

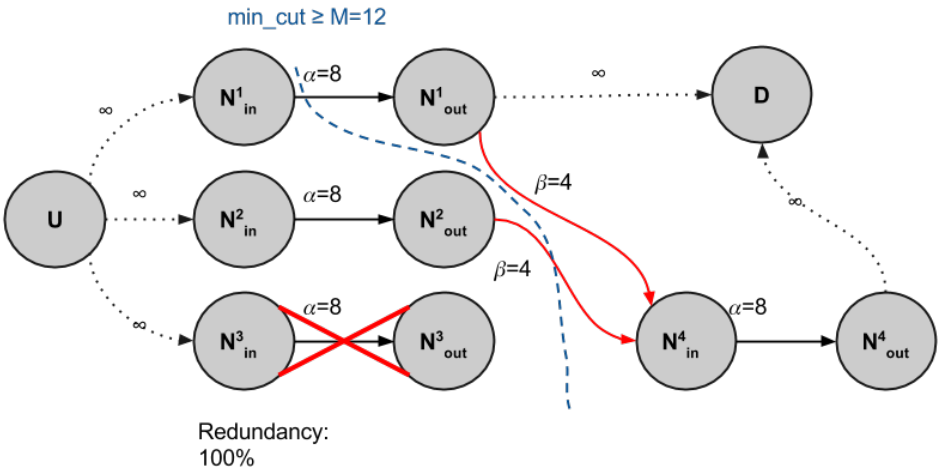

FIGURE 3.7: Example showing the information flow graph for MBR codes and a file of 12Mb stored in 3 nodes. All sizes are in Mb.

3.2.1 Random Linear Network Coding

Since we are interested in implementing a distributed storage system decentralized and with the minimum possible coordination, we will consider random linear network coding for our approach. Results from [36] show that RLNC achieves good network codes with high probability. This means that with high probability it will be possible to achieve the multicast capacity in the information flow graphs described previously.

In RLNC a coded packet p_j is generated producing linear combinations of the the original data pieces $x_1, x_2, x_3, ..., x_k$. Such as:

$$p_j = \sum_{i=1}^{k} c_{i,j} \cdot x_i \tag{3.3}$$

The coefficients of equation 3.3 are chosen randomly from a q-element finite field usually of the form $\mathbf{GF}(2^m)$. A finite field is a finite set with well-defined and efficiently implementable addition, subtraction, multiplication

and division [35]. This means that we can consider each coded packet as a linear equation with k variables. Since addition and multiplication are performed over the finite field, then the size of the coded packets will be the same as the size of the original pieces. And it is possible to use all the known tools from linear algebra for solving linear equations, e.g. matrices, Gauss-Jordan elimination. With these tools, a decoder will need only k linear independent packets to be able to reconstruct the whole data. Equation 3.4 shows the relation between the coded packets, the coding coefficients and the original packets.

$$\begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ \vdots \\ p_n \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} & c_{13} & \cdots & c_{1k} \\ c_{21} & c_{22} & c_{23} & \cdots & c_{2k} \\ c_{31} & c_{32} & c_{33} & \cdots & c_{3k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_{n1} & c_{n2} & c_{n3} & \cdots & c_{nk} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_k \end{bmatrix} \tag{3.4}$$

The coefficients used to generate each p_i constitute a vector known as the *coding vector* which is appended as an overhead to the coded packet. The size of this coding vector in bytes depends on the number of original pieces used, known as *generation size* and the size of the finite field used.

For example to transfer a file of 100KB, it is split into 100 pieces of 1KB each. The generation size is then $k = 100$. Each coded packet is generated making linear combinations of the 100 original pieces and appending the coding vector. If the size of the finite field used is $q = 2$ (i.e. $\mathbf{GF}(2) = \{0, 1\}$) then the size of the overhead due to coding vectors will be $k \cdot \log_2(q) = 100$bits appended at the end of each coded packet. This means that each p_i contains 1KB of information and 100bits of overhead. The overhead corresponds to approximately 1.2% of the transfered packet. When the symbol size get bigger the overhead due to the coding vector becomes negligible.

When using RLNC there is another type of overhead due to linear dependency. Since the coefficients are being chosen randomly, then the probability of generating linear dependent packets is a function of the generation size and the field size [37, 38]. The overhead due to linear dependencies occurs because linear dependent packets do not provide new information to the decoder, so it becomes necessary to transmit extra packets. The greater the size of the field, the smaller the probability of generating linear dependent packets, but the greater the overhead due to the size of coding vectors.

The computational complexity associated with encoding and decoding increases when the generation size becomes bigger. For that reason, if the system needs to encode or decode a big file it first divides it into *blocks* or *generations* and then performs the encoding operations over these blocks of a more manageable size.

3.3 Our Contribution: Functional Repair Into a Reduced Set of Nodes

Imagine that we distributed a file into three different nodes in a network. One of them fails and we do not possess at the time the resources to connect a newcomer node to the system. Would we then be doomed to not be able to maintain the reliability of the information? If we have the storage resources in the available nodes we might be interested in building redundancy into them, just to keep the data reliable, transferring it later to a newcomer node when we get the resources. We are interested then into applying the concept of the information flow graph to the described scenario. How would the information flow graph look like in such case?.

To illustrate this, we are making the addition of *time windows* to the information flow graph. In every time window we draw the storage nodes N^i_{in}, N^i_{out}. The idea is inspired in the concept of trellis graphs and it is basically that if a storage node N^i is available in two time windows t_{m-1} and t_m, then there is an edge with capacity α equal to the data that the node is storing from N^i_{out} at t_{m-1} to N^i_{in} at t_m. If a node N^j downloads β bits from a node N^i then a time window is spanned and we draw an edge of capacity β from N^i_{out} at t_{m-1} to N^j_{in} at t_m. The first time window t_0 has an uploader node U connected to the storage nodes that it contacts to distribute the file with an edge of infinite capacity. In the same manner in the last time window there is a downloader node D connected to the nodes from which it downloads the file, also with an edge of infinite capacity.

Extending the idea in [9] to our scenario, the file can be reconstructed after several repairs if the minimum cut of the modified information flow graph is at least equal to the size of the file. If we use RLNC with a big field, for example $\mathbf{GF}(2^8)$, then the recovering of the file is possible with high probability.

In Figure 3.8 it is illustrated an example of a modified information flow graph of a file distributed into four nodes using the MSR codes. Using equation 3.1 when $k = d = 3$ meaning that for a repair we need to access

all the surviving nodes in order to retrieve the file, we get that the values of α and γ are

$$(\alpha_{MSR}, \gamma_{MSR})|_{k=d} = \left(\frac{M}{k}, M\right) \tag{3.5}$$

So in Figure 3.8 the uploader stores $M/3$ bits (neglecting the overhead due to coding vectors) into each of the storage nodes. This results in a total storage of $4 \cdot M/3$ bits i.e. a 33% of redundancy. When the node N^4 disconnects the other storage nodes trigger a repair. They elect a leader and each node transfers $M/3$ to it. This means that each node transfers to the leader all the data that they are storing. The leader node was storing already $M/3$ bits of the file. So after the transfer from the other nodes it has now enough packets to recover the file. However the intention of this node is not to reconstruct the file, but to recode the packets and transfer $M/6$ bits to each of the other nodes, to store itself $M/6$ more of the file and remove the extra information. These transfers are appreciated in the time window t_2 in Figure 3.8. After doing this, the system has stored $M/2$ of the file in each node, for a total of 50% redundancy. In total $\gamma = M$ bits were transfered over the network. This type of codes have the same behavior as MDS codes in this system since M bits need to be transfered to repair one node. In that sense, we can see the repair of a disconnected node using MSR codes when $k = d$ as the process of making, from a $(4, 3)$ MDS code (four nodes out of which only three are needed to recover the file), a $(3, 2)$ MDS code.

It is important to note that when using RLNC this is possible to achieve with high probability (with a big field), however it cannot be guaranteed. One possible approach to overcome this issue in a practical scenario using random codes is to store some more information in each node. And at the same time transfer some extra packets as the values of β_1 and β_2

In a similar way we illustrate in Figure 3.9 the modified information flow graph for the case of a repair using MBR codes. Using equation 3.2 when $k = d$ we obtain equation 3.6

$$(\alpha_{MBR}, \gamma_{MBR})|_{k=d} = \left(\frac{2M}{k+1}, \frac{2M}{k+1}\right) \tag{3.6}$$

This means that, as shown in Figure 3.9, the uploader node stores $M/2$ bits into each of the storage nodes i.e. a 100% of redundancy is in the system. But when a node disconnects the repair session requires only a total of $M/2$ bits transfered.

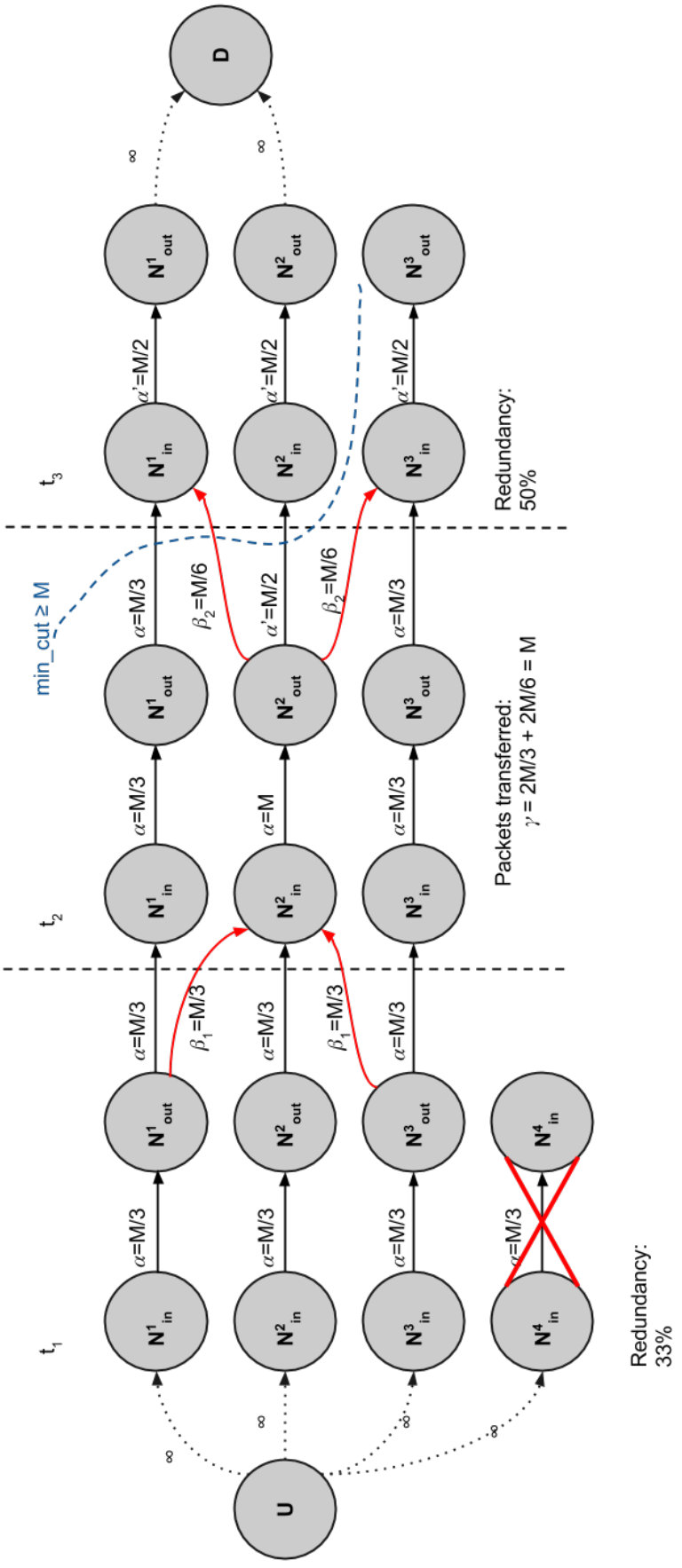

FIGURE 3.8: Information flow graph for a repair session in a system of four nodes using MSR codes.

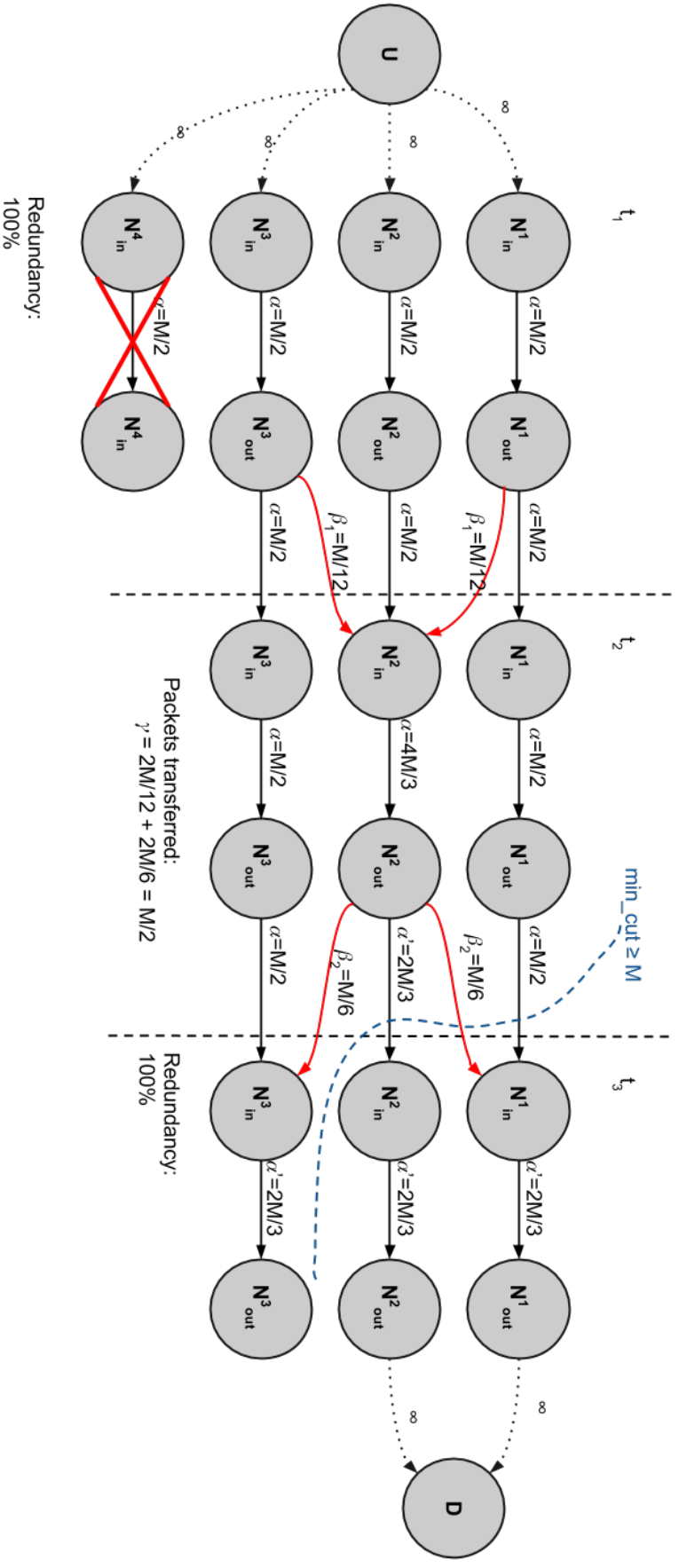

FIGURE 3.9: Information flow graph for a repair session in a system of four nodes using MBR codes.

