Network Coding for Distributed Cloud, Fog and Data Center Storage

Sipos, Marton A.

DOI (link to publication from Publisher):
10.5278/vbn.phd.tech.00047

Publication date:
2018

Document Version
Publisher's PDF, also known as Version of record

NETWORK CODING FOR DISTRIBUTED CLOUD, FOG AND DATA CENTER STORAGE

ELOSZTOTT ADATTÁROLÁS NETWORK CODING TÁMOGATÁSSAL CLOUD, FOG ÉS ADATKÖZPONTI KÖRNYEZETBEN

BY
MÁRTON ÁKOS SIPOS

DISSERTATION SUBMITTED 2018

AALBORG UNIVERSITY
DENMARK

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Aalborg University
Technical Faculty of IT and Design
Department of Electronic Systems

NETWORK CODING FOR DISTRIBUTED CLOUD, FOG AND DATA CENTER STORAGE

ELOSZTOTT ADATTÁROLÁS NETWORK CODING TÁMOGATÁSSAL CLOUD, FOG ÉS ADATKÖZPONTI KÖRNYEZETBEN

Ph.D. Dissertation

Márton Ákos Sipos

Advisors:
Hassan Charaf, Ph.D.
Daniel Enrique Lucani Roetter, Ph.D.

Budapest, 2018.

Márton Ákos Sipos

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

1117 Budapest, Magyar Tudósok körútja 2. QB-207.

Aalborg University
Technical Faculty of IT and Design
Department of Electronic Systems

DK-9220 Aalborg, Fredrik Bajers Vej 7B

e-mail: siposm@aut.bme.hu,
maaks@es.aau.dk
tel: +36(1)4632870
fax: +36(1)4632871

Advisors:

Budapest: Hassan Charaf, Ph.D.
Aalborg: Gilberto Berardinelli, Ph.D.
Daniel Enrique Lucani Roetter, Ph.D.
Frank H. P. Fitzek, Ph.D.
Morten V. Pedersen, Ph.D.

Dissertation submitted: July, 2018

PhD supervisors: Prof. Hassan Charaf
Budapest University of Technology and Economics

Associate Prof. Gilberto Berardinelli
Aalborg University

Previous PhD Supervisor: Associate Prof. Daniel Enrique Lucani Rötter
Aarhus University

Prof. Frank H. P. Fitzek
Dresden University of Technology

Associate Prof. Morten V. Pedersen
Aalborg University

PhD committee: Professor Jan Østergaard (chairman)
Aalborg University

Professor Pascal Felber
University of Neuchâtel

Professor Janos Levendovszky
Budapest University of Technology and Economics

PhD Series: Technical Faculty of IT and Design, Aalborg University

Institut: Department of Electronic Systems

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-177-4

Published by:
Aalborg University Press
Langagervej 2
DK – 9220 Aalborg Ø
Phone: +45 99407140
aauf@forlag.aau.dk
forlag.aau.dk

Printed in Denmark by Rosendahls, 2018

Abstract

As the amount of data created and consumed each day increases at an alarming rate, distributed storage systems have turned towards erasure coding to keep up with demand. Several well-known codes exist that decrease storage costs significantly, but introduce new challenges that limit their use. Furthermore, there is no single technique that is widely applied across different types of systems. One technique that may be an exception due to its flexibility is network coding.

We set out to look at how network coding can be applied in three distinct scenarios ranging from the mostly static data center environment to the highly dynamic fog computing setting. We also looked at how aggregating multiple cloud storage services can alleviate some of the issues related to single–cloud solutions and examined the problem of updating data after it has been erasure coded. We used network flows to model repair and reconstruction processes, followed by results from measurements and simulations to validate theory. When looking at erasure coding for fog computing, we used real–world traces to evaluate its feasibility. We assessed the effectiveness of our proposed solution on updating erasure coded data using git and a publicly available repository.

We proposed techniques to reduce the burden of repairs by up to 35% on data center networks and presented a system of checks for network coding that determines whether a repair maintains the required level of reliability. We showed that aggregating multiple cloud storage services and applying network coding improves retrieval performance by between 34%–61% and alleviates some of the the reliability and privacy concerns related to single–cloud solutions. We proposed a schema that adapts data distribution with the goal of reducing retrieval time. For fog computing, we compared different erasure codes on their ability to maintain the integrity of an edge cloud of mobile devices. We also touched on predicting node availability and showed that mobile storage clouds are feasible with as little as 25% redundancy. Finally, we proposed a mechanism to solve a challenge common across most storage systems, updating erasure coded data after it has changed. Given realistic update patterns, it requires 5 orders of magnitudes less storage compared to state-of-the-art solutions.

Our hope is that the results presented in this dissertation advance the state of the art by enabling erasure coding to be applied in more cases, more effectively.

Összefoglaló

Modern világunk adattermelési és fogyasztási szokásai fokozatosan növekvő terhet rónak az elosztott tároló rendszerekre. Ennek csökkentésére egy kézenfekvő megoldás a hibajavító kódok használata. Több olyan eljárás ismert, amely lényegesen csökkenti a tárolandó adat mennyiségét, de ezek gyakran újabb kihívásokat teremtenek, ráadásul nem létezik univerzálisan alkalmazott megoldás. Rugalmasságának köszönhetően kivételt jelenthet a network coding.

Célul tűztük ki a network coding vizsgálatát három alkalmazási területen: megvizsgáltuk jellemzőit a többnyire statikus adatközpontoktól a többfelhős rendszereken keresztül a dinamikus fog computing megközelítésig. Továbbá, foglalkoztunk a kódolt módon tárolt adatok frissítésével. A javítási és rekonstruálási folyamatokat hálózati folyamokkal modelleztük, majd szimulációs és mérési eredményekkel validáltuk. A fog computing kivitelezhetőségének vizsgálatára valós rendszerből kinyert adatokat használtunk, akárcsak a kódolt adatok frissítésénél, ahol a gitre és egy nyilvánosan elérhető kódbázisra támaszkodtunk.

Javasoltunk egy hibajavító technikát adatközponti használatra, amely akár 35%-al csökkenti a hálózat terhelését azzal, hogy elkerüli a túlterhelt részeit. Továbbá bemutattunk egy ellenőrző mechanizmust, amely meghatározza, hogy egy bizonyos javítás megfelel–e a szükséges megbízhatósági kritériumoknak. Mérésekkel igazoltuk, hogy a network coding használata egy több felhőből álló tárhelyszolgáltatás esetén csökkenti az adatátviteli időt 34%–61%–al, valamint megbízhatóbb és biztonságosabb rendszerekhez vezet egy hagyományos egy felhős megoldáshoz képest. Továbbá javasoltunk egy adaptív technikát a rendszer teljesítményének növelésére, amely módosítja az adatok elosztását. A fog computing terén hibajavító kódokat hasonlítottunk össze azon képességük szerint, hogy milyen mértékben biztosítják az adatok megbízható tárolását. Megkíséreltük megbecsülni a rendszerben szereplő mobilkészülékek rendelkezésre állását és beláttuk, hogy ezen rendszerek már 25%–os redundancia esetén is megvalósíthatóak. Végül javasoltunk egy eljárást a hibajavító kóddal ellátott adatok frissítésére. Valós mintákon végzett mérések alapján megállapítottuk, hogy 5 nagyságrenddel kevesebb tárhelyet ígényel az elterjedt megoldáshoz képest.

Reményeink szerint az eredményeink segíteni fogják a korszerű rendszerek tervezését és lehetővé teszik a hibajavító kódok hatékonyabb, szélesebb körben történő alkalmazását.

Resumé

Da mængden af data oprettet og lageret hver dag stiger med en alarmerende hastighed, har distribuerede lagringssystemer vendt sig imod fejlkorrigerende koder for at kunne effektivisere og følge med udviklingen. Der findes flere kendte koder, der reducerer lageromkostningerne betydeligt, men disse introducerer nye udfordringer der begrænser deres brug. Desuden er der ingen kode der alene kan anvendes bredt, på tværs af forskellige typer systemer. Én kode der kan være undtagelsen, på grund af sin fleksibilitet, er netværkskodning.

I dette projekt undersøgte vi hvordan netværkskodning kan anvendes i tre forskellige scenarier lige fra det mest statiske data center miljøtil de meget dynamiske "fog computing" miljøer. Vi undersøgte også hvordan aggregering af flere data cloud lagringstjenester kan afhjælpe nogle af problemerne i forbindelse med anvendelsen af én enkelt cloud løsning. Vi undersøgte derudover problemet med opdatering af data, efter den er blevet kodet med en fejlkorrigende kode.