3.4 Our Contribution: Distributed Storage Systems With RLNC in Wireless Lossy Channels

When managing a distributed storage system, we might be interested in storing the information as close to the peers that are consuming it as possible. This reduces the time between a message is requested and when it is served due to the round-trip delay. It could be possible to distribute the information not in peers but only within commercial cloud providers, this however would mean that when accessing the information, the system would need to wait enough time for the requests to be served. For instance, in [39] the authors distribute information into four commercial cloud providers (Dropbox, Box, Skydrive and Google Drive). And when they perform requests of packets to those services they obtain round-trip delays that vary from one second for the fastest responding cloud to 4.1 seconds for the slowest cloud.

Most of the time, the peers where we can distribute information share a wireless channel which they can use for the communications. In multi-hop networks, this can result in an advantage. For example by exploiting the spatial diversity and the broadcast nature of wireless channel it is possible to reduce the packet loss probability. The example in Figure 3.10 illustrate this scenario. The sender node is in a bad position in the network and it wants to transmit information to a destination node. If the sender had a single path to do this, then the best scenario would be achieved when the sender uses the path with less loss probability (the blue line). In this case the packet loss probability would be of 50%. However if the sender broadcasts the information, then any node that receives the packet can forward it to the receiver, thus reducing the packet loss probability in this case to $0.5^3 = 12.5\%$.

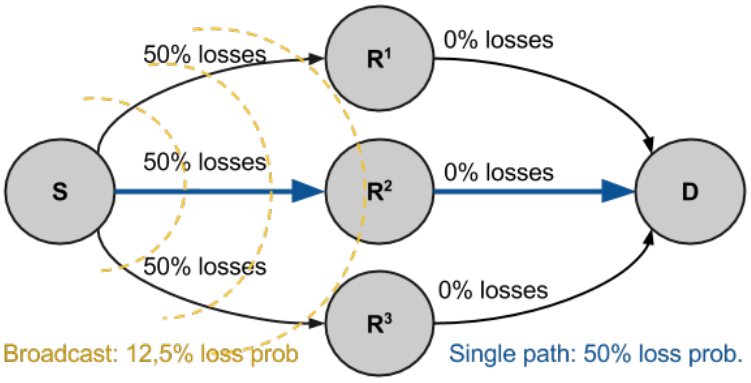

FIGURE 3.10: Example showing the advantages of spatial diversity in wireless networks

When the distributed storage system operates in a wireless lossy channel, then it is possible to exploit the benefits that RLNC offers for the communications in these scenarios, as well as the benefits of these types of codes in storage systems described in the previous sections. For instance, RLNC allows the recoding at intermediate nodes which can increase the number of innovative transmissions in case of losses in multi-hop networks.

When exploiting the broadcast nature of the wireless channels, it is possible to reduce the number of transmissions in the repair sessions. To illustrate this, let us consider an example of the MSR storage codes where a file consisting of six original pieces is stored into four storage nodes N^1, N^2, N^3 and N^4, in a systematic fashion using network coding with a binary field. After some time N^4 disconnects, which triggers a repair session among the surviving nodes. Let $\mathbf{C_i}$ be the matrix of the coding vectors of the packets stored in the storage node i. Since the system is storing the data in a systematic manner then

$$\mathbf{C_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}, \mathbf{C_2} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix},$$

$$\mathbf{C_3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Our previous results tell us that since we are using MSR codes, then we need to transmit 6 packets during the repair session (illustrated in Figure 3.11 with slightly simplification of the notation for the information flow graph). If each link has a loss probability of 50% then on average the repair session will require 12 transmissions to complete.

If instead of performing multicasting, the nodes involved in the repair broadcast their messages then the number of transmissions could be reduced. For example, lets consider the six initial broadcasts, two per node as shown in Figure 3.12, (full lines mean that the message was received, while doted lines means that the message got lost). Lets assume that for the initial transmissions each node sends a copy of the packets they are storing.

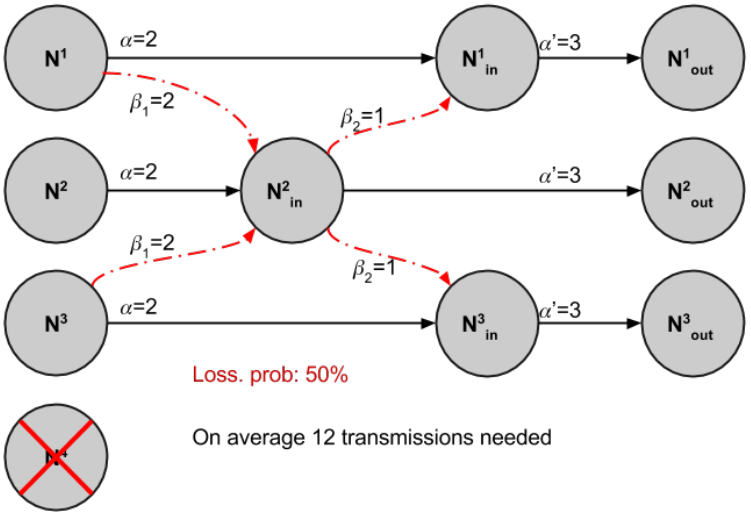

FIGURE 3.11: Example showing a simplified information flow graph of the repair session performed by 3 nodes storing a file of 6 pieces using MSR codes.

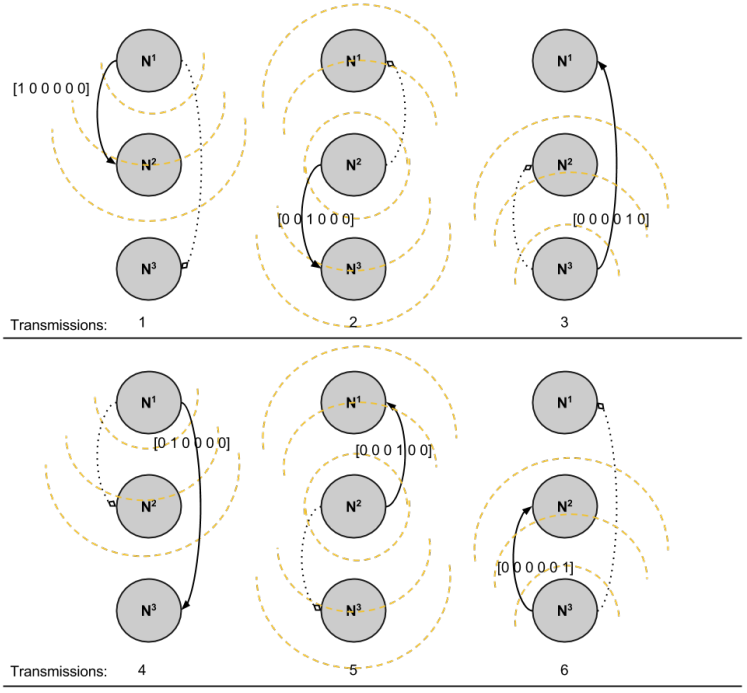

FIGURE 3.12: Initial six transmissions of a repair session using broadcast messages.

After the six initial transmissions with the losses described in Figure 3.12 the nodes have received the following packets:

$$\mathbf{r_1} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{bmatrix}, \mathbf{r_2} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix},$$

$$\mathbf{r_3} = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \end{bmatrix}$$

Where $\mathbf{r_i}$ is the matrix with the received coding vectors during the repair session.

Subsequently, in the following three transmissions, the nodes send linear combinations of all the packets received as shown in Figure 3.13. After the 9th transmission is complete, each node has enough information to store linear independent packets. In this example each link has a 50% loss probability. At the end of the repair, the matrices of coding coefficients are:

$$\mathbf{C'_1} = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 1 \end{bmatrix}, \mathbf{C'_2} = \begin{bmatrix} 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 \end{bmatrix},$$

$$\mathbf{C'_3} = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

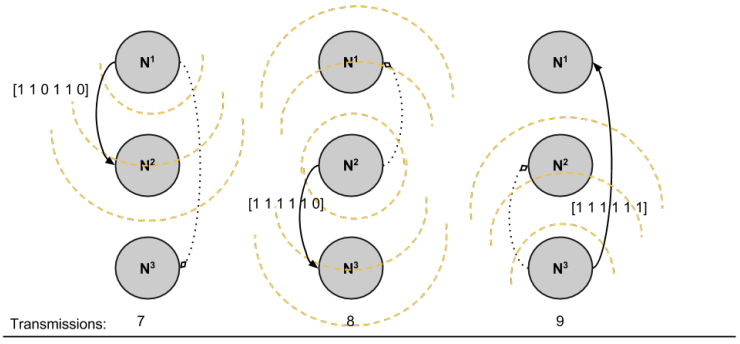

FIGURE 3.13: Final three transmissions of a repair session using broadcast messages.

Any two matrices $\mathbf{C'_i}$ are enough to recover the information. This example illustrates that only 9 transmissions are needed to perform a repair session.

Chapter 4

Protocol Design and Software Implementation

For the purpose of studying in a real scenario the obtained results in chapter 3 we implemented a system and installed it into a group of Raspberry Pis.

The system seen from the most general perspective is illustrated in Figure 4.1. It can be seen three different types of entities.

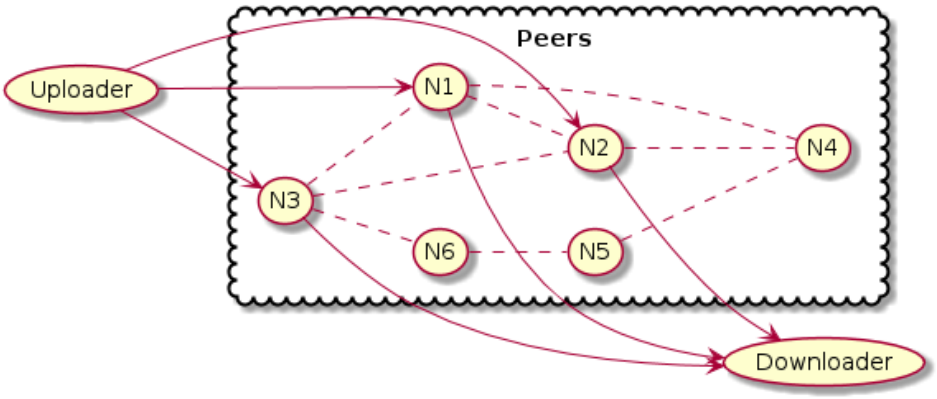

FIGURE 4.1: General view of the software implemented

- There is present a set of nodes $N1, N2, ..., N6$. Those nodes together constitute the set of all the peers in the scenario. To simplify the diagram, only six nodes are shown, however the system may contain many more. These peers might departure from the network at any moment and return also at any moment. The peers might experience total failure. In that case they don't return to the network, or return without the data they were storing.

- There is an *uploader*. It is a node with the task of uploading a file into the peers. After the upload is completed, it then dissapears from the scenario.
- A *downloader* is an object that contact the nodes of the system and requests to download a specific file from them.

The distinction between uploader, downloader and peer is merely logical. In practice, in the P2P network, the uploader and downloader are at the same time members of the set of peers. In that sense, any peer can request at any time to upload a file into any other of the peers. If the other peers possess the resources, then they allow the upload to happen. For example $N1$ can request to $N2, N3$ and $N5$ permission to upload 10 MB on each. If $N2, N3$ and $N5$ accept, then in the logical distinction of Figure 4.1, $N1$ becomes the *uploader* entity. When the upload session finishes, the *uploader* becomes $N1$ again. This example is shown in Figure 4.2.

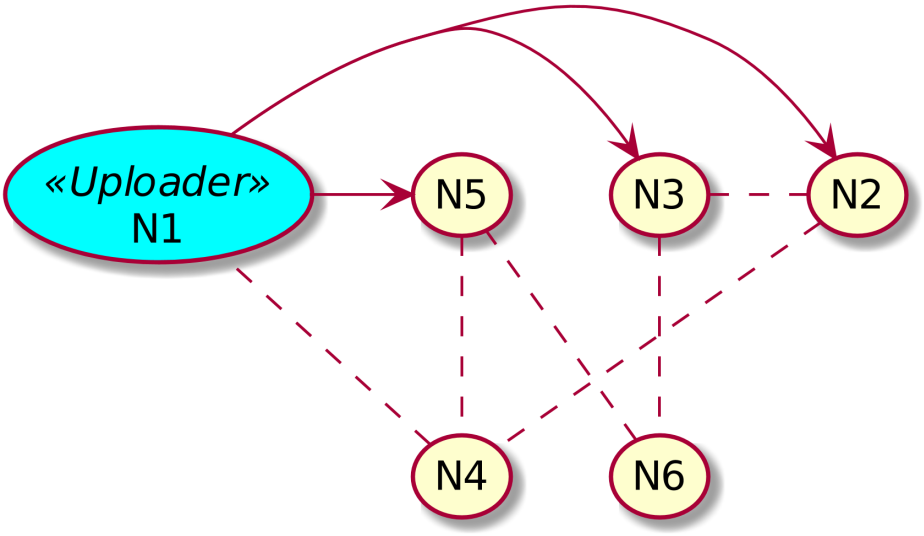

FIGURE 4.2: Upload session seen without the logical distinction of the *uploader* entity.

In the same manner, any peer can contact the nodes on the network and request to download a file. If the contacted peer possesses part of the file, they accept the request and the download session starts. Similarly as in the upload session, the node that requested the download becomes for a short period of time, i.e. until the download is finished, into the *downloader* entity.

The peers are constantly waiting for requests to upload or download from other peers. If a subset of the peers accept the upload of a file, that subset form a *pal network*. The set of pals of a peer is the set of all the peers

that are sharing the storage of a file with such peer. The objective of a pal network is to maintain the file or files uploaded into them reliable. The reliability is achieved adding redundancy to the system when any of the peers disconnects. This is done through *repair sessions*.

The following sections of this chapter describe the protocols implemented in the system for the upload, download and repairs of files in the system.

4.1 The Uploader

The uploader is an object whose objective is to distribute a file into a set of peers. It has the tasks of splitting the file into blocks or chunks, decide into how many peers will distribute it and decide how much data will upload into the peers. The messages that it exchanges with the target nodes are shown in Figure 4.3.

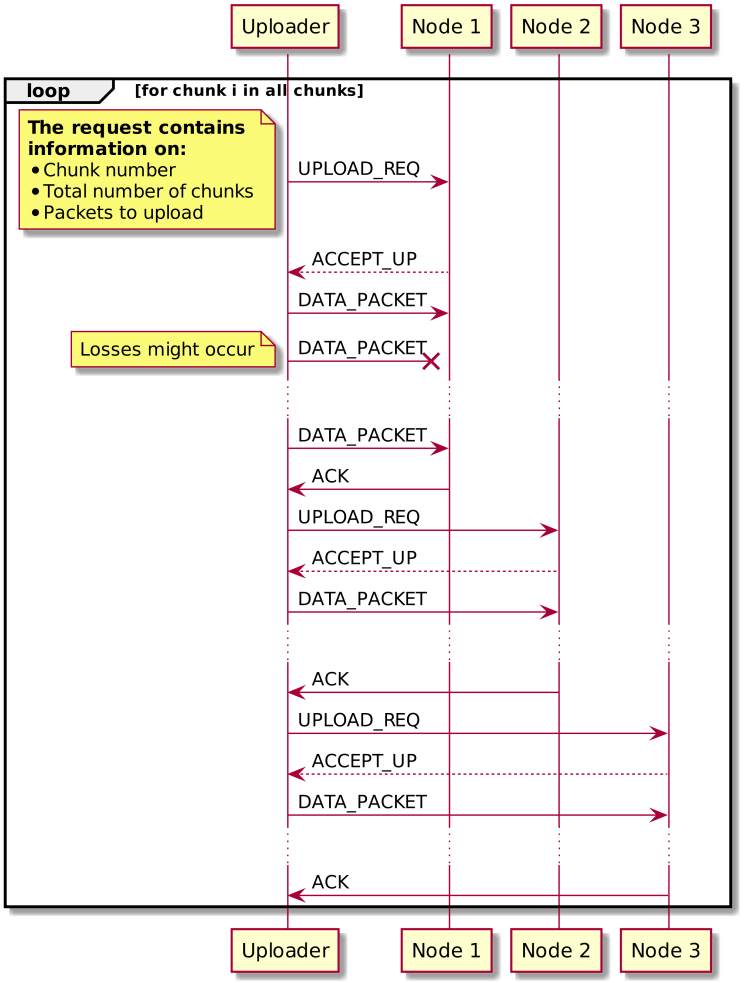

FIGURE 4.3: Example of messages exchanged between an uploader and three target nodes.

The uploader starts the session deciding the parameters of the encoder it is going to use. These parameters include the maximum generation size and the symbol size. After deciding these numbers it collects information on the file it is going to upload. The uploader determines the SHA-1 sum of the file and the file size. At this point it has enough information for splitting the file into *blocks* or *chunks*. The uploader calculates the size of each chunk according to the block partitioning algorithm proposed in the Request for Comments (RFC) publication 5052 [40] and described in Section 4.1.1.

Once the file is split into chunks, the uploader builds an encoder object, decides into how many nodes it is going to distribute that particular chunk and based on that, decides how many packets it is going to transfer to each peer. The uploader then contacts a target node sending a `UPLOAD_REQ` message. This request has information on what chunk is it transferring, the total number of chunks, the number of packets that it is requesting to upload and into what other nodes it is uploading the file. Based on this information, the node decides if it accepts or rejects the upload, answering `ACCEPT_UP` or `REJECT_UP` respectively. In our system the size of the symbols is fixed and it is known to all the peers. This means that each peer can calculate, when it process the `UPLOAD_REQ` message, how much data in bytes the uploader wants to upload into it. In case of a non fixed size for the symbols, this imformation could also be included in the `UPLOAD_REQ` message.

If the target peer answers with an `ACCEPT_UP` message, the uploader node proceeds to generate coded packets transferring them in `DATA_PACKET` messages. When the uploader receives an `ACK` message, then it picks the next node in the list of target nodes for that specific chunk and repeats the process of upload request and data transfer.

When this has been done for all the chunks then the upload has finished and the uploader object can be destroyed. Figure 4.4 shows a flow diagram describing this process.

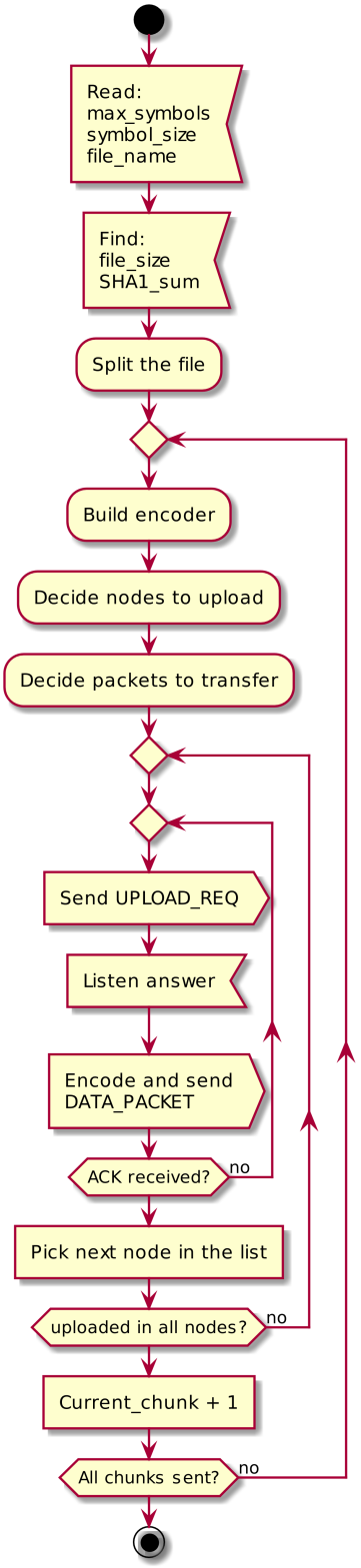

FIGURE 4.4: Flow diagram of the uploader object.

4.1.1 The Block Partitioning RFC5052

When deciding the size of each block, we use the algorithm proposed in [40] to compute the partitioning of an object into source blocks so that all blocks are as close to being equal to the same length as possible. The algorithm divides the object in two groups of source blocks: one group of the same larger length and a second group of smaller equal length.

The algorithm takes as input the size of the file, the desired maximum number of symbols and the maximum desired symbol size.

It basically fixes the symbol size to the size provided in the input as maximum desired symbol size, and based on that it finds the total number of blocks and the length in symbols of each block.

The algorithm is described in two steps. The first step, which computes the total number of symbols and the total number of blocks, is shown in equations 4.1 and 4.2.

$$Total\_Symbols = ceil\left(\frac{File\_Size}{Maximum\_Symbol\_Size}\right) \tag{4.1}$$

$$Total\_Blocks = ceil\left(\frac{Total\_Symbols}{Maximum\_Symbols}\right) \tag{4.2}$$

The second step of the algorithm computes the number of symbols of the larger blocks and the smaller blocks (shown in equations 4.3 and 4.4) and the number of larger blocks and smaller blocks (shown in equations 4.5 and 4.6).

$$Large\_Blocks\_Symbols = ceil\left(\frac{Total\_Symbols}{Total\_Blocks}\right) \tag{4.3}$$

$$Small\_Blocks\_Symbols = floor\left(\frac{Total\_Symbols}{Total\_Blocks}\right) \tag{4.4}$$

$$Large\_Blocks = Total\_Symbols - (Small\_Blocks\_Symbols \cdot Total\_Blocks) \tag{4.5}$$

$$Small\_Blocks = Total\_Blocks - Large\_Blocks \tag{4.6}$$

4.2 The Downloader

The task of the downloader object is to request a file to the peers. If the peers have coded packets of any of the parts of the file, they answer the request and the download session starts.

In Figure 4.5 it is illustrated the messages sequence between a downloader that wants to download a file and three target nodes that it contacts for this purpose.

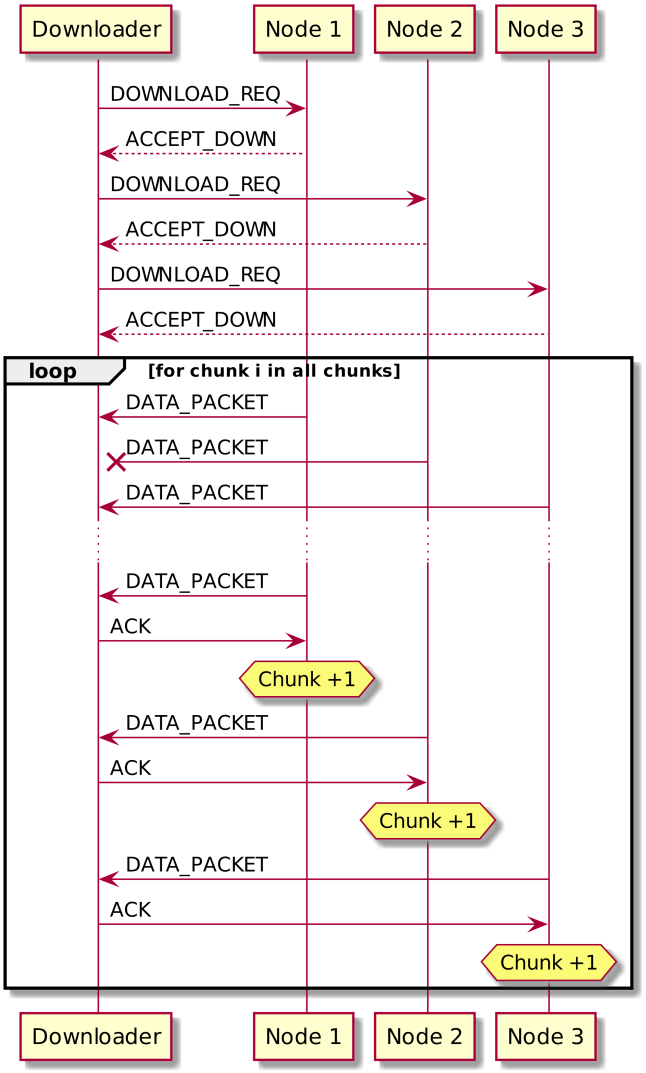

FIGURE 4.5: Example of messages exchanged between a downloader and three target nodes.

The downloader first obtains a file similar to the torrent file of the BitTorrent protocol described in Section 2.2.4 which contains meta-data information on the file that is stored in the network. Information about the size of the file, the number of blocks into which it was split, the size of each

block, the SHA-1 sum of the file and the IP addresses of the peers that are storing it in the network. With this information the downloader is able to start communications with the peers.

The communications start with the downloader sending a DOWNLOAD_REQ message with the SHA-1 sum of the requested file. The peers check if they have any coded packet belonging to that file, and if they do, they answer with the message ACCEPT_DOWN, otherwise they send the message REJECT_DOWN. If the downloader receives any positive answer by any of the peers then it instantiate all the decoders objects (one per each chunk) and waits for the transmissions of coded packets.

Every time the downloader receives a coded packet in a message DATA_PACKET which also contains information about the chunk that it belongs to, it feeds the coded packet to the corresponding decoder. When a chunk is successfully decoded the downloader sends an ACK message to the peer that sent the coded packet. These ACK messages contain information about the chunk that is being acknowledged. If all the file was successfully decoded, then the download session finishes and the downloader object is destroyed.

The downloader unicasts the ACK messages every time it receives a coded packet belonging to a chunk already decoded. This might lead to situations where extra and unnecessary overhead is introduced to the system. For instance, lets imagine the scenario where a downloader needs one more coded packet to decode a chunk as illustrated in Figure 4.6. The node that is sending the coded packets might send two packets p_i and p_{i+1} before the downloader can process the first one. So, after the downloader processed p_i it sends an ACK, however p_{i+1} is already in the queue of received packets in the downloader. This means that right after the ACK is sent, when it processes p_{i+1} it unicasts a second ACK to the transferring peer.