Vi brugte netværkstrafik til at modellere reparations- og genopbygnings-processer, efterfulgt af resultater fra må linger og simuleringer til validering af teori. Når vi kiggede på fejlkorrigerende koder til anvendelse "fog computing", brugte vi realistiske målinger til at vurdere dens anvendelighed. Vi vurderede effektiviteten af vores foreslåede løsning til opdatering af data kodet med en fejlkorrigende kode ved hjælp af git og et offentligt tilgængeligt git repository. Vi har foreslået teknikker der kan reducere belastningen af netværket i forbindelse med reparationer med op til 35% i datacentre, og fremlagt ét system til kontrol af netværkskodning, der afgør, om en reparation opretholder det krævede niveau af pålidelighed. Vi viste at aggregering af flere cloud lagringstjenester, samt at anvende netværkskodning forbedrer systemets ydelsens i forbindelse med adgang til lageret data med 34 % - 61 % og mindsker nogle af de pålidelighed og privatlivs problemer, der er relateret til anvendelsen af én enkelt cloud løsning. Vi foreslog et skema, der tilpasser datadistribution med det formål at reducere adgangstiden. Til "fog computing" sammenlignede vi forskellige fejlkorrigerende koder med deres evne til at opretholde integriteten af en "edge cloud" af mobile enheder. Vi har også kigget på hvordan man kunne forudsige tilgængeligheden af en enkelt mobile enhed og viste, at mobile clouds er mulige med så lidt som 25% redundans. Endelig foreslog vi en mekanisme til at løse en udfordring, der er almindelig på tværs af de fleste lagringssystemer, nemlig at opdaterer data kodet med en fejlkorrigende kode, efter at den er ændret. Baseret på realistiske opdaterings mønstre kræver det 5 størrelsesordner mindre lagerplads i forhold til de nyeste løsninger.

Vores håb er, at de resultater, der præsenteres i denne afhandling, fremmer den nyeste teknologi ved at muliggøre brugen af fejlkorrigerende koder i flere tilfælde områder end hvad der i dag er tilfældet.

Preface

Acknowledgement

I am grateful for their supporting love to Kata, my family from Brassó and Budapest and my friends. It might sound like a cliché, but I would not have made it this far without them.

During my time at BME I had Hassan as both my advisor and mentor. I am certain that he plays a big part in how I am just as happy to set out each morning for the university today as when I started. I would like to thank Luca, Péter and Kristóf for their continual and helpful support, Patrik for the great discussions, Professors István Vajk and János Levendovszky for their contributions to increasing the scientific value of my work.

During my time in Aalborg I have relied on Daniel for almost everything. I am grateful for many things, both work-related and personal, most importantly for looking out and always having my best interests at heart. I am thankful for Frank for giving me a strong start and a push in the right direction, Morten and Péter for helping me settle into life in this at first strange world.

During my year in Boston I got to know an interesting and exciting new world. I learned a lot from my colleagues, Atif, Dave, Josh, Narayan and Praveen. I am grateful that they supported my research and made it possible for me to travel and get a taste of the American pie. It is thanks to them and the friends from hiking that I felt at home very quickly on the other side of the pond.

Last but not least I want to thank the Csibi Sándor Foundation, their scholarship provided me with encouragement and financial support to finish my doctoral studies. The scholarship I received from the Új Nemzeti Kiválóság Program helped me in putting the final touches to this dissertation.

Köszönet

Hálás vagyok támogató szeretetéért feleségemnek, Katának, brassói és budapesti családtagjaimnak, barátaimnak. Noha közhelyként hangozhat, de nélkülük nehezen jutottam volna el idáig.

A Műegyetemen töltött időm során Hassan egyszerre volt konzulensem és mentorom. Úgy érzem nagy része van abban, hogy ma is ugyanolyan szívesen veszem reggelente az irányt az egyetem felé, mint tanulmányaim elején. Meg szeretném köszönni Lucának, Péternek és Kristófnak rendszeres segítségüket a PhD-s mindennapokban, Patriknak a remek szakmai vitákat, Vajk István és Levendovszky János Professzor Uraknak a munkám szakmai színvonalának növelésére tett javaslataikat.

Az Aalborgban töltött időm során Danielre támaszkodtam szinte mindenben. Szakmailag és emberileg is sokat köszönhetek neki, amiért mindig az én érdekeimet helyezte előtérbe. Hálás vagyok Franknek az erőteljes kezdeti löketért, remek iránybahelyezésért, valamint Mortennek és Péternek, amiért segítettek elhelyezkedni ebben a kezdetben idegen világban.

A Bostonban töltött egy év során egy nagyon érdekes és izgalmas új világgal ismerkedtem meg. Kollégáimtól, Atiftól, Davetől, Joshtól, Narayantól és Praveentől rengeteget tanultam. Hálás vagyok, amiért támogatták a kutatásom és lehetővé tették, hogy utazhassak és belekóstoljak az amerikai élménybe. Nekik és a kirándulásokon szerzett barátoknak köszönhetem, hogy nagyon hamar otthon éreztem magam az óceán túlsó partján is.

Végső, de nem utolsósorban meg szeretném köszönni a Csibi Sándor Alapítványnak, mely ösztöndíjával bátorítást és pénzügyi támogatást biztosított a doktori tanulmányaim befejezésére. A doktori iskola abszolválása után az Új Nemzeti Kiválóság Program ösztöndíja segített befejezni a disszertációt.

Statement

I, Márton Ákos Sipos, hereby declare that the content of this thesis is a product of my own and my coauthors', original work except where explicitly stated otherwise. All parts originating from sources other than our own have been clearly marked.

Nyilatkozat*

Alulírott Sipos Márton Ákos kijelentem, hogy ezt a doktori értekezést magam készítettem, és abban csak a megadott forrásokat használtam fel. Minden olyan részt, amelyet szó szerint, vagy azonos tartalomban, de átfogalmazva nem saját forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Budapest, 2018. február 15.

(Sipos Márton Ákos)

*A bírálatok és a védésről készült jegyzőkönyv a későbbiekben a Dékáni Hivatalban elérhetőek.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Our society creates and consumes more data than ever before. This trends looks to accelerate as more and more IoT devices come online in the following years. Keeping data safe from malicious entities, available and quickly accessible is a daunting task. To achieve these goals, several copies can be created and distributed across several locations that are unlikely to fail at the same time. However, a more cost–effective solution is to use erasure coding to protect against partial losses. The idea is several decades old, yet it's adoption rate is somewhat limited in many scenarios by the challenges it poses.

I have identified three environments of interest, namely *data centers*, *aggregated cloud storage services* and *fog computing*, then set out to look at the most pressing and interesting challenges in each of them. Figure 1.1 sums up their most distinctive features.

Much of the research on erasure–coded storage has so far focused on data centers, a mostly static scenario where the number of nodes does not change. Erasure codes are employed in many existing systems, but tend to be applied only to a small part of the data. This is the case even if a large proportion of data is rarely accessed archival storage that is a prime candidate for erasure coding. The network traffic associated with repairing data on failed storage nodes can overload network devices and cause the system to fail in meeting its performance criteria. I wish to find a solution to reduce this burden by taking advantage of information about the network's topology and state. Since it is hard to predict what network topology future systems will use, I seek a general solution.

Cloud storage services are popular solutions for both enterprises and end users. However, single–cloud solutions have some limitations that hinder their growth. The perceived reliability of cloud providers is hurt by how even short, small–scale failures can lead to several services becoming unavailable or slow. Many of these events, such as the Netflix outage that took place during Christmas 2012, the difficulty in reaching the Healthcare.gov