The second transmission of an acknowledgment is unnecessary if the transferring peer received the first one. However we decided to opt for this approach because in the presence of a lossy channel, the ACK messages might get lost. If that happens, the downloader is not able to know if the transmission of p_{i+1} occurred because the transferring node did not receive the ACK message and did not know that the chunk was decoded. This is shown in Figure 4.7.

As a complement to the message sequence in Figure 4.5, the flow diagram of the downloader object is shown in Figure 4.8.

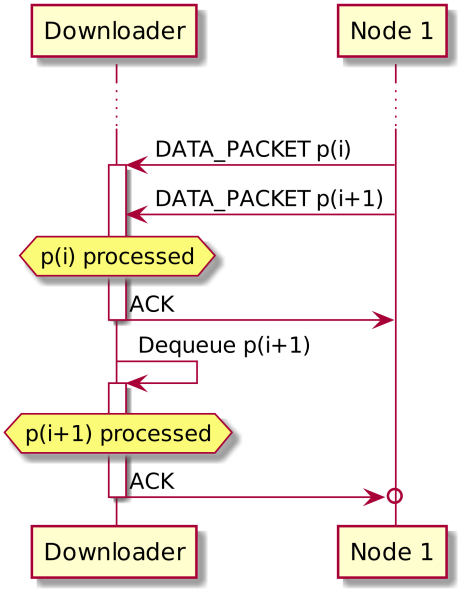

FIGURE 4.6: Scenario where multiple ACK messages brings unnecessary overhead to the system.

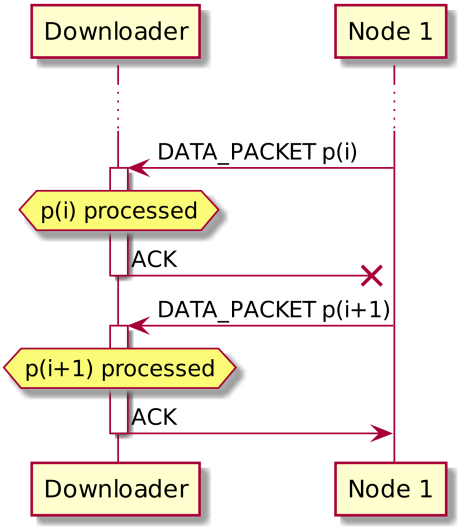

FIGURE 4.7: Scenario where multiple ACK messages are needed due to losses in the channel.

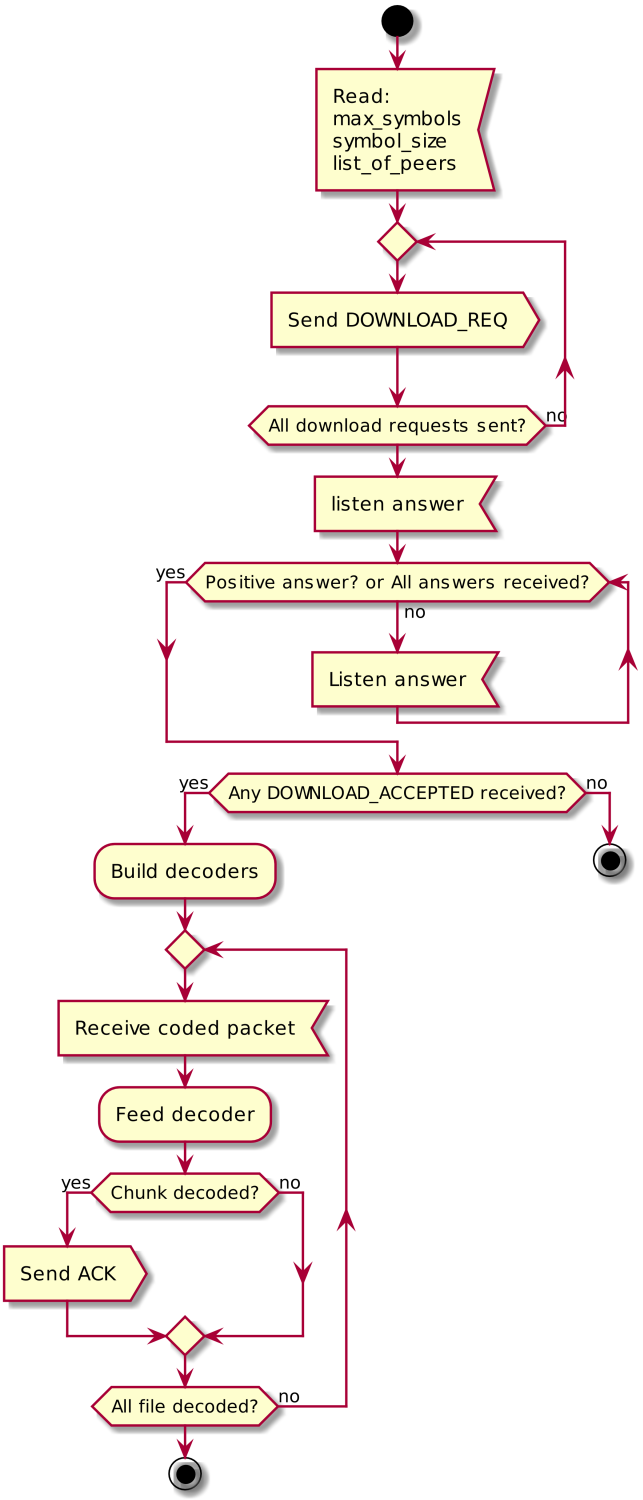

FIGURE 4.8: Flow diagram of the downloader object.

4.3 The Node

The node object is the base of the implementation. As mentioned earlier and shown in Figure 4.2 it is within the node object that the uploader and downloader objects are constructed. This object is created at the beginning of the application and remains in memory until the termination of the program.

The communications in our application were implemented using the library Boost Asio C++ [41], specifically the asynchronous calls for the read, write and timers operations. The advantages of this method over synchronous calls is that we can design our application to be single threaded, thus avoiding all the complications that multi threaded programming brings. The cost is the complexity involved when thinking asynchronously [42]. The logic of the program switches from a sequential perspective such as do A then do B then do C... and so on where A, B and C are blocking operations, to an event-driven perspective [41]. It is defined when asynchronous operations start but not when they end. Instead, when the asynchronous operation is called, a handler function is given as an argument of the call and the operation is scheduled. When the operating system performs the operation, the handler function is called. As long as there is any asynchronous operation scheduled, the program does not end. So in an asynchronous approach, when the handler of an operation is called, it should schedule a new asynchronous call so there are always scheduled tasks preventing the program to end. This is known as chaining asynchronous operations.

For that reason the best way to understand the node object is to think of it in terms of a state machine as the one shown in Figure 4.9. When the application starts, it binds itself to three sockets. One socket for signaling purposes such as upload and download requests, another socket for upload and download of data packets and a third socket used for the repair messages exchanged during the repair sessions described in Section 4.4.

If the binding of the sockets is accomplished successfully, the node then invokes the asynchronous operation of listening for requests. This operation is called with the handler `handle_request()` as an argument. The node remains in this state until a request arrives. When any request is received through the signaling socket, then the handler `handle_request()` is called changing the state of the node. In this state, the node processes the request, it evaluates it and decides if it is going to accept it, reject it or ignore it (in case of receiving an unknown message). If the request was accepted then the node schedules the asynchronous operation of sending the `ACCEPT_UP` or `ACCEPT_DOWN` message with the handlers `serving_uploader()` and `serving_downloader()` respectively. Before

leaving this state, it makes a new asynchronous call to `async_listen_request()` in order to be ready for receiving a new request.

At this point, after finishing the execution of the `handle_request()` function, if the message was not ignored, the node has made, at least two asynchronous calls: an `async_listen_requests()` and an asynchronous call to send the response to the request. If the request was accepted, the handler provided as argument to the asynchronous send is `serving_uploader()` or `serving_downloader()`. Inside these handlers new asynchronous operations are scheduled and chained in order to comply with the established protocols described in Figure 4.3 and Figure 4.5.

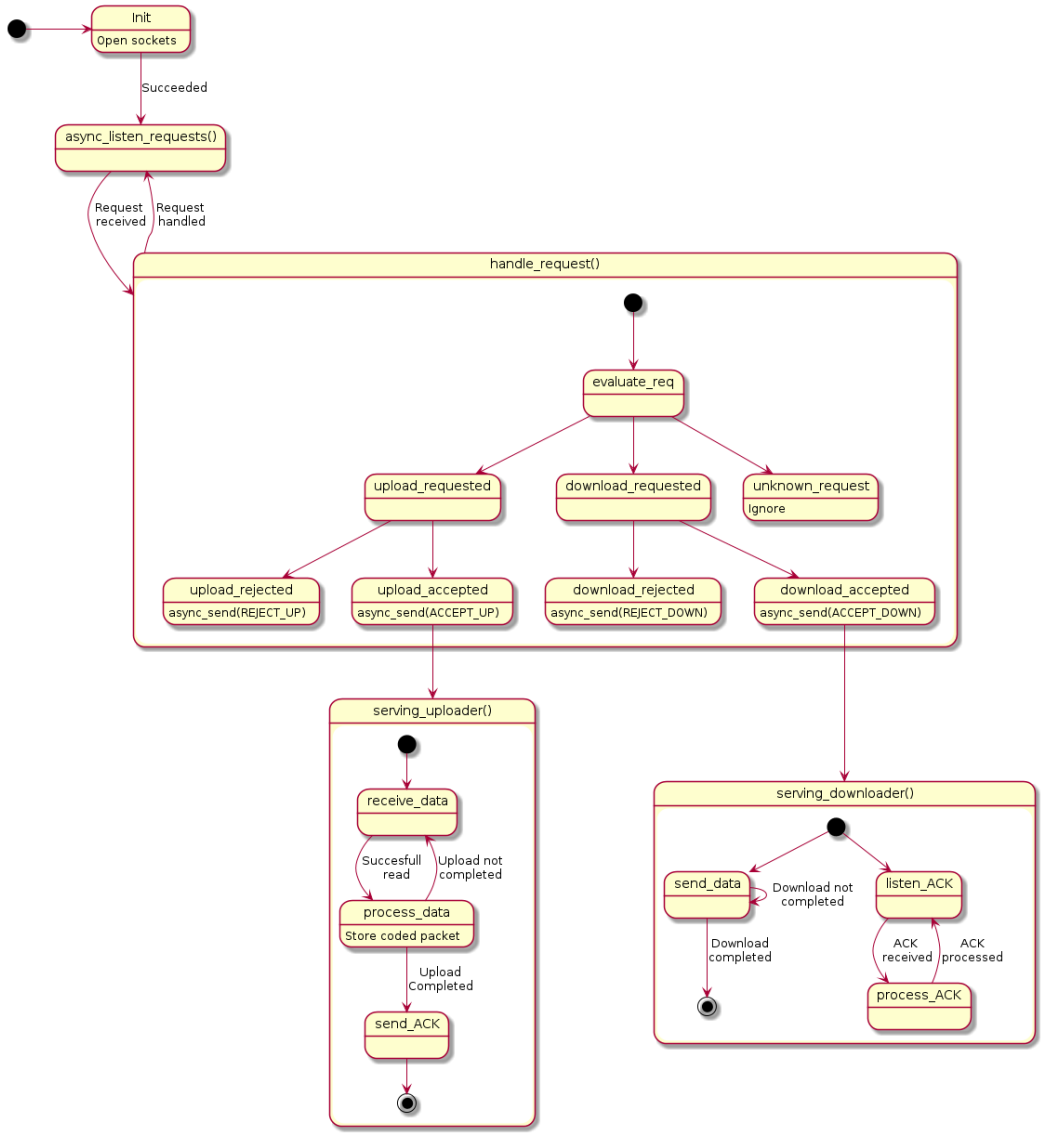

FIGURE 4.9: State diagram of the node object. It is the base of the application.

4.4 The Pal Web and the Repair Sessions

When the uploader uploads a file into a set of nodes, it also provides them with information about the list of nodes where the file was uploaded. A set of nodes into which a file has been uploaded form a *pal web*. Once a node joins a pal web, it broadcasts HEARTBEAT messages constantly to let the other nodes know that it is alive. The frequency of the HEARTBEAT messages can be modified, however for this application the time between broadcasts was set to three seconds.

From an implementation perspective, the pal web is an object that is constructed when the node accepts an upload request. It contains information on the files that are being shared and protected with the rest of the pals. It keeps track of the vital signs of each of the pals i.e. connected or disconnected. The pal web objects create the repair sessions objects when it detects the disconnection of a pal.

4.4.1 The Repair Session Due to Disconnection

When a node has not received the HEARTBEAT messages from a pal after a period of time, it creates an object called the repair session. This object is in charge of organizing the build of redundancy within the pal network to prepare the system for the disconnection of another peer.

Three stages constitute a repair session due to disconnection:

The First Stage of the Repair

In the first stage, a leader is chosen among the pals. When the pals broadcasts the HEARTBEAT messages, they also include a randomly generated key of four bytes long. This key is then used for choosing who the leader is going to be in a repair session. The node with the greater key, becomes the leader. In this stage, the pals also calculate how many packets should be transferred in the repair session and how many coded packets should they be storing at the end of the session. These calculations are performed based on the results obtained in Chapter 3.

The election of a leader in distributed systems is a complex problem by itself. Researchers have worked for decades in complex algorithms to solve this problem efficiently in terms of bytes transferred and time spent. For instance, [43] is an algorithm developed in 1983 which received the Dijikstra prize for its influence in distributed computing, and it is able to choose a

leader in a network of N nodes and E edges transferring at most $5N \log_2 N + 2E$ messages. However, for small networks our method of assigning and broadcasting a four-bytes key as `HEARTBEAT` used for deciding who is the leader showed to be sufficient and not that expensive in terms of resources. Since our protocol already needs the broadcasts of `HEARTBEAT` messages, making them of a size of four-bytes is not much expensive.