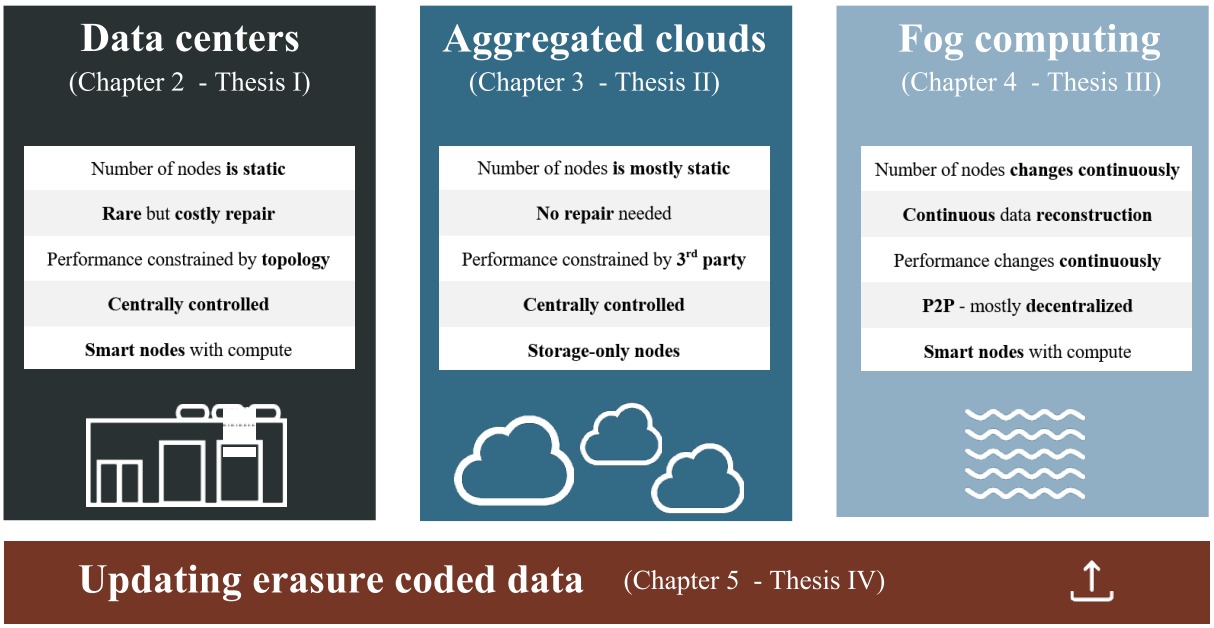


Figure 1.1: Overview of the main subjects of the dissertation

website at its launch, the Exchange Online availability issues in June 2014 or the AWS outage in February 2017 have received widespread media coverage. Likewise, high–profile hacking scandals, such as that involving iCloud in August 2014, have questioned the security of single–cloud solutions. I propose creating an aggregated solution to address some of these challenges. Like data centers, an aggregated cloud storage solution is a mostly static scenario where the number of storage nodes rarely changes. However, since data must cross the Internet, the speed with which it can be retrieved changes continuously. Designing and adapting the data distribution to account for this is a key enabler in improving performance. I also seek to take into account other important factors for the consumers of these services and aim to increase data availability and security, while reducing costs.

Most current online services are highly centralized, backed by reliable but expensive data centers. However, a move towards decentralization can be observed for services that involve transferring a lot of shared data to the end user. Several content distribution and streaming services already cache data at the edge of the network. I envision fog computing as the next step in this regard, a cloud of inherently unreliable mobile devices that act as an integral part of the infrastructure of the service. Since storage nodes join and leave the system continuously, erasure codes face the added challenge of providing enough slack to account for the variation in the number of nodes. It is not clear which codes are suitable in this regard or more importantly, which ones can reconstruct data on leaving nodes most efficiently. Indeed, even whether such a system is feasible is an open question.

Finally, one of the main obstacles in making erasure coding a more pervasive technology in distributed storage, regardless of scenario, is the difficulty in working with mutable data. If an erasure-coded file changes slightly, current systems simply delete the old file and store the new version. A method for storing and applying encoded differences similarly to how today's version control systems work would expand the scenarios where erasure coding is used beyond archival data.

A particularly interesting technology due to its flexibility, efficiency and ability to work with limited coordination between the elements of a system is network coding. One of the goals of this work is to establish whether its most practical implementation, Random Linear Network Coding (RLNC), has enough benefits in the aforementioned environments to justify its use.

This dissertation sets out to answer the following questions:

- How can the burden of repairs on storage networks be reduced? Can awareness of topology and conditions help in this regard?
- How can coefficients be checked efficiently for linear dependence for codes such as RLNC?
- What are the benefits to using RLNC for distributed cloud storage in terms of reliability, retrieval performance, security and privacy?
- Is it possible to adapt the distribution of data to increase retrieval performance? If so, what is the network cost of the adaptation?
- Can a decentralized, P2P cloud of mobile devices that regularly leave and join be used for storing data? If so, what erasure codes ensure data survival using the least amount of storage and network bandwidth? How does this change if the cloud contains high availability devices or if it is centrally controlled?
- Is it possible to update erasure–coded data? If so, how does a coded solution compare to traditional systems on storage efficiency?

1.2 Related work

1.2.1 Erasure coding in distributed storage systems

In recent years, distributed storage systems (DSS) have seen a trend towards erasure coding as a means to control the costs of storing and ensuring the resilience of large volumes of data. Even though traditional distributed storage systems employ replication [Shvachko et al., 2010], erasure coding provides equivalent or better resilience while

using a fraction of the raw storage capacity. Both new technologies and increasing storage volume requirements suggest that erasure coding will continue to increase in importance as a factor in data center design. Offloading of encode and decode operations to GPUs, FPGAs and/or the use of modern software libraries such as ISA-L [Intel, 2017], jerasure [Plank and Greenan, 2014] and Kodo [Pedersen et al., 2011] promises to lower the computation costs of these operations, potentially expanding the set of cost–effective use cases for erasure coded storage. Additionally, the increased IOP density* and IO bandwidth of next generation storage devices, such as NVMe (Non–Volatile Memory Express), as compared with rotating media or earlier SSD devices, promises to lower the effective IO costs associated with coded storage, further expanding the set of use cases.

Network interfaces have arguably seen a less dramatic increase in throughput than either storage or compute. A large bulk of the research in the area of erasure coding for distributed storage has focused on ameliorating the increased strain on the network during repair of lost erasure–coded data by reducing the amount of data transferred, commonly referred to as repair bandwidth. Unlike replicated storage where data can be recovered by simply copying the lost pieces (packets) from surviving nodes, repairing erasure coded pieces involves retrieving significantly more data. For example, Reed–Solomon (RS) [Reed and Solomon, 1960] is widely employed due to its optimal storage efficiency for a given level of reliability. More precisely, it is a Maximum Distance Separable (MDS) code. However, repairing lost pieces requires as many coded pieces as are required to recover the original data. To address this, codes with more efficient repair have emerged employing techniques such as functional repair [Dimakis et al., 2007], interference alignment [Wu and Dimakis, 2009], piggybacking [Rashmi et al., 2013b] and subpacketization [Guruswami and Wootters, 2015].

1.2.2 The trade–off between storage and repair efficiency

Dimakis et al. characterize in [Dimakis et al., 2007, Dimakis et al., 2010] the inherent trade–off between storage efficiency and repair bandwidth. Their seminal work introduced regenerating codes, a class of codes based on concepts from network coding that achieve various points on the resulting trade–off curve. The two extremal points on the curve are of particular interest. Minimum Storage Regenerating (MSR) codes need the least amount of storage to ensure a given level of reliability and are therefore equivalent with Maximum Distance Separable (MDS) codes in this regard. Conversely, Minimum Bandwidth Regenerating (MBR) codes store more information in order to decrease the amount of information transmitted during reconstruction to a minimum.

*IOP density is calculated by dividing the number of input/output operations a storage device can sustain by its useful storage capacity.

In [Shah et al., 2010], Shah et al. introduced the concept of flexible regeneration, where storage nodes contribute different amounts of data to repairs and introduced a lower bound on the total repair bandwidth. We introduce a slightly simpler lower bound for MSR codes in Chapter 2 that forms the basis of our theoretical results for making functional repair network–aware. We have decided against the use of a cap on the amount of data transferred from any single node, as argued for in [Shah et al., 2010], as we felt it unduly excludes some repair strategies. The same research team has provided [Shah et al., 2012] a code for the MBR point that uses a form of exact repair (i.e. lost pieces are replaced with identical copies) termed 'repair–by–transfer'. This makes it interesting in severely bandwidth and computationally–limited systems. We refer to this code as RBT–MBR and include it in our evaluation. The paper also establishes the non–achievability of most of the interior points on the curve for exact repair. [Rashmi et al., 2015] proposed a scheme to create MSR codes which are also optimal in terms of the number of I/O operations performed on each of the nodes participating in the reconstruction. This is an important consideration in disk–based systems.

One of the assumptions [Dimakis et al., 2007] makes is that both the amount of stored information (α) and transmitted (β and γ respectively) information can have non–negative, real values. In real systems erasure codes are only able to store and transmit an integer number of bits and, most likely, are bound by other system requirements to manage and store larger data portions. A probabilistic approach could be introduced to model the storage and transmission of real number of bits. For example, $\alpha = 2.3$ could be seen as each node storing 2 with 0.7 probability and 3 with 0.3 probability. However, it is unclear how such an approach would affect the aforementioned bounds. Furthermore, increasing the granularity with which data is viewed in a system is generally not feasible in practice due to a cubic increase in the complexity of encoding and decoding operations. Therefore, we have chosen to introduce a more strict condition that these values must instead be positive integers. Thus, the bounds that we introduce in Chapter 2 and 4 are at least as constraining as those in the original work.

Beyond this practical consideration, our experimental results in Chapter 4 go a step further by also restricting the number of nodes that are available to use during the reconstruction, but without explicitly designing the system, as done in [Dimakis et al., 2007], to compensate for it. This means that we do not consider the same guarantees of reliability on each recovery, but can still provide reliability over an arbitrarily large number of loss/regeneration operations. This additional constraint can be used to model storage nodes that are unavailable temporarily due to network connectivity or are unable to respond in due time due to uneven, dynamically changing loads in the system. This is also a novel constraint absent in [Dimakis et al., 2007].

1.2.3 Code locality and network awareness

Repair traffic is such an important issue that it limits the proportion of data in a data center that can be erasure coded [Rashmi et al., 2013a]. One of the ways to get around this is to use locally repairable codes that create redundancy local to the rack. Thus, for most failures "cheap" bandwidth inside the rack can be used to repair the lost data. Several commercial implementations use locally repairable codes, such as Windows Azure Storage [Huang et al., 2012a] and Facebook [Weiyan Wang, 2014, Sathiamoorthy et al., 2013a]. More recently, Hu et al. [Hu et al., 2016] introduced double regenerating codes, specifically tailored to minimizing inter–rack traffic in a multi–rack data center. They proposed a two–stage regeneration procedure that recombines data stored inside racks before using it in the repair. They show gains of up to 45.5% compared to a regenerating code at the MSR point on the assumption that intra–rack traffic is free. These solutions greatly reduce the loads on the most congested parts of the storage networks. As such, they are tailored to specific static scenarios.

Other more general solutions have been proposed that take network topology into account when doing repairs. However, most do this when distributing the data initially, thus limiting their ability to adapt to dynamic network conditions. In [Li et al., 2010] Li et al. proposed a tree–based topology–aware repair scheme based on Prim's algorithm that does not have this limitation. The paper described a heuristic–based approach to manage the large repair space and deal with the scenario when a second storage node is lost during repair. Unfortunately, the practical applicability of this approach is slightly diminished by the delay such a selection process introduces. Other research has focused on looking at specific network topologies. In [Akhlaghi et al., 2010] Akhlaghi et al. grouped nodes into two sets, a "cheap" and an "expensive" set, based on the cost of access. They introduced generalized regenerating codes and showed that by downloading more packets from "cheap" nodes, the weighted cost of repairing failed nodes can be reduced. In [Gastón et al., 2013] Gastón et al. presented a similar model for a 2 rack system employing regenerating codes. It considered the different cost of accessing inter–rack and intra–rack data as well as the location of the newcomer node to define a threshold function which minimizes the amount of stored data per node and the bandwidth needed to repair a failed node.

Our work differs from the state of the art in three notable ways. Firstly, we have proposed a general solution to make erasure codes network-aware that is not limited to one particular network topology or code. Secondly, our solution allows for changing network costs that reflect real-time changes in traffic, rather than static costs based on topology. Thirdly, repairs can be initiated almost instantly when a node fails, allowing for up-to-date traffic information to be used.

1.2.4 Network coding

Most erasure codes perform repair by replacing lost pieces with exact copies. This is termed as exact repair and has the advantage of deterministic operation. In contrast to this, in functional repair lost pieces are replaced with functionally equivalent ones instead of exact copies. The basic idea emerged from network coding [Ahlswede et al., 2000], a highly effective technique to disseminate information through a network that allows intermediary nodes to transmit functions of the data they received as opposed to simply forwarding it based on a routing table. Ho et al. proposed [Ho et al., 2006] using functions based on linear coefficients selected uniformly at random from the elements of a finite field and called the technique Random Linear Network Coding (RLNC). It was quickly applied to storage [Szymonacedaski et al., 2005, Deb et al., 2005]. The code's construction makes it well–suited for use in dynamic, heterogenous systems as it provides significant flexibility in the selection of pieces and nodes to repair or reconstruct data. Furthermore, it can be tailored to achieve many points on the trade–off curve, even dynamically after the initial data distribution. However, it suffers from significant computational overhead and crucially, due to the random selection of coefficients, requires a relatively large finite field to retain data integrity with high probability.

An information flow graph is typically used to determine what lower bounds must be met on the amount of data transferred during repairs to ensure that the maximum flow has the required value. Since the coefficients used for RLNC to create the repaired packets are drawn uniformly randomly from a finite field, there is a small, but non–zero probability that they introduce unwanted linear dependence not represented on the information flow graph. A significant amount of work has been done to study this probability [Khan and Chatzigeorgiou, 2016, Heide et al., 2011b]. Hu et al. proposed [Hu et al., 2012a] a heuristic two–phase checking mechanism on the coefficient matrices to deal with this problem. They noted that as long as a file is split into a small number of fragments, the number of matrices for which the rank should be checked stays manageable. We advocate a similar solution in Chapter 2. The novelty of our approach lies in decomposing the problem into smaller steps, instead of doing conventional Gaussian elimination. This allows for intermediary results to be reused. This reduces the number of computations to manageable levels, even if the number of fragments is large, expanding the number of scenarios where RLNC can be applied.

1.2.5 Cloud storage services

Beyond data centers, erasure codes can also benefit more dynamic storage systems. Cloud storage is widely adopted as a cost-effective solution for both enterprise and end user data.

A key limitation of single cloud solutions is that users are typically tied to a single provider and thus dependent on the provider's offered reliability. Providers outages are relatively rare and tend to last anywhere between a couple of minutes to several hours [MSPmentor, 2016]. However, their impact to business and the daily lives of private users can be significant. Another key issue is privacy. Even when the data is encrypted, it still lies on a single provider, which can make it particularly vulnerable to attacks or even disclosure to governmental bodies of a foreign country. Aggregating several cloud storage services into a single storage pool allows users to take advantage of the storage space offered by each provider. Many products are on the market that simplify this aggregation, [Otixo, 2017], [ZeroPC, 2017] and [odrive, 2017] to name a few. However, these still store individual files on individual providers and thus do not increase reliability or privacy.

A straightforward solution would be to replicate the data across several providers. We study this approach briefly in terms of storage efficiency and to highlight the difficulties in scheduling packets during data retrieval, also providing a comparison to our proposed solution. Recently, a PhD thesis [Joshi, 2016] studied this problem. It dealt with different queuing models based on whether requests can be canceled and also touched on how the problem differs if an MDS code is applied. A similar problem occurs when scheduling jobs with small tasks on multiple machines. Replicating or cloning tasks is also one of the approaches used to mitigate the effect of stragglers in this scenario. [Ananthanarayanan et al., 2013] focuses on small tasks and argues that the short time makes it hard to gather statistically significant amounts of measurements to apply conventional predictive mitigation techniques. Furthermore, the time taken to build a predictive model may in itself be a large part of the execution time of the task. This problem is similar for replicated cloud storage, since retrieving a file that is highly distributed consists of several short communications with the provider(s) and delays on any one of these may result in the delay of the whole process.

Distributing erasure coded data can ensure that all the data is accessible even if a given number of clouds is unavailable for a fraction of the cost of replicated storage. Cloud–RAID [Schnjakin and Meinel, 2013] and Saveme [Song et al., 2015] proposed using RAID–like techniques to this effect. These two solutions have the benefit of low encoding and decoding complexity, but do not provide the flexibility to accommodate for clouds with different performance levels or changing redundancy levels on the fly. Ladóczki et. al. [Ladoczki et al., 2015] showed that Network Function Virtualization can be used to steer data effectively during content storage and retrieval when multiple network coded storage nodes are involved.

Despite the added decoding overhead, erasure coding can decrease retrieval time. The authors of [Soljanin, 2010] compare an uncoded approach with network coding in several types of communication environments modeled as random walks [Pearson, 1905]. By reducing the uncoded approach to the coupon collector problem, it is shown that the final 10% of packets are responsible for most of the delay. The varying characteristics of cloud providers is an added challenge and some approaches may be better than others. The problems related to storing data on heterogenous storage devices has been studied previously with some scenarios of particular practical importance having seen more attention. In [Van et al., 2012], the authors look at the unbalanced scenario where one "super node" offers significantly more storage, reliability and availability then the others. They tailor two MDS and a non–MDS code specifically for this scenario. Their major contributions are the repair schemes that ensure a higher level of availability then those offered by schemas that do not take into consideration the non–homogenous nature of the system. Retrieval performance is not discussed.

Apart from the heterogeneous performance characteristics, an added challenge is dealing with variations. Clouds may suffer partial or total outages and some degree of variance is expected during normal operation. Furthermore, the users may move freely across the globe, changing the network between them and the service. The idea of moving data around to follow the users is not new. Follow–Me Cloud [Taleb and Ksentini, 2013b, Taleb and Ksentini, 2013a] proposes a framework to migrate services between data centers with the goal of keeping it close to the mobile user. It deals with the technical aspects of ensuring seamless transitions, whenever the mobile devices are issued with a new IP address. While the resulting system model is somewhat closer to that of a mobile storage cloud, it shares the same high–level goals we seek for aggregated cloud storage: ensure the best quality of experience for users by continuously adapting the underlying cloud services transparently in the background. Adapting the distribution of data to match the requirements of data consumers is a problem that is not unique to storage. Orca [Bal and Kaashoek, 1993] provides mechanisms through it's run–time to share objects between physically different machines. Objects with high read–write ratios are replicated. Replicas can also be destroyed when the ratio changes. The paper shows that the adaptation reduces the overall execution time of tasks that use shared objects.

We propose distributing network coded data across the cloud storage providers to improve reliability, security and storage efficiency compared to the single cloud scenario. We also present a comparison to simpler aggregated cloud solutions that employ replication. The novelty of our work lies in tailoring well-known ideas and mechanisms to this scenario, then solving any problems specific to it. We rely on the flexibility, the rateless nature and the MDS-like property of RLNC to improve on state-of-the-art solutions.