Our approach adds extra overhead to the system in case of two pals choosing the same key. In this scenario, the problem is solved as shown in Figure 4.10. The nodes with the same key must roll again a new key but this time constrained to a certain range in the key space. To better illustrate the solution of this problem, lets imagine that four nodes in a network chose the keys k_1, k_2, k_3 and k_4 such as $k_i \in K$ and $k_1 < k_2 = k_3 < k_4$. Where K is the set of all possible four-bytes keys. In this example the pals with keys k_2 and k_3 must choose new keys k_2' and k_3'. But instead of picking a key from the whole space K, the new keys must be picked from $\{x \mid x \in K,\ k_1 < x < k_4\}$.

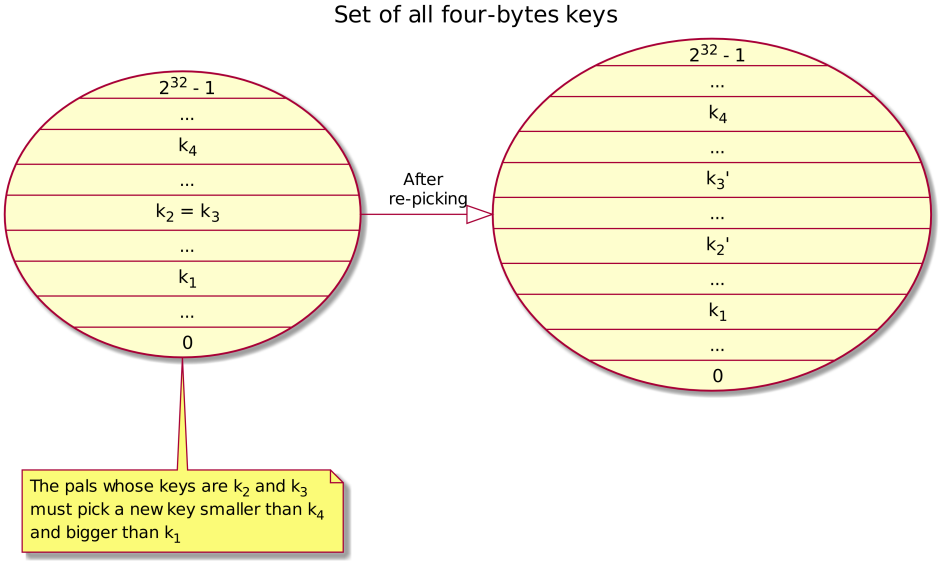

FIGURE 4.10: Scenario describing the problem of two pals picking the same key.

The system is ready to react in case of equal keys, but for a small number of nodes, the event of two pals choosing the same key is very unlikely.

Finding the probability of this event occurring is similar to the famous birthday problem, which computes the probability of two persons out of N randomly chosen having the same birthday. Equation 4.7 shows the probability of at least two pals choosing the same key out of a pool of $|K|$ keys in a pal web of N nodes.

$$p(N, |K|) = 1 - \frac{|K|!}{|K|^N \cdot (N-1)!} \tag{4.7}$$

With the Taylor expansion of $e^x \approx 1+x$ it can be approximated to Equation 4.8.

$$p(N, |K|) \approx 1 - e^{-N^2/2|K|} \tag{4.8}$$

Using Equation 4.8, it can be shown that for at least two pals choosing the same key out of the space of all four-bytes keys with a probability of 0.1%, the number of peers in the pal web, N, must be of approximately $2.9 \cdot 10^3$.

The Second Stage of the Repair

In the second stage, the pals builds all the recoders (one per each chunk that they are protecting) feed them with the coded packets that they are storing. Then, they send coded packets to the leader. The leader feeds these coded packets to its own recoders until they reach the desired rank for this the repair session, calculated on the first stage based on the results of 3. When such rank is reached in each individual recoder, the leader broadcasts an ACK message with information about the chunk that reached such rank. This messages sequence is summarized in Figure 4.11.

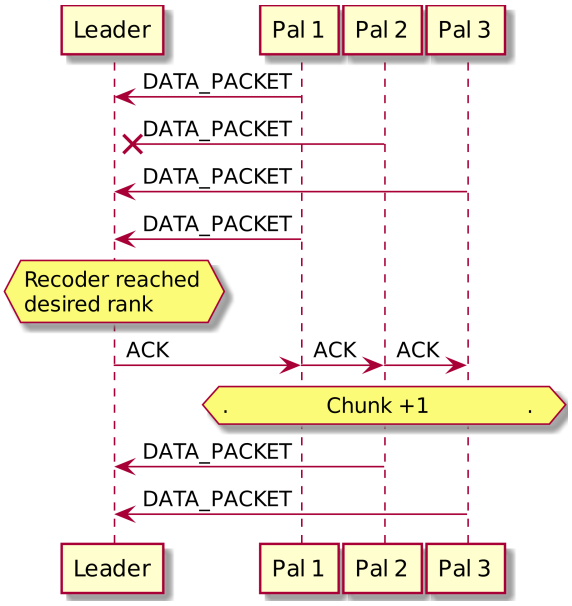

FIGURE 4.11: Example of the message sequence of the second stage of the repair with one leader and three pals.

The Third Stage of the Repair

In the third stage, the leader transmits recoded packets to the rest of the pals, one by one, in a round-robin fashion. The pals save these packets appending them to the already stored coded packets. When the nodes receive all the packets scheduled to store, number which was calculated in stage one, they send an ACK message to the leader. When this has been done for all the pals and all the chunks, the leader append some of the packets to its own file and the repair session ends.

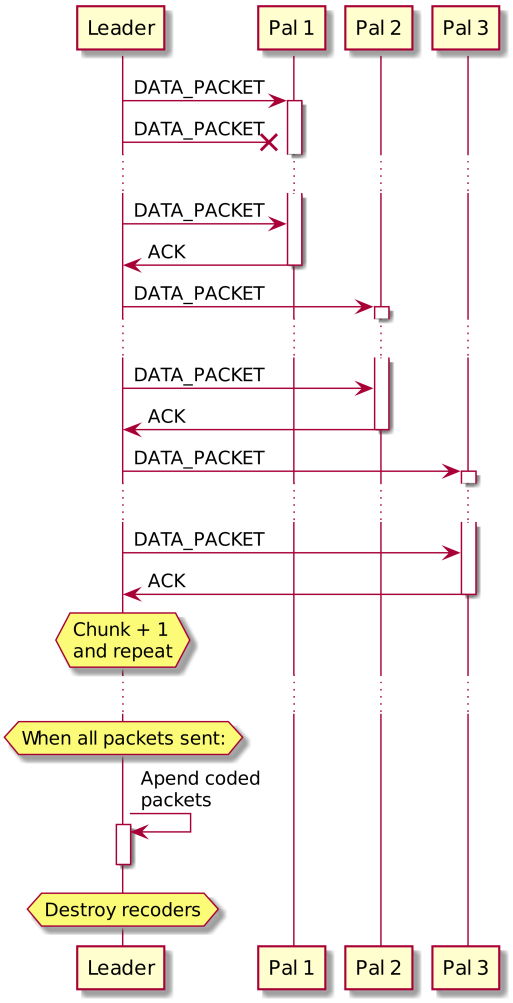

FIGURE 4.12: Example of the message sequence of the third stage of the repair with one leader and three pals. Each pal stores the received coded packets.

4.4.2 The Repair Session Due To Reconnection

When a pal reconnects to its pal network, it must inform the other pals that it has returned in order to organize a repair session due to reconnection. The

objective with this repair is to remove the extra redundancy in the system and distribute it equally among all the pals now that a pal has reconnected. This means three things depending on the amount of data stored in each pal at the moment of the disconnection. To explain them lets imagine the case where the system stores the minimum possible redundancy. If only three nodes are alive, it means that each one is storing coded packets summing up to a total of 50% of the size of each chunk (in this way if another peer disconnects, the whole file is recoverable). If a fourth pal reconnects, then after the repair session ends, all the peers must be storing 33% of the size of each chunk. So depending on the amount of coded packets stored in the reconnected peer at the moment of the reconnection there are three possible scenarios:

- If the peer disconnected in a moment when the pal network had a greater number of peers alive that it has at the moment of the reconnection, then the rest of the peers must send some coded packets to the reconnected pal so it can catch up, so it can store the same number of coded packets. This is illustrated in Figure 4.13. Then the pals must remove some of the redundancy. This is the same scenario in case of a new node joining the system.

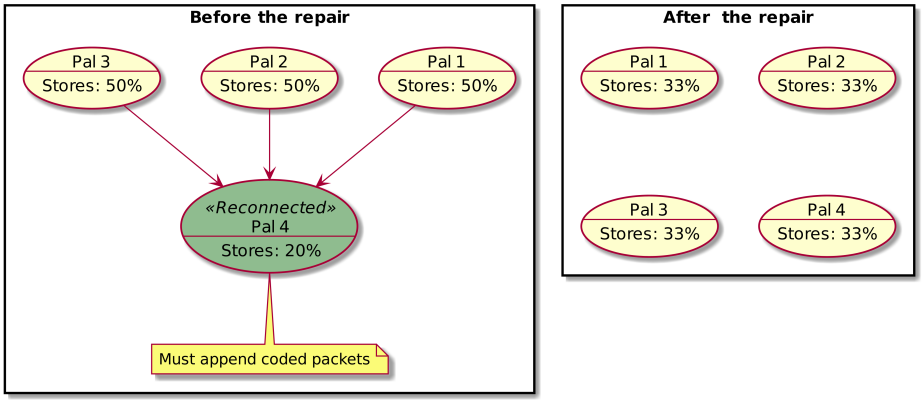

FIGURE 4.13: Actions that the reconnected peer must perform during the repair session when it must append coded packets.

- If the peer disconnected in a moment when the pal network had a lesser number of peers alive that it has at the moment of the reconnection, then all the peers, including the reconnected one must remove redundancy. This is illustrated in Figure 4.14.

- If the peer disconnected in a moment when the pal network had an equal number of peers alive that it has at the moment of the reconnection, then all the peers except for the reconnected one must

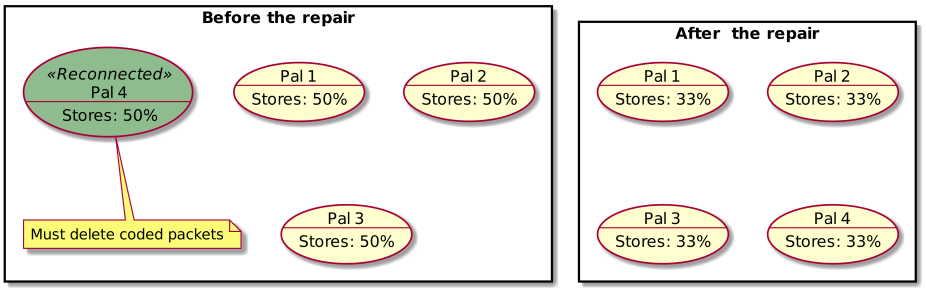

FIGURE 4.14: Actions that the reconnected peer must perform during the repair session when it must delete coded packets.

remove redundancy. The reconnected peer must not append nor remove coded packets. This is illustrated in Figure 4.15.

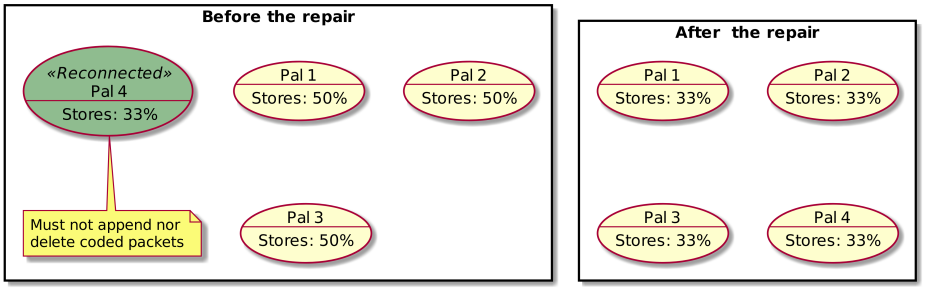

FIGURE 4.15: Actions that the reconnected peer must perform during the repair session when it must not delete nor append coded packets.

To distribute equally the file and the redundancy, the reconnected peer must notify the rest of the pals the number of packets that it stored at the moment of the disconnection. To do this, it broadcasts a `RECONNECTED` message which contains this information as shown in the message sequence in Figure 4.16.

The other pals, based on this message, calculate how many packets the peers on the pal web should be storing, how many packets should each individual pal remove and how many packets the reconnected peer should append or remove. They notify the last calculation to the reconnected peer with the messages `REMOVE_N` or `APPEND_N` depending on if the reconnected node must append or delete extra packets.

After sending a `REMOVE_N` message, each peer proceeds to remove the extra packets they store. Similarly, when receiving a `REMOVE_N` message, the reconnected peer also removes a certain number of packets it was storing depending on the information contained in the message.

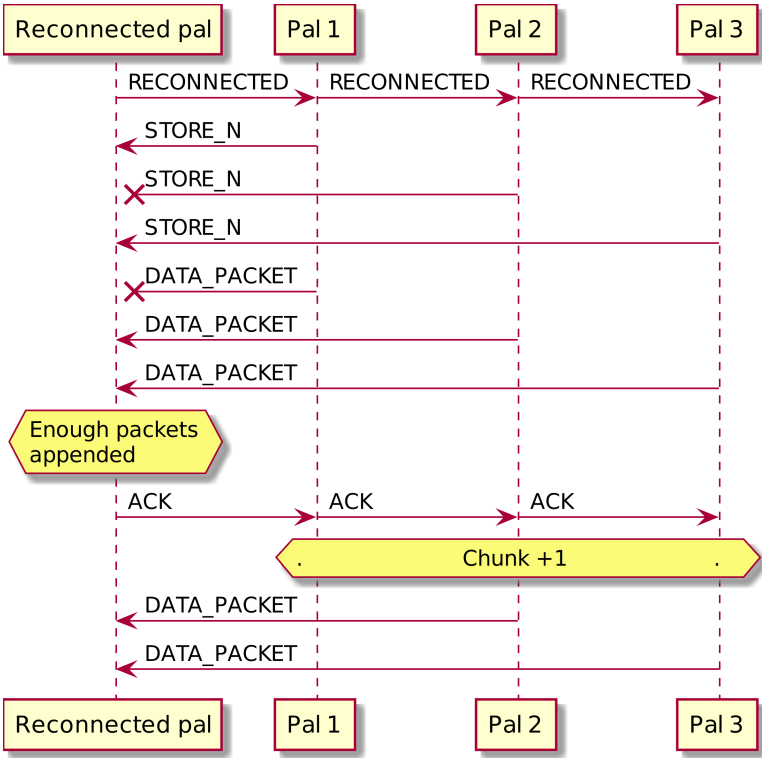

FIGURE 4.16: Messages sequence for a repair session due to reconnection where the reconnected peer must append coded packets.

If on the other hand the reconnected peer needs to append some extra packets to those that it is storing, then after receiving the `APPEND_N` message, it waits for the transmission of coded packets from the other pals.

Chapter 5

Measurements Results

Using the protocol described in Chapter 4 for distributed storage systems, we implemented an automated test scenario with eight Raspberry Pis devices connected in a network. In the automated test scenario, a central computer connects through Secure Shell (SSH) to all the devices, and it starts the application software in seven of them. These devices then wait for the request to upload a file from another node. After some seconds, the central computer starts the software in the eighth node. This node uploads a file of 500K bytes distributedly into the rest of the peers, using one of the strategies (MSR or MBR) and then disconnects from the network.

After the upload is completed, the computer controlling the test stops the running application in a Raspberry Pi chosen randomly. This disconnection, as explained in Chapter 4, triggers a repair session in the remaining nodes. For this repair session a number of packets are transferred among the nodes depending on the strategy agreed during the upload of the file. The number of packets transferred and the time spent in the different stages of the repair session are recorded. When the repair session finishes, the Raspberry Pi chosen as leader in the protocol notifies the central test machine that the repair session has finished. After being notified, the computer stops the running application in another node chosen randomly, which triggers again another repair. The running applications are stopped until there are only two nodes remaining.

This process is repeated 50 times for different generation sizes and symbol sizes. The size of the symbols and the generation sizes are chosen such that the size of the coded packet plus the coding vector is less than the size of the Ethernet frame (1472 bytes). This is done in order to prevent fragmentation of the UDP packets and being able to take advantages of

network coding in communications over lossy channels. The recorded data is then processed showing the results in the following sections.

5.1 Measurements of the Time Spent in the Repair Sessions

The measurements of the time spent in the different stages of the repairs sessions for the strategy MSR are shown in Figure 5.1

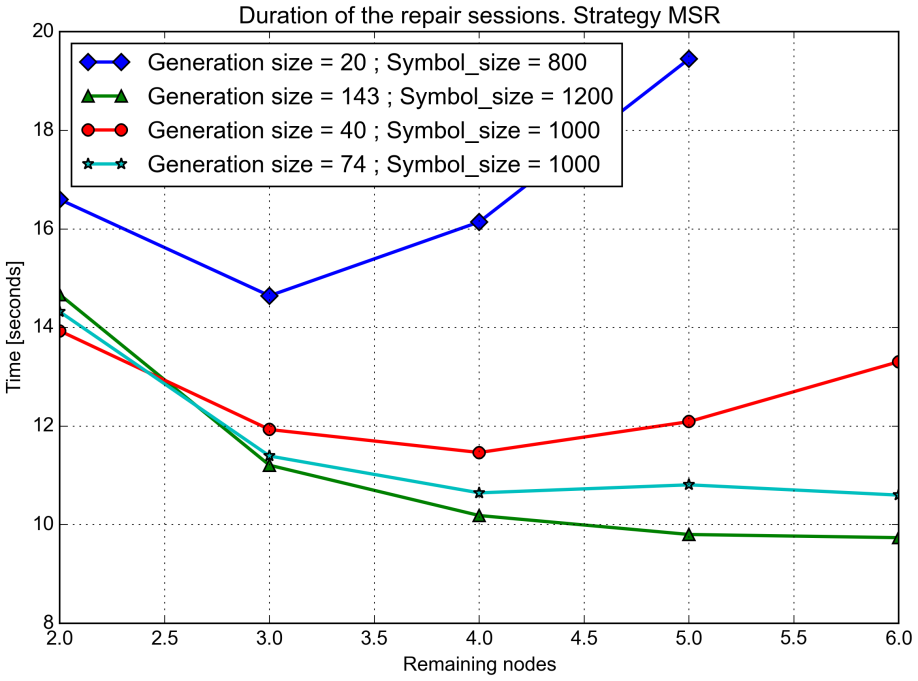

FIGURE 5.1: Total time spent in the repair sessions for different generation sizes and symbol sizes using MSR codes.

It is important to mention that for a generation size of 20 symbols, the curve in Figure 5.1 seems to be incomplete. This is simply because the uploader node distributes enough redundancy in the system so the first repair session is not necessary. One must remember that the values of α and γ are calculated per chunk, i.e., each chunk is seen as an individual file that needs to be repaired in the system. This means that for a generation size of 20 symbols, each packet contains 5% of the information of the file to repair.

The implemented system requires a long time to perform the repairs when the generation size is small. This is due the fact that we are using UDP protocol for our communications. When the generation size is small, the processing time for encoding packets is short causing that the nodes produce

and send a large amount of coded packets. All these coded packets cause a congestion in the network and as a consequence, the ACK messages experience a long delay before they can be received by the nodes sending the data packets.

This reduction of the goodput can be verified in Figure 5.2. It can be appreciated that the total number of transferred packets when the repair is performed in 5 nodes is around 1350 on average, which represents more than double of the calculated required packets for the repair session.

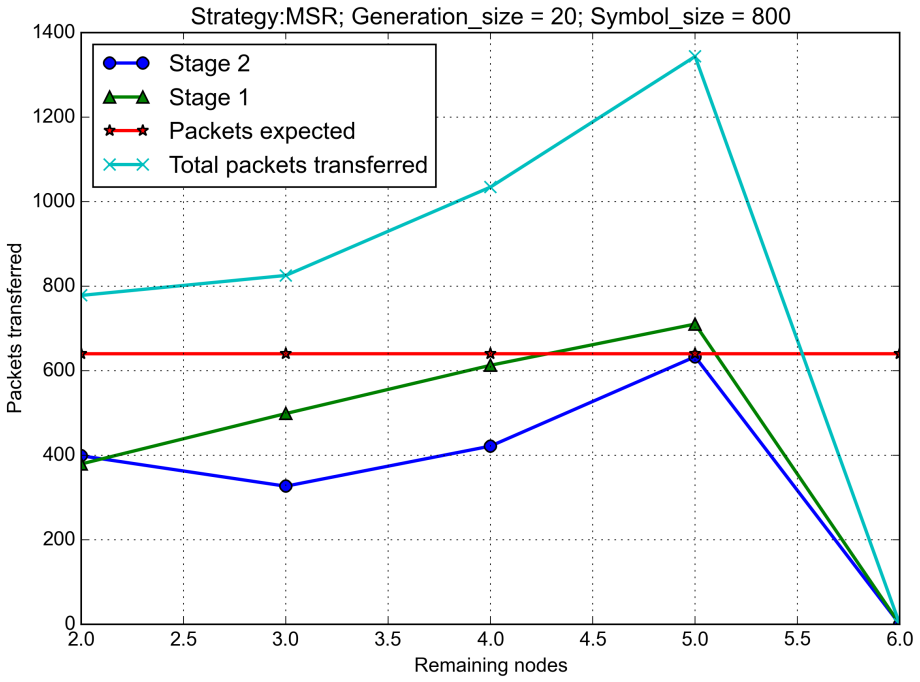

FIGURE 5.2: Total number of packets transferred in the repair sessions for different generation sizes and symbol sizes using MSR codes.

This issue with congestion is present also for the MBR strategy as shown in Figure 5.3. It can be seen that for smaller generations, the time required for the repair is longer than for larger generations. Nevertheless, for the MBR the effect of congestion is less dramatic when the number of remaining nodes is two. In Figure 5.3 it can be appreciated that the total repair time for all the generation sizes is almost the same when the repair session is performed only in two nodes.

For both strategies, the total required time for the repair sessions is shorter for larger generations when performed on a larger number of nodes. For instance, for the MSR strategy shown in Figure 5.1, when the generation sizes are of 143 packets, and the repair is performed on 6 nodes, then the repair is done approximately three seconds faster than when the generation size is of 40 symbols. This difference in time is more dramatic for the MBR

strategy. It can be seen in Figure 5.3 that the repair requires more than double of the time when the generation size is of 40 symbols than when it is of 143 symbols. It takes around 12 seconds and 5 seconds respectively.

When the repairs are performed by the last two nodes, then they take around the same amount of time for all generation sizes (except those affected dramatically by congestion). So, in Figure 5.1, the repair performed by two nodes using the MSR strategy lasts around 14 seconds for all generation sizes. Meanwhile, in Figure 5.3, it can be seen that this last repair takes around 11 seconds for all generation sizes.

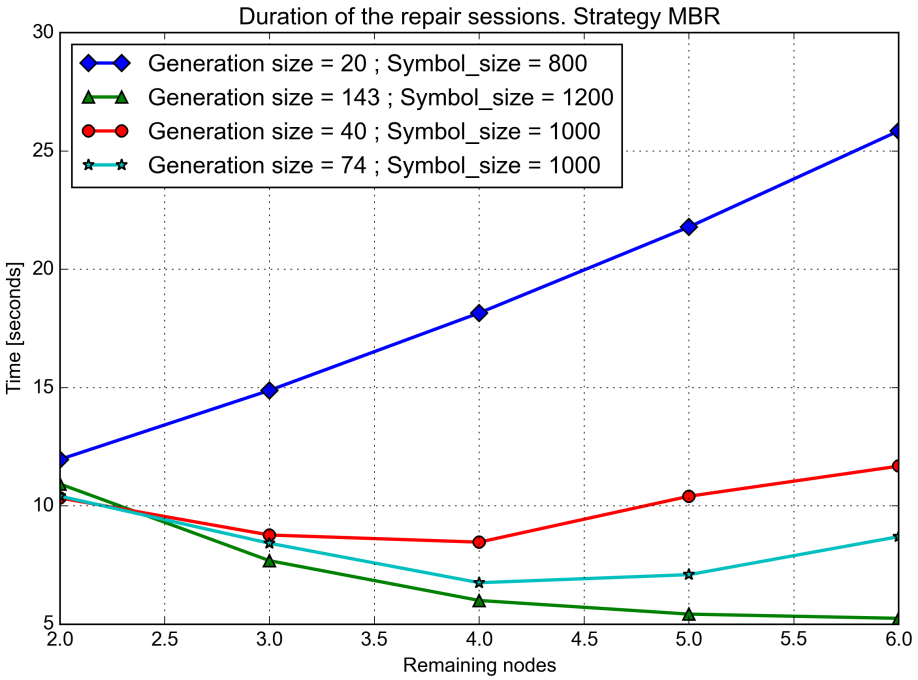

FIGURE 5.3: Total time spent in the repair sessions for different generation sizes and symbol size using MBR codes.

Figure 5.4 and Figure 5.5 show a comparison between the time required to complete each stage of the repair and the total time required for the entire repair session for both strategies when the generation sizes consist of 143 and 40 symbols respectively. In both figures it can be seen that, as expected, the total required time for all the repairs is longer in the MSR strategy than in the MBR strategy. The results were the expected, since given that more packets need to be transferred over the network then the time it takes for the repair sessions to be completed is longer.

It is interesting to notice that the time required for the first stage of the repair is almost constant for all the repair sessions using the MSR strategy. This result is appreciated for both generation sizes, and the value is close to 6.5 seconds in both cases.

Another interesting result is that for the MSR strategy the first stage of the repair takes longer than the second stage for almost all the repair sessions and for both generation sizes. On the other hand, for the MBR strategy, it is the other way around, i.e., for allmost all the repair sessions, the second stage takes longer than the first stage of the repair.

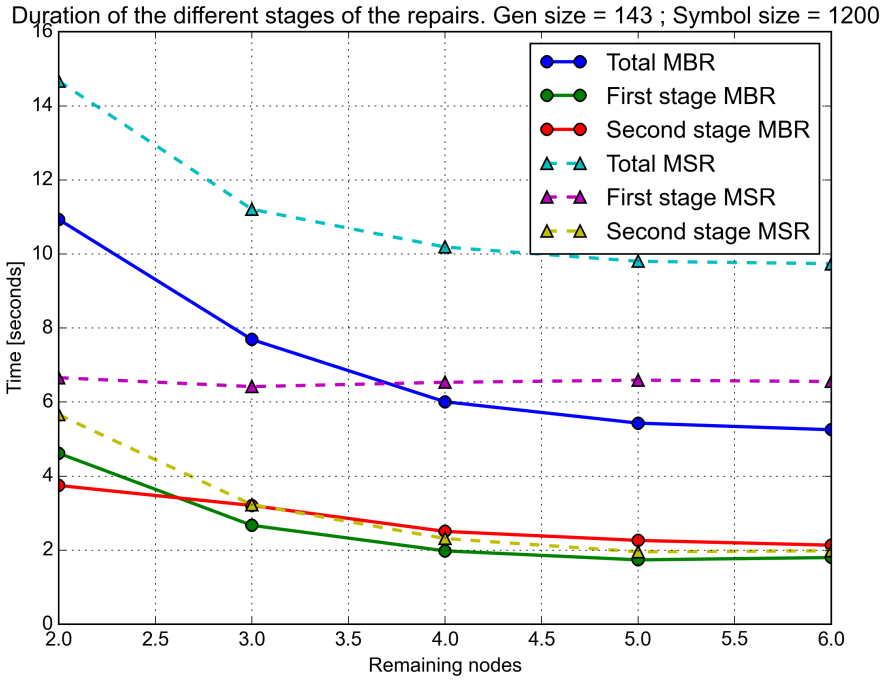

FIGURE 5.4: Comparison of the time spent in the different repair stages using MSR and MBR codes.