1.2.6 Fog computing

With the spread of smart mobile devices and the network infrastructure to support them, a shift towards networks with highly decentralized resources can be observed [Benet, 2014]. It is seen by some as one key characteristic of 5G [Hu et al., 2015]. Mobile devices have considerable unused storage space, computational capabilities and network bandwidth. Users have also become accustomed to backing up data to cloud storage providers, making it accessible from anywhere around the globe. We envision the mobile storage cloud [Fitzek and Katz, 2014] as a system that combines these two concepts and the key enabler of fog computing [Bonomi et al., 2012, Bonomi et al., 2014].

A mobile cloud stores data reliably on individually unreliable user devices over a peer–to–peer (P2P) network. It is *resilient* due to not having a single point of failure and *scalable*, ensured by the new resources each joining user contributes. Data is closer to where it is consumed and can "follow" users. It can be used on its own as a fully decentralized P2P system or as a support for more traditional system. This second scenario makes it possible for example to deploy mobile clouds in a hybrid storage system as an edge caching solution. In this special scenario the availability of data is ensured by a central storage system and the role of the mobile cloud changes somewhat to facilitating fast and reliable data delivery. As such, it can also be seen as an enabling technology for mobile cloud computing [Huang and others, 2011].

The mobile cloud faces a big challenge that is less marked in today's mostly static, centrally controlled storage solutions. It stores data on unreliable nodes, which join and leave the network frequently. To address this, specialized erasure codes must be employed to maintain data availability and mechanisms must be established to spread data to newly joined nodes. Unlike conventional erasure codes, they must be able to change their rate to adapt the level of storage redundancy to the number of nodes. A significant new challenge is the lack of a central controlling entity to coordinate the regeneration of data lost when nodes leave the system. Traditional erasure codes like Reed–Solomon, designed for environments that are more or less centralized, may behave poorly. In this regard, the inherently distributed nature of RLNC lends itself to improving data integrity in these scenarios as it does not require coordination between the storage nodes.

However, even if efficient data regeneration techniques exist, an open question still remains: can such a cloud sustain itself? More precisely, can online nodes fill new nodes with data? To answer it, we must examine how the dynamics of the joining and leaving processes relate to traffic and storage requirements. Thus, understanding user behavior and predicting node availability is crucial in defining the circumstances in which mobile storage clouds are feasible.

There are many studies of availability in computer networks, several in the area of distributed systems. [Mickens and Noble, 2006] examines different types of predictors and proposes a hybrid model for large distributed networks of PCs. It presents an evaluation of the application of the proposed methods on traces from existing systems. There have been some attempts to improve the accuracy of predictions by aggregating local ones. One of these can be found in [Forlines et al., 2012], here the main objective is to find a technique for a group of forecasters to outperform individual forecasts by experts in a certain field. Given a large enough number of forecasters and time to mark their performance, they achieved significant improvements over the expert–only approach. An interesting, although altogether different aggregation is described in [Van Horssen et al., 2002]. They propose employing logistic regression to predict the occurrence of plants in a given area of wetland based on field measurements, and then aggregate these results to give predictions for larger unmeasured areas.

The key difference between previous research and our work is that in our case mobile nodes are also involved in the peer–to–peer network as their capabilities allow them to participate in distributed data storage solutions. This makes the approaches mentioned in the previous paragraphs less efficient or in certain cases unfeasible, because the factors influencing the availability of mobile devices differ from those of stationary devices. This generally stems from three things: the behaviour of mobile users, the mobile nature of the devices (which can result in great variations in their network connectivity) and battery capacity. However, the aforementioned papers contain several concepts and ideas which have been adapted, or could be adapted to the technique we propose in this dissertation.

1.2.7 Working with mutable data

This is a key issue that has not been thoroughly studied for network codes (or other erasure codes) and where the state–of–the–art follows a costly approach. The difference between the original and the modified data can be stored using less space in an unencoded form, for example, using delta encoding. Several publications deal with applying differencing and compression techniques simultaneously [Korn and Vo, 2002, Suel and Memon, 2002], with [Hunt et al., 1996] providing an empirical study on several of them. Such techniques are not limited to storage applications and have been previously used for example to enable the caching of websites with dynamic content [Mogul et al., 1997, Naaman et al., 2004]. This class of algorithms belongs to a mature field of information theory with several applications in computer science. However, when applied on its own, it lacks the benefits offered by erasure codes and does not allow for a seamless update of the file, without first decoding the original file to later apply the history of changes.

The amount of research available on updating erasure–coded data has been very limited until recently. State–of—the–art solutions followed a costly approach [Esmaili et al., 2013], namely, to update encoded fragments entirely regardless of the nature or size of the change. Beyond a strain on the network, this process also requires a large use of space, especially if several versions of the file need to be maintained. The reason for this issue comes from the fact that each encoded fragment is created by the combination of all fragments from the original file. Thus, even small changes in various fragments can be compounded to large changes in the encoded fragments. Our proposed solution solves this for changes of bursty nature, limiting their effect on the rest of the file. It works with any linear block code. [Esmaili et al., 2013] deals with both repair and update of data and limits discussion to a particular 2 dimensional product code and size–preserving block updates. Recently, following the initial submission of our work, the field has seen interest from two, independent research groups. [Rouayheb et al., 2015] presented synchronization protocols to update concurrent modifications performed by different clients of the distributed storage system, focusing on the case where each encoded block has the same number of deletions. Data recoverability is maintained by changing the structure of the code during each update. [Wang et al., 2016] examined adding and deleting individual symbols and information–theoretically optimal compression. Both are solid theoretical contributions, but it is not clear how widely they can be applied in actual systems. Our work differentiates itself by also including practically–minded algorithms and measurements on an actual working system.

1.2.8 Software environment

Developing software for erasure coding can be deceptively simple. A library that supports basic linear operations over a finite field suffices. However, a key consideration is speed and efficiency. Fortunately, there are several fast open–source solutions available. Perhaps the best known library is ISA–L [Intel, 2017], developed by Intel specifically for use in storage. It takes advantage of the extended instruction sets of newer generation x86 processors. Kodo [Pedersen et al., 2011] is a general–purpose library with support for RLNC, Reed–Solomon and Fulcrum codes among others. It too is hardware optimized on x86 and also supports ARM–based systems. Jerasure [Plank and Greenan, 2014] is a storage–focused solution, while OpenFEC [Mathieu Cunche, 2017] is designed for LDPC.

1.3 Grouping of contributions into theses

I have condensed and formulated the results of our work as four separate theses. These follow the main areas of interest and the structure of the dissertation closely. The accompanying Thesis Booklet contains a more detailed presentation.

Thesis I: Erasure coding in data centers

I have proposed a framework to find the lowest cost feasible repairs in a network with heterogeneous, dynamic transfer costs. I have defined a set of bounds on the effectiveness of functional repair for scenarios where different nodes transmit different amounts of data to the failed node. Finally, I have introduced a mechanism to check the feasibility of repairs for codes that select coding coefficients at random. To reduce their computational cost, I have proposed decomposing them into smaller steps and reusing partial results.

This thesis is covered in Chapter 2 and by publications [1][4]. It also has two related patents [18][19].

Subthesis I.1: Network–aware repair

I have proposed a repair mechanism that minimizes transfer costs and, breaking with conventional systems, takes into consideration that costs may be heterogeneous and may change over time. Since the possible space of repairs can be large, I have examined where the feasible repairs are for three distinct erasure codes, only considering repairs that may be optimal given an arbitrarily selected cost function. I have shown using simulations that all three codes benefit significantly from being network-aware.

Subthesis I.2: Bounds on the effectiveness of functional repair

I have described the condition for maintaining data availability using functional repair for heterogenous β_i. I have shown that it is sufficient for all network codes and also necessary if $(N - L)\alpha = k$.

Subthesis I.3: Checking the feasibility of repairs

I have proposed a technique to check the rank of a large number of matrices ($\mathbf{M}$) by decomposing the checks into smaller steps, then reusing previously memoized results. I have shown that finding good decompositions is a difficult problem as individual steps are instances of set cover, an NP-hard problem. I have characterized the computational costs associated with the checks and proposed two decomposition schemes to find and two schemes to apply decompositions.

Thesis II: Distributed cloud storage using random linear network coding

I have created a model for storing data in a distributed fashion on multiple cloud providers using random linear network coding. I have provided theoretical results on the system's reliability, performance and security. I have examined performance using measurements on commercially available systems and proposed using a sparse code to adapt the data distribution to maximize performance. I have provided an algorithm to adapt the distribution of data in case a storage node becomes unavailable or a new one joins the system.

This thesis is covered in Chapter 3 and by publications [3][8][9][10].