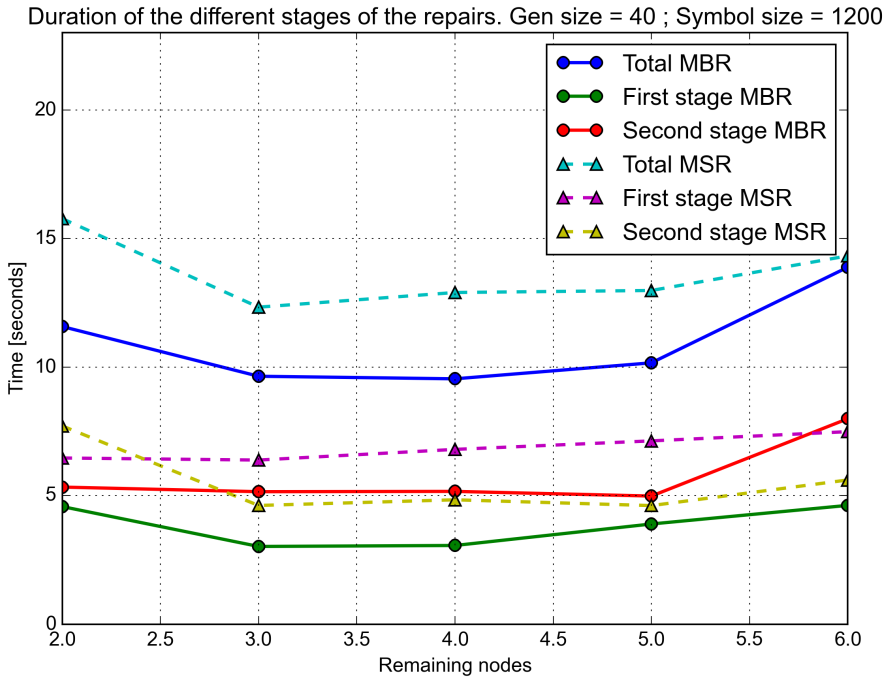

FIGURE 5.5: Comparison of the time spent in the different repair stages using MSR and MBR codes.

5.2 Measurements of the Packets Sent in the Repair Sessions

In Figure 5.6 and Figure 5.7 it is seen that the congestion of the network makes necessary to transfer many more packets than the necessary for the repair sessions. The effect of the congestion is more dramatic when the repairs are performed in a greater number of nodes. This is due the fact that since more nodes are sending packets, then the network presents greater delays due to congestion.

In Figure 5.8 and Figure 5.9 it is shown that the total number of packets sent during the repair sessions is closer to the calculated and expected. This is specially visible for the MSR strategy. For the MBR strategy, the protocol still have room for improvement, specially when the repair is performed in a greater number of nodes. For instance, when the first node disconnects, and 6 nodes remain, it should be transferred around 30% of the file over the network, however, our system transferred around 53% of the file. One solution for this issue would be to transfer at a lower data rate, specially when there are more nodes performing the repair. The current problem is that the nodes are sending packets faster than the receiver node can process them.

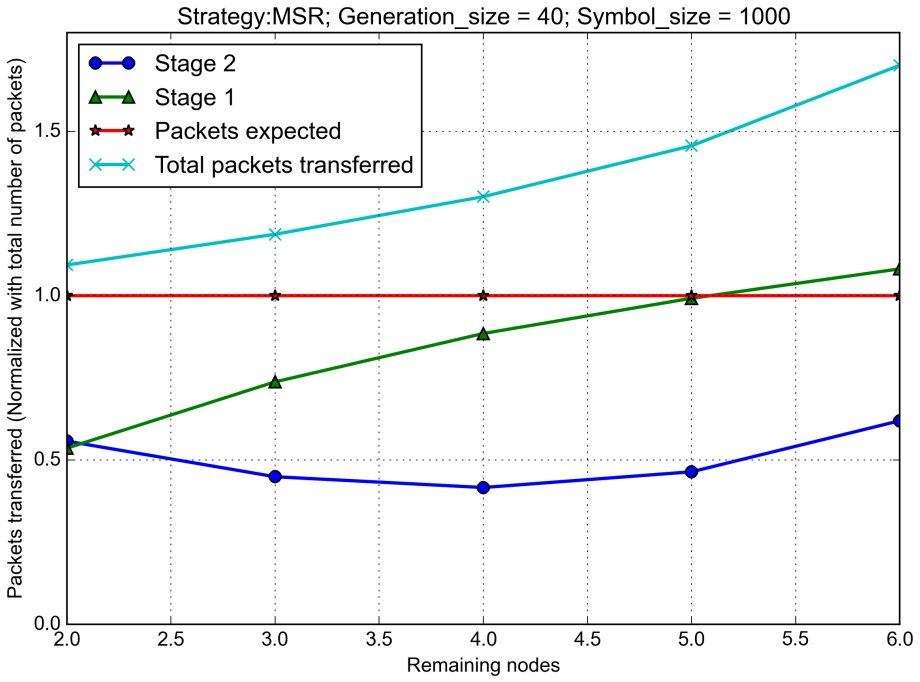

FIGURE 5.6: Total number of packets transferred (normalized) in the repair sessions MSR codes.

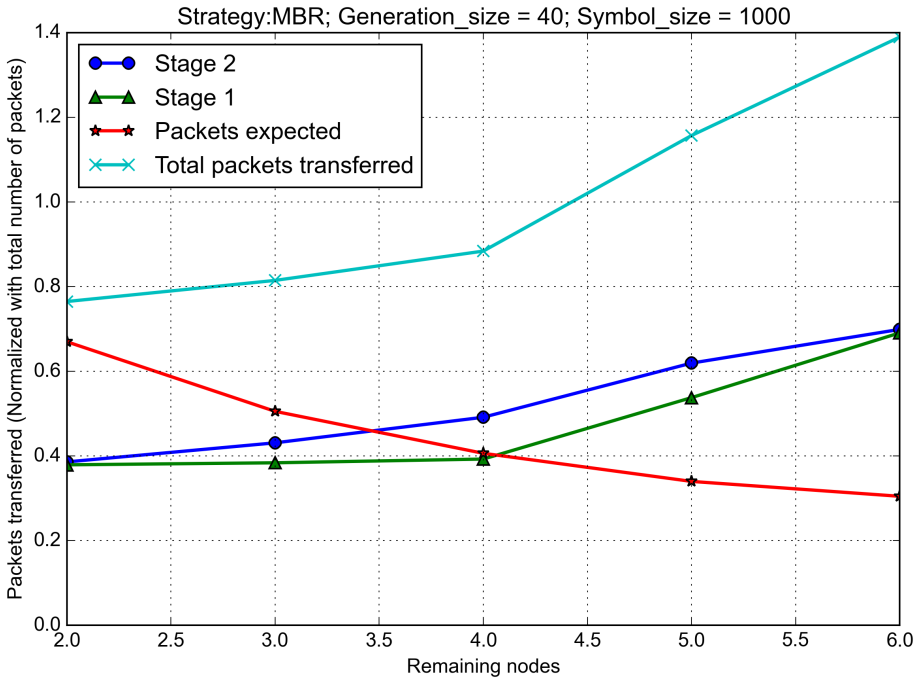

FIGURE 5.7: Total number of packets transferred (normalized) in the repair sessions using MBR codes.

In Figure 5.10 and Figure 5.11 it can be seen the consequences of the nodes sending packets faster than the speed at which the receiver nodes can

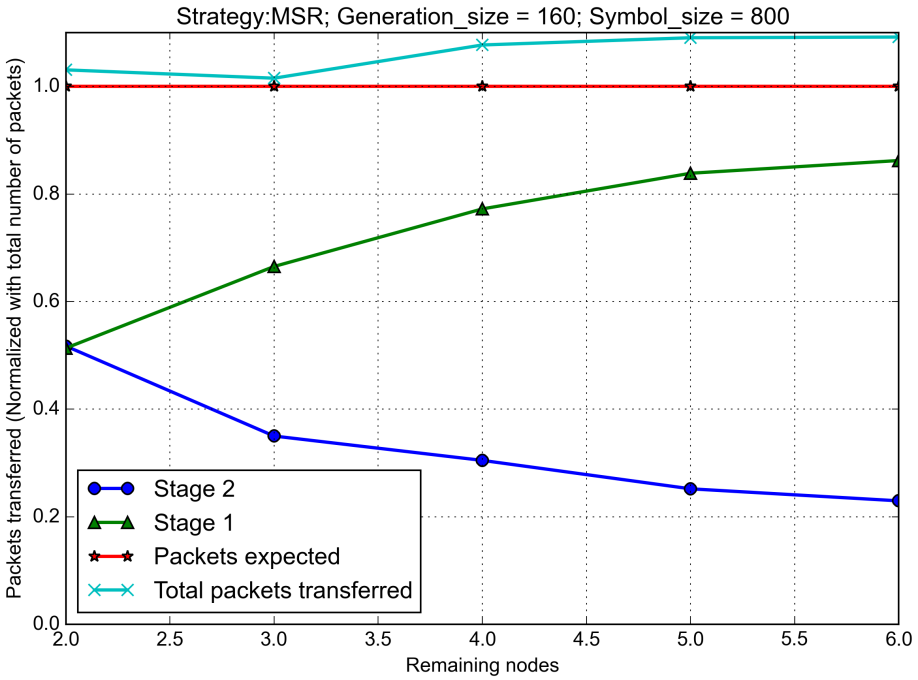

FIGURE 5.8: Total number of packets transferred (normalized) in the repair sessions using MSR codes.

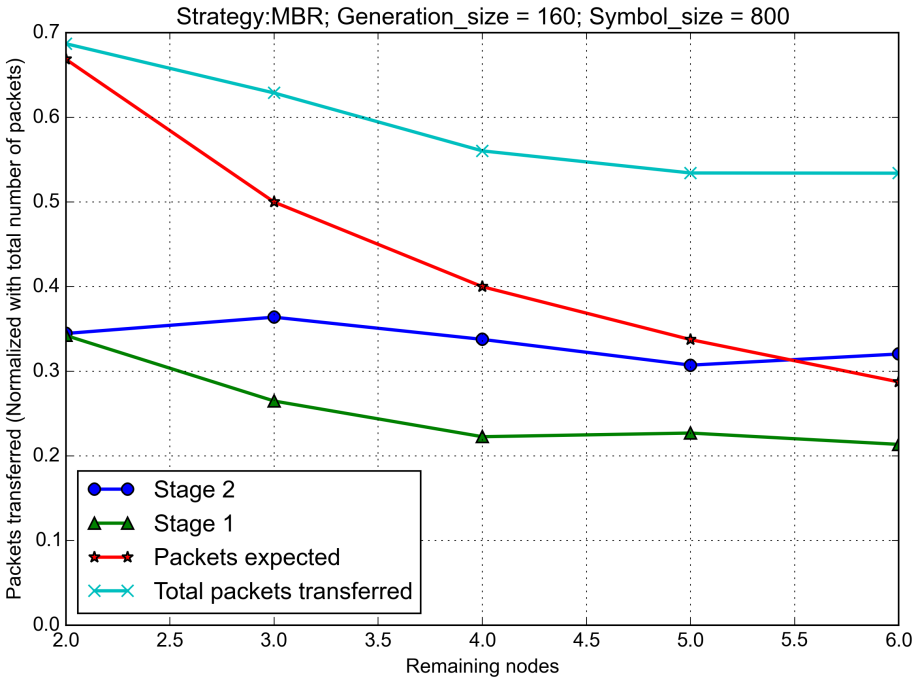

FIGURE 5.9: Total number of packets transferred (normalized) in the repair sessions using MBR codes.

process them. This produces the consequence of several ACK messages being transferred over the network. The reason for this, as described in Section

4.2, is that when the receiver node process a packet, and its recoder reaches the desired rank, it unicasts an ACK message to the sender node. The problem is that if the sender node transferred more packets in the meantime, then those packets are still in the queue of the receiver node.

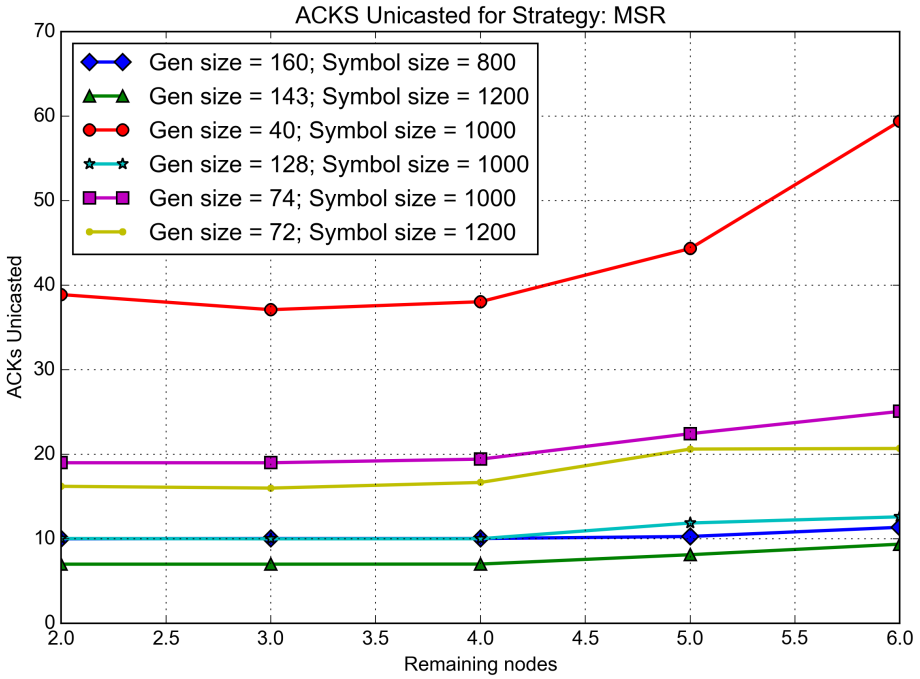

FIGURE 5.10: Total number of ACK messages unicasted in the different repair sessions using MSR codes.

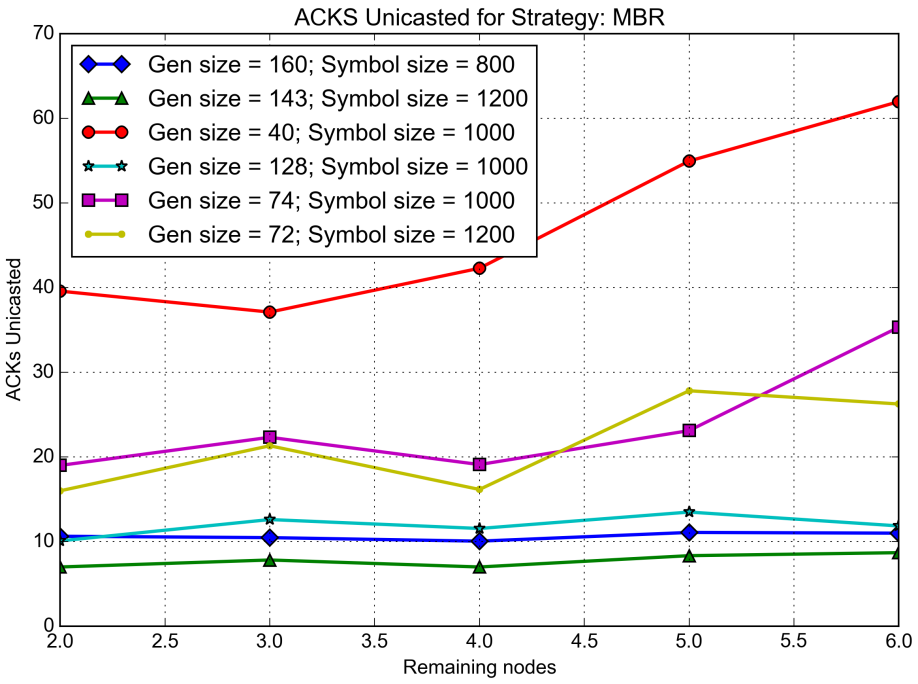

FIGURE 5.11: Total number of ACK messages unicasted in the different repair sessions using MBR codes.

Chapter 6

Conclusions and Future Work

The goal of this report has been to evaluate the plausibility of extending the state of the art results for standard repair of redundancy in distributed storage systems against node failures to the scenarios where the nodes are peers members of a P2P network. Through this report we evaluated the use of network coding as a common ground to manage the distributed storage of information and communications of peers with a single code structure.

In this project we studied the scenario of performing the repair of lost redundancy not just due to nodes failing and losing all the data that they were storing. Instead we focused on the need of repair due to homogeneous peers departing temporarily from the network. In this scenario we found that the protocols must be prepared to allow the nodes to build redundancy similarly to the standard state-of-the-art repair approaches. But we went beyond and considered that at the same time the protocols should allow the peers to destroy some of the extra redundancy built when departed nodes reconnect. We also found that it is necessary not just to build and destroy redundancy but also to redistribute it when a newcomer node joins the network or when a node that departed rejoins the system but it has less information than the peers maintaining the file reliable.

We designed a protocol and implemented an automated test scenario to show that a practical system that uses RLNC for generating MSR and MBR codes is achievable. Furthermore we measured and compared the time spent and the number of packets transfered for both types of codes in repair sessions and showed that even with the difficulties and flaws of a practical implementation, it is possible to observe a trade-off between storage and repair bandwidth as the theory suggested.

Moreover, we found that by taking advantage of the broadcast nature of the wireless channels, it is possible to reduce the average number of transmissions in repair sessions in P2P wireless networks, when compared with systems that use only unicast messages.

6.1 Future Work

Several problems need to be addressed in a practical implementation of a distributed storage system in P2P networks. For example, if UDP is used as the transport protocol, then it is necessary to implement a method for the congestion control. As it was shown in our results, congestion becomes a serious issue even when the number of nodes participating in the storage is small. The possibility of implementing network coding and the repair sessions by modifying an existing and robust open source transport protocol for file sharing in P2P network (such as uTorrent or Swift) should be evaluated.

Moreover studying the distribution of the processing operations in P2P networks where the nodes are heterogeneous is an interesting topic of research. For example, if powerful nodes in terms of processing power like personal computers share the network with mobile devices with less processing power or with a lifetime determined by its batteries, then it would be interesting to evaluate a modification in the protocol to choose the powerful devices as the leaders of the repair sessions.

The security in these distributed storage systems is another topic that must be addressed. If the users of the network are storing personal or sensitive information, encryption mechanisms must be implemented in the protocol to protect the system against malicious nodes, for example, eavesdroppers nodes.

Bibliography

[1] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in globally distributed storage systems. In *OSDI*, pages 61–74, 2010.

[2] Robert McMillan. (real) storm crushes amazon cloud, knocks out netflix, pinterest, instagram., 2012. URL `http://www.wired.com/2012/06/real-clouds-crush-amazon/`.

[3] P. Krishna Gummadi, Stefan Saroiu, and Steven D. Gribble. A measurement study of napster and gnutella as examples of peer-to-peer file sharing systems. *SIGCOMM Comput. Commun. Rev.*, 32(1):82–82, January 2002. ISSN 0146-4833. doi: 10.1145/510726.510756. URL `http://doi.acm.org/10.1145/510726.510756`.

[4] Jian Liang, Rakesh Kumar, and Keith W Ross. Understanding kazaa. *Manuscript, Polytechnic Univ*, page 17, 2004.

[5] Johan Pouwelse, Paweł Garbacki, Dick Epema, and Henk Sips. The bittorrent p2p file-sharing system: Measurements and analysis. In *Peer-to-Peer Systems IV*, pages 205–216. Springer, 2005.

[6] Chi-Jen Wu, Cheng-Ying Li, and Jan-Ming Ho. Improving the download time of bittorrent-like systems. In *Communications, 2007. ICC '07. IEEE International Conference on*, pages 1125–1129, June 2007. doi: 10.1109/ICC.2007.191.

[7] P. Sandvik and M. Neovius. The distance-availability weighted piece selection method for bittorrent: A bittorrent piece selection method for on-demand streaming. In *Advances in P2P Systems, 2009. AP2PS '09. First International Conference on*, pages 198–202, Oct 2009. doi: 10.1109/AP2PS.2009.39.

[8] Pietro Gonizzi, Gianluigi Ferrari, Vincent Gay, and Jérémie Leguay. Data dissemination scheme for distributed storage for iot observation systems at large scale. *Information Fusion*, 22:16–25, 2015.

[9] Alexandros G Dimakis, P Brighten Godfrey, Yunnan Wu, Martin J Wainwright, and Kannan Ramchandran. Network coding for distributed storage systems. *Information Theory, IEEE Transactions on*, 56(9):4539–4551, 2010.

[10] Alexandros G Dimakis, Kannan Ramchandran, Yunnan Wu, and Changho Suh. A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489, 2011.

[11] Hakim Weatherspoon and John D Kubiatowicz. Erasure coding vs. replication: A quantitative comparison. In *Peer-to-Peer Systems*, pages 328–337. Springer, 2002.

[12] Irving S Reed and Gustave Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial & Applied Mathematics*, 8(2):300–304, 1960.

[13] Thomas Mager, Ernst Biersack, and Pietro Michiardi. A measurement study of the wuala on-line storage service. In *Peer-to-Peer Computing (P2P), 2012 IEEE 12th International Conference on*, pages 237–248. IEEE, 2012.

[14] John Kubiatowicz, David Bindel, Yan Chen, Steven Czerwinski, Patrick Eaton, Dennis Geels, Ramakrishan Gummadi, Sean Rhea, Hakim Weatherspoon, Westley Weimer, et al. Oceanstore: An architecture for global-scale persistent storage. *ACM Sigplan Notices*, 35 (11):190–201, 2000.

[15] Barton Gellman and Ashkan Soltani. Nsa infiltrates links to yahoo, google data centers worldwide, snowden documents say, 2013. URL `http://www.washingtonpost.com/world/national-security/nsa-infiltrates-links-to-yahoo-google-data-centers-worldwide-snowden-documents-say 2013/10/30/e51d661e-4166-11e3-8b74-d89d714ca4dd_story.html`.

[16] Barton Gellman, Ashkan Soltani, and Todd Lindeman. How the nsa is infiltrating private networks, 2013. URL `http://apps.washingtonpost.com/g/page/world/how-the-nsa-is-infiltrating-private-networks/542/`.

[17] Szymon Acedanski, Supratim Deb, Muriel Médard, and Ralf Koetter. How good is random linear coding based distributed networked storage. In *Workshop on Network Coding, Theory and Applications*, pages 1–6, 2005.

[18] Supratim Deb, Michelle Effros, Tracey Ho, David R Karger, Ralf Koetter, Desmond S Lun, Muriel Médard, and Niranjan Ratnakar. Network coding for wireless applications: A brief tutorial. IWWAN, 2005.

[19] Yuchong Hu, Henry CH Chen, Patrick PC Lee, and Yang Tang. Nccloud: applying network coding for the storage repair in a cloud-of-clouds. In *FAST*, page 21, 2012.

[20] R. Schollmeier. A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications. In *Peer-to-Peer Computing, 2001. Proceedings. First International Conference on*, pages 101–102, Aug 2001. doi: 10.1109/P2P.2001.990434.

[21] Ralf Steinmetz and Klaus Wehrle, editors. *Peer-to-peer systems and applications*. Lecture notes in computer science. Springer, Berlin, New York, 2005. ISBN 3-540-29192-X.

[22] Rüdiger Schollmeier and Gerald Kunzmann. Gnuviz–mapping the gnutella network to its geographical locations. *Praxis der Informationsverarbeitung und Kommunikation*, 26(2):74–79, 2003.

[23] Xuemin Shen, Heather Yu, John Buford, and Mursalin Akon. *Handbook of peer-to-peer networking*. Springer, New York, NJ, London, 2010. ISBN 978-0-387-09750-3.

[24] PUB FIPS. 180-4. *Federal Information Processing Standards Publication, Secure Hash*, 2011.

[25] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Middleware 2001*, pages 329–350. Springer, 2001.

[26] Ion Stoica, Robert Morris, David Liben-Nowell, David R Karger, M Frans Kaashoek, Frank Dabek, and Hari Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *Networking, IEEE/ACM Transactions on*, 11(1):17–32, 2003.

[27] Ernesto Van der Sar. The pirate bay tracker shuts down for good, 2009. URL `http://torrentfreak.com/the-pirate-bay-tracker-shuts-down-for-good-091117/`.

[28] R Vliegendhart. *Swarm Discovery in Tribler using 2-Hop TorrentSmell*. PhD thesis, TU Delft, Delft University of Technology, 2010.

[29] Rob Waugh. Tribler: New file-sharing technology is immune to government attacks, 2012. URL `http://www.dailymail.co.uk/sciencetech/article-2098759/`

`Tribler-New-file-sharing-technology-IMMUNE-government-attacks.html`.

[30] Shawn Wilkinson and Jim Lowry. Storj: Decentralized autonomous file storage.

[31] Yunnan Wu, Alexandros G Dimakis, and Kannan Ramchandran. Deterministic regenerating codes for distributed storage. In *Allerton Conference on Control, Computing, and Communication*. Citeseer, 2007.

[32] Yunnan Wu. Existence and construction of capacity-achieving network codes for distributed storage. *Selected Areas in Communications, IEEE Journal on*, 28(2):277–288, 2010.

[33] S-YR Li, Raymond W Yeung, and Ning Cai. Linear network coding. *Information Theory, IEEE Transactions on*, 49(2):371–381, 2003.

[34] Ralf Koetter and Muriel Médard. An algebraic approach to network coding. *Networking, IEEE/ACM Transactions on*, 11(5):782–795, 2003.

[35] Muriel Médard and Alex Sprintson. *Network coding: fundamentals and applications*. Academic Press, 2012.

[36] Tracey Ho, Muriel Médard, Ralf Koetter, David R Karger, Michelle Effros, Jun Shi, and Ben Leong. A random linear network coding approach to multicast. *Information Theory, IEEE Transactions on*, 52(10):4413–4430, 2006.

[37] Janus Heide, Morten Videbæk Pedersen, Frank HP Fitzek, and Muriel Médard. On code parameters and coding vector representation for practical rlnc. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–5. IEEE, 2011.

[38] Oscar Trullols-Cruces, Jose M Barcelo-Ordinas, and Marco Fiore. Exact decoding probability under random linear network coding. *Communications Letters, IEEE*, 15(1):67–69, 2011.

[39] Márton Sipos, Frank Fitzek, D Lucani, and Morten Videbæk Pedersen. Distributed cloud storage using network coding. In *IEEE Cons. Comm. and Netw. Conf*, pages 6–19, 2014.

[40] Lorenzo Vicisano, Michael Luby, and Mark Watson. Forward error correction (fec) building block. 2007.

[41] John Torjo. *Boost. Asio C++ Network Programming*. Packt Publishing Ltd, 2013.

[42] Christopher Kohlhoff. Thinking asynchronously: Designing applications with boost.asio. Presented at the BoostCon (Boost Libraries Conference), 2011.

[43] Robert G. Gallager, Pierre A. Humblet, and Philip M. Spira. A distributed algorithm for minimum-weight spanning trees. *ACM Transactions on Programming Languages and systems (TOPLAS)*, 5(1):66–77, 1983.