Subthesis II.1: Finding the optimal data distribution

I have described and characterized a system that distributes data over cloud storage services that employs random linear network coding. I have characterized the reliability, performance and security of such systems and formulated the optimal data distribution as an integer programming problem.

Subthesis II.2: Characterizing retrieval performance

I have examined retrieval performance, comparing a network coded approach to replication based on several aspects. I have shown that although erasure codes require additional decoding time to retrieve a file, it is small compared to the benefit of not having to deal with stragglers.

Subthesis II.3: Adapting the data distribution to the current performance of the storage clouds

I have proposed four algorithms to dynamically adapt the data distribution to the current transfer rates offered by the clouds. I have formulated the benefits of using a sparse code to enhance data retrieval performance and have shown with measurements that it provides significant adaptation bandwidth saving whilst maintaining retrieval performance close to that of a dense code. Finally, I have described a simple algorithm to change the data distribution when a cloud joins or leaves the system.

Thesis III: Erasure coding for mobile storage clouds

I have examined the high-churn environment of mobile storage clouds and proposed reconstruction strategies to maintain reliable data storage. I have looked at how network coding can maintain cloud integrity in an uncoordinated system using both simulations with three different node failure patterns and an analysis of the information flow graph. I have defined the criteria for a self-sustaining mobile cloud and used a trace from a mobile P2P system to assess the practical feasibility of mobile storage clouds. I have proposed a two-phase scheme to predict node availability and performed statistical analysis on user behavior in a mobile P2P system.

This thesis is covered in Chapter 4 and by publications [2][5][6][11][12][13].

Subthesis III.1: Maintaining cloud integrity in high–churn environments

I have evaluated the effectiveness of different erasure codes for high churn decentralized distributed storage systems. I have proposed three repair strategies for RLNC and compared their effectiveness to traditional coding techniques subject to storage and network constraints. I have introduced three node failure patterns, one based on real-world traces, and shown that RLNC is more effective in maintaining cloud integrity even if Reed-Solomon and replicated solutions have the benefit of using a central controller.

Subthesis III.2: Bounds on the effectiveness of recoding

I have shown that the reason for RLNC's effectiveness lies in its recoding ability by examining information flow graphs and establishing constraints on the parameters of the system.

Subthesis III.3: Assessing the feasibility of a mobile storage cloud

I have established the requirements for a self-sustaining mobile cloud and analyzed traces from a popular mobile P2P application to validate that such a system is practically feasible. I have looked at the effects of the variations of node numbers on data availability if high-availability storage nodes are present in the system.

Subthesis III.4: Predicting node availability using a two–phase model

I have proposed a method of using device-local knowledge to predict the availability of a single node using a binary classifier and created a model for aggregating the individual predictions. The resulting formula can be used to calculate the required level of redundancy needed in a distributed storage system to be able to recover the stored data with a given degree of confidence.

Thesis IV: Updating erasure coded data

I have proposed a novel method to update fragments encoded with a linear block code. First, I proposed a general method and showed how it can handle changes, deletions and additions to the data. I have characterized its overhead compared to a non-encoded approach for bursty changes. Second, I provided detailed algorithmic descriptions for the individual steps and proposed a method to remap data in order to signal deletions. Finally, I showed using measurements on a Git repository, that the proposed approach requires several orders of magnitude less storage then a naive solution and is comparable to a non-encoded approach.

This thesis is covered in Chapter 5 and by publications [3][7]. It also has a related patent [20].

Subthesis IV.1: Representing and applying modifications in an encoded form

I have presented a general overview of how changes can be applied to a file that is stored in an erasure coded distributed storage system. I have examined three types of changes: modifying and deleting individual parts and adding new elements to existing data. I have described the expected storage overhead of representing changes in an encoded form compared to a non-encoded form.

Subthesis IV.2: Detailed algorithmic description

I have described the algorithms required for the three supported operations in great detail. The thorough presentation focuses on showing further aspects that must be taken into consideration to ensure that the proposed method can be applied with efficiency close to that of theoretical results. I have proposed a low-overhead method to remap symbols, necessary for signaling deletions. Finally, I have shown using measurements in a version control system that the size of the coded changes is comparable to the non-encoded approach.

1.4 Structure and source of the material presented in the dissertation

The dissertation presents an overview of the work done during my PhD, and is not intended to be a separate scientific publication. It contains material that I have previously written and published. I have improved the quality of the language and some of the figures, made the notation consistent across the document, tried to put the findings from different chapters into context by reflecting on the differences and similarities among them. Through these changes, I aimed to present the reader with a dissertation that can be read as stand-alone document while diverging as little as possible from peer-reviewed results. Chapter 5 is perhaps an exception to the latter intention, as I have increased its theoretical scope significantly and thus contains unpublished material.

The following chapters are organized as follows:

- Chapter 1 introduces the topic of the dissertation and presents related work. It contains edited material published in [1][3][4][5][8][10][13].
- Chapter 2 is a detailed description of Thesis I. It presents a network–aware framework for doing repairs and introduces bounds on the repair efficiency of functional repair. It also includes a description of a mechanism to check the feasibility of repairs for RLNC. It contains edited material published in [1][4].
- Chapter 3 is a detailed description of Thesis II. It proposes an aggregated cloud storage system and evaluates its characteristics. It presents measurement data and a method to adapt the data distribution based on the performance of the cloud services. It contains edited material published in [3][8][9][10].
- Chapter 4 is a detailed description of Thesis III. It examines the feasibility of mobile storage clouds, presenting different methods to maintain data integrity. It also includes an analysis on user behavior in a mobile P2P network and a two–phase method to predict node availability. It contains edited material published in [2][5][6][11][12][13].
- Chapter 5 is a detailed description of Thesis IV. It presents a method to seamlessly update erasure coded data and includes a detailed algorithmic description of the steps required to handle modification, deletions and insertions in files. It contains edited material published in [3][7].
- Chapter 6 summarizes the contributions of this work and presents some of the areas where it may be applied in the future.

Chapter 2

Cost-effective erasure coding for data centers

2.1 Introduction

A significant amount of research on using erasure coding for distributed data center storage has focused on reducing the amount of data that needs to be transferred to replace failed nodes. This continues to be an active topic as the introduction of faster storage devices looks to put an even greater strain on the network. However, with a few notable exceptions, most published work assumes a flat, static network topology between the nodes of the system. Neither assumption is true for most current data centers.

Network topology and changing traffic conditions play a crucial role in repair performance. To reflect these attributes, costs can be assigned to the transfer of packets between nodes. For example, costs could reflect transfer time or the proportion of available bandwidth used on the most congested link. Under these conditions, a solution that aims to

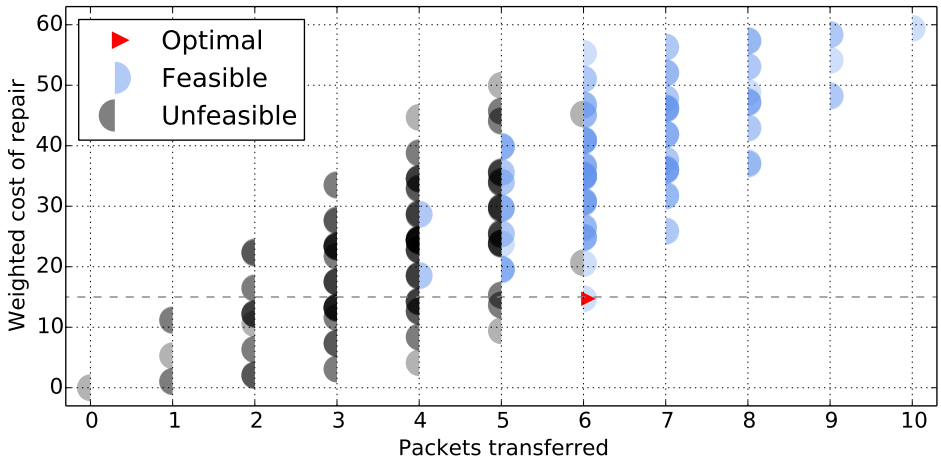


Figure 2.1: An example of the network-aware repair space of an erasure code

minimize the number of transferred packets would give a suboptimal solution from the perspective of a network–aware cost function. Figure 2.1 shows an example of such a case.

The horizontal axis shows the number of packets transferred by each repair. Traditionally, this metric has been considered as the de facto network cost of a repair and is sometimes denoted with γ [Dimakis et al., 2010] and referred to as repair bandwidth. The vertical axis shows the number of transferred packets weighted with a linear cost function that reflects the current state of the network. Blue semicircles denote feasible repairs, grey semicircles unfeasible ones. We define a repair as feasible if the resulting system maintains its ability to reconstruct the data from any subset of nodes of a predefined size. Thus, unfeasible repairs do not necessarily result in immediate data loss, but rather decrease the reliability of the system over the course of several rounds of failures and repairs. A red triangle denotes the repair with the lowest weighted cost. A naive approach that is not network–aware would select one of the feasible repairs on the $x = 4$ column, a suboptimal choice for this particular cost function. This invokes two questions that this chapter seeks to answer: how much do different types of codes benefit from being network–aware and where can the lowest cost feasible repairs be found in the repair space independent of the cost function used. An answer to the latter would provide a solution to reduce the size of the space that should be considered.

Guaranteeing data recoverability is a crucial aspect of data center design. Thus, codes employing randomly drawn coefficients, such as RLNC, must be used with caution to avoid data loss due to linearly dependent coefficients. We present a solution to this problem in the second part of this chapter.

2.1.1 Structure and overview of the contributions of this chapter

Our first contribution is the introduction of lower bounds on the repair traffic of erasure codes using functional repair. We look at scenarios where different storage nodes contribute different amounts of data to the repair effort. This serves as a base for some of our more practically–minded results on network–aware repairs.

Our second contribution is to make the repair of erasure–coded data network–aware by introducing a general framework that computes the feasibility of different possible repairs in advance. When a storage node fails, a repair is selected based on some cost function that reflects the current state of network connectivity among the storage nodes. By performing the potentially computationally–intensive feasibility checks in advance, the system is able to react to a node loss quickly and can base the repair selection on up–to–date network traffic data. This chapter investigates the gains for different types of erasure codes. The practical applicability of the proposed framework is also considered by presenting techniques to reduce the number of repairs to consider independent of the cost

function in use. This aspect is especially important for RLNC, where the set of feasible repairs of potentially lowest cost is of exponential size when using a naive approach.

Our third contribution is a technique to make checking the feasibility of a large number of repairs less computationally demanding. We present two methods to decompose the problem into smaller parts and formulate some of the general properties of decompositions. We also propose a technique to apply a decomposition as a schema for the actual checks as part of our proposed framework. We characterize the effectiveness of our solutions using both analytic and simulation–based tools.

This chapter is organized as follows. Section 2.2 establishes lower bounds on the repair efficiency of functional repair and introduces a large part of the model in use throughout this chapter. Section 2.3 includes an algorithmic definition of our proposed network–aware repair framework and provides methods on determining the set of feasible repairs that are potentially optimal for three distinct erasure codes. We show using experiments the degree to which each code benefits from being network aware. Section 2.4 looks at the cost of performing the feasibility checks and describes two algorithms to find good decompositions, providing experimental evidence to show their benefits. Section 2.5 summarizes our findings.

2.2 Theoretical bounds on the effectiveness of functional repair

2.2.1 System model

A file to be stored in the DSS is broken up into k pieces of identical size. Then, it is encoded using an erasure code to produce n coded pieces (packets). These are then distributed to the N nodes: $\Omega_N = \begin{pmatrix} node_1 & node_2 & \cdots & node_N \end{pmatrix}$, with each storing exactly α. When $node_f$ fails, all packets it stored are considered lost and must be repaired onto a replacement node. We designate the replacement node with the same name and consider repairs, where the surviving nodes can transfer different numbers β_i of packets to $node_f$: $\xi = \begin{pmatrix} \beta_1 & \beta_2 & \cdots & \beta_N \end{pmatrix}$. We call all possible repairs of a code where $node_f$ was lost its repair space: $\Xi_f = \{\xi \mid 0 \leq \xi[i] \leq \alpha \text{ and } \xi[f] = 0\}$ and use the term generation to denote a round of loss and repair. We require the system to maintain its properties over an arbitrarily large number of generations. We only consider single node losses as they are most common in systems with well–separated failure domains [Rashmi et al., 2013a]. Performing concurrent repairs allows for techniques that can further reduce network usage [Silberstein et al., 2014], but these are outside the scope of our work. However, we expect our proposed framework to be also useful in reducing network costs in the case

of multiple concurrent failures. The effectiveness of RLNC and some other erasure codes is better if the storage nodes are able to perform some basic operations, mainly additions and multiplications, on the data during repairs. The models and evaluation in this thesis study this case. We consider storage systems and codes with parameters that are $N, n, k, \alpha \in \mathbb{N}^+$, $\beta_i \in \mathbb{N}$.

We define a repair as feasible if the resulting system state maintains data recoverability after sustaining subsequent concurrent node losses. Each code, based on its parameters, therefore has a maximum number of L nodes it can lose concurrently while maintaining data recoverability. For codes employing exact repair like Reed–Solomon and RBT–MBR, the set of feasible repairs $\tilde{\Xi_f}$ as well as L is defined by the structure of the code. For regenerating codes employing functional repair, the set of feasible repairs is constrained by both the information flow graph [Ahlswede et al., 2000] and the coefficient selection method. On the information flow graph, a flow to a data collector with a value of at least k must be maintained with any L vertices from the final level of topological sorting removed from the graph*. In this sense, on an information flow graph with edges of capacity 1, i edge–disjoint paths must necessarily correspond to i linearly independent packets retrieved by the data collector.

2.2.2 Establishing bounds

The authors of the seminal paper [Dimakis et al., 2010] introduced lower bounds on repair traffic, given a level of storage efficiency and characterized the trade–off between these two metrics. They also showed that codes that achieve this bound exist. [Shah et al., 2010] extended this work by looking at heterogeneous repair scenarios, where surviving nodes are able to transfer different amounts of data to the repair node. The updated bounds included a cap on the amount of data transferred by any one node with the goal of reducing the total generated repair traffic.

Unfortunately, the introduction of the cap restricts the flexibility of the search for the most cost–effective repair strategy. Once transfer costs are weighted, some valid repairs may be disregarded. We have derived a significantly different formula for this heterogeneous case that does not place a cap on the transferred data. Based on this, we established the set of feasible repairs that have the potential to be optimal in Section 2.3.5.

To establish lower bounds on the effectiveness of codes employing functional repair in terms of limiting the amount of transferred data, we resort to analyzing the information

*For codes using random coefficients such as RLNC, further checks are necessary to ensure that the selection of coefficients does not introduce linear dependence not portrayed on the information flow graph. We deal with this aspect in Section 2.4.

flow graph (IFG). The results of this section can be applied for any code using functional repair and are not limited to RLNC. RLNC has been shown to be able to store and recover as many individual encoded pieces as the max–flow of the IFG as long as some constraints are met [Jaggi et al., 2006]. We use the system of checks described in Section 2.4 to select capacity–achieving coding coefficients and meet these constraints.

$\Omega_N^{(g)} \setminus node_f^{(g)}$ is the set of surviving storage nodes in generation g of loss and repair, and let $[S^{(g)}]^l$ denote the set of its l–subsets, i.e. $[S^{(g)}]^l := \{X \big| X \subset (\Omega_N^{(g)} \setminus node_f^{(g)}), |X| = l\}$.

Theorem 2.1. *Consider a DSS that uses a capacity–achieving erasure code that initially has the property of being able to recover the data from any* $(N-L)$ *nodes, where* $(N-L)\alpha = k$*. It maintains this property through an arbitrarily large number of single node failure and repair generations for any failure pattern if and only if the condition in Equation* (2.1) *is met.*

$$\forall g \geq 1, \forall s \in [S^{(g)}]^L : \sum_{node_i^{(g)} \in s} \beta_i^{(g)} \geq \alpha \tag{2.1}$$

Less formally, during a repair in generation g, any L sized selection of nodes must transfer at least α packets for the system to be able to sustain the loss of L nodes following the repair. This constraint is sufficient to ensure that the number of edge–disjoint paths on the IFG between the data source and a data collector does not decrease to below k if L nodes are subsequently lost in the following generation. It is also necessary for MSR codes, where $(N-L)\alpha = k$.

Corollary 2.2. *Theorem 2.1 also applies if* $(N-L)\alpha > k$ *with the change that the condition in Equation* (2.1) *is only sufficient, not necessary.*

To prove Theorem 2.1 and Corollary 2.2 we look at the set of information flow graphs defined in the following way and illustrated with an example on Figure 2.2. Let every state of a storage node in a generation be represented by an *in* and an *out* graph node, connected with an edge of capacity α to reflect that each node stores α packets. The *in* graph node of a surviving storage node is connected to the respective *out* node from the previous generation with an edge of value α. The *in* nodes of replacement storage nodes are connected to the $N-1$ *out* nodes of the previous generation of surviving nodes. Since such IFGs are directed acyclic graphs with edges only running between subsequent generations portraying surviving and repaired packets, each generation corresponds to a level in a topological sorting. The initial data distribution and the final data recovery are the first and respectively last levels of the IFG and we denote the data source and collector with s and t. The data source has edges of capacity α going to the the initial

N *in* nodes. Likewise, the data collector is connected to $N - L$ surviving *out* nodes from the final generation.

$$\text{data recoverability} \Rightarrow \text{mincut}(s,t) \geq k \tag{2.2}$$

The minimum cut that separates s and t must have a value of at least k to facilitate data recovery. If this condition is met, linear network codes exist that achieve data recoverability [Dimakis et al., 2010]. In the case of RLNC, the probability that a data collector can recover the data tends towards 1 as the field size increases [Ho et al., 2006]. We are ready to start the formal proof of Theorem 2.1.

Proof. Theorem 2.1 states that a flow of at least k on the IFG corresponding to an MSR code is maintained after an arbitrarily large number of single node failures and repairs followed by the concurrent loss of L nodes for any permutation of node failures if and only if in each generation every L sized subset of survivor nodes transmits at least α pieces to the replacement node. Formally:

$$\text{mincut(s,t)} \geq k \iff \forall g \geq 1,\, \forall s \in [S^{(g)}]^L : \sum_{node_i^{(g)} \in s} \beta_i^{(g)} \geq \alpha, \tag{2.3}$$

where $node_i^{(g)}$ is the i-th *out* node in generation g and $\beta_i^{(g)}$ is the capacity of the repair edge with which it is connected to the *in* node that failed in generation g.

We split the proof into two parts. First, we use an example to show that the condition formulated in the theorem is necessary to ensuring data recoverability. Second, we show that it is also sufficient by extending the analysis of min-cuts to a larger family of graphs. The second part also proves Corollary 2.2.

Let us look at the graph on Figure 2.2 that illustrates the loss of L nodes right after generation $g = 1$. The node that failed in the first generation is not among them. The grey bubble denotes a cut of value $\alpha(N - L - 1) + \sum_{i=N-L+1}^{N} \beta_i^{(1)}$ that separates s from t. By substituting the value of the cut and $k = \alpha(N - L)$ into the right side of (2.2), we get:

$$\begin{aligned} \alpha(N-L-1) + \sum_{i=N-L+1}^{N} \beta_i^{(1)} &\geq k \\ \alpha(N-L-1) + \sum_{i=N-L+1}^{N} \beta_i^{(1)} &\geq \alpha(N-L) \\ \sum_{i=N-L+1}^{N} \beta_i^{(1)} &\geq \alpha \end{aligned} \tag{2.4}$$

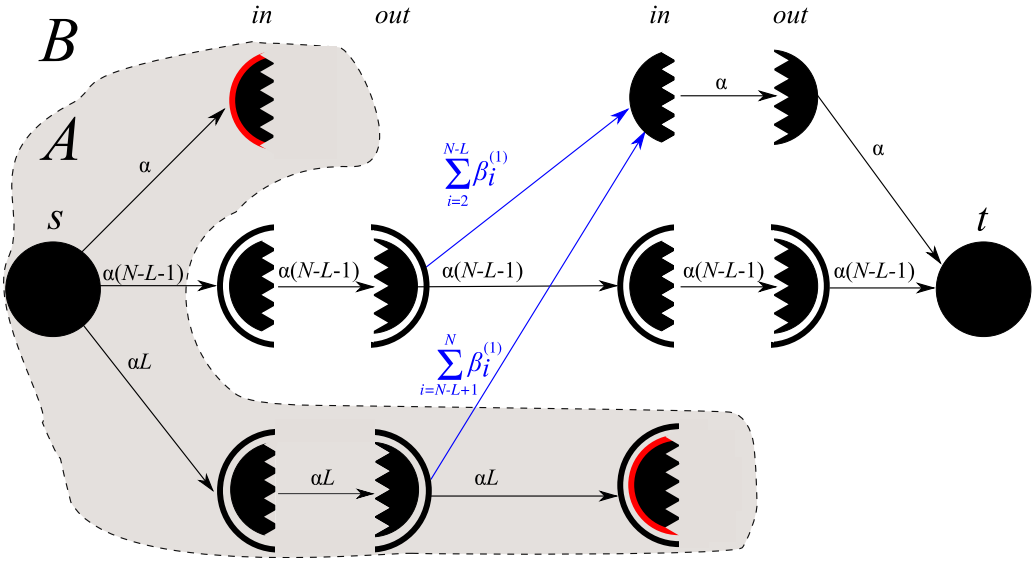


Figure 2.2: Information flow graph after a single generation of node failure and repair, followed by the loss of L storage nodes. Nodes with black circles around them represent groups of nodes that have edges from and to the same nodes or groups of nodes. The multiplicity of the group in the middle and bottom rows is $N-L-1$ and L respectively. Nodes and groups of nodes that have failed have a red circle around them. Black edges denote data stored on surviving nodes, blue edges denote repairs and the grey bubble denotes a cut. Capacities are shown above the edges.

If the L intersected repair edges have a capacity lower than α, the cut will have a value that is less than k. This is the only point in the proof where Theorem 2.1 and Corollary 2.2 diverge, as the same argument cannot be made if $k < \alpha(N-L)$. In fact, it is easy to find examples where this cut has a lower value for non-MSR codes while data recoverability is maintained. Thus, the condition formulated in the theorem is not necessary for all non-MSR codes.

In the second part we show that it is sufficient to ensure data recoverability after an arbitrarily large number of generations for both MSR and non-MSR codes (given a viable set of coding coefficients). We base this part on a slightly extended model of an IFG presented in [Dimakis et al., 2010], in particular on the proof of Lemma 2. Let us consider a larger family of graphs defined in the following way: as before, the initial N nodes are connected to s using edges of α capacity. However, unlike in the first part, a storage node is represented by a single pair of *in* and *out* graph nodes throughout its existence[†]. Replacement nodes connect to any $N-1$ of the existing nodes using edges of capacity $0 \leq \beta_i^{(g)} \leq \alpha$. The data collector t may connect to any combination of $N-L$ initial and replacement *out* nodes with edges of value α.

[†]This generalization to the model can be avoided, but makes the proof much simpler by revealing the most constraining cuts.

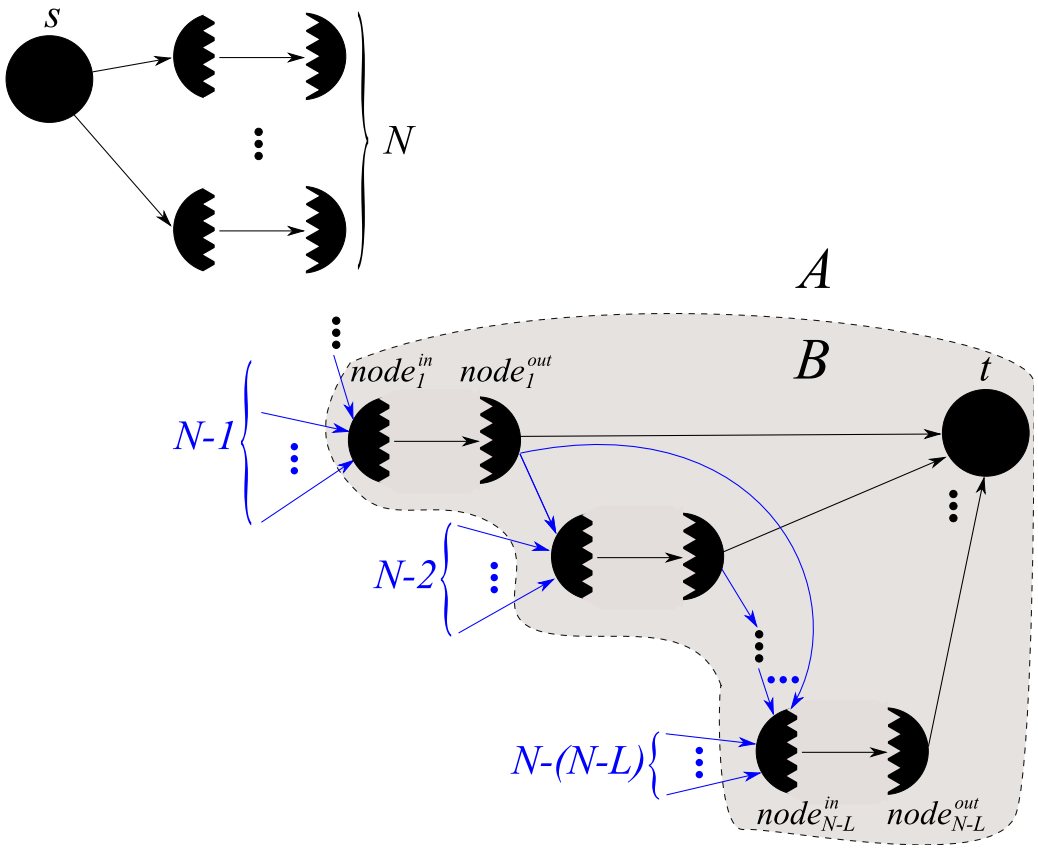


Figure 2.3: An example of an extended IFG and the most restrictive type of cut. Black edges have capacity α. Blue edges have differing capacities between 0 and α.

Figure 2.3 shows an illustration of such a graph. Let us look at all possible cuts that separate the nodes into sets A and B such that $s \in A$ and $t \in B$ in order to establish the value of the mincut of such graphs. We order the nodes t connects to based on their position in the topological sorting of the graph, where $node_i^{in}$ and $node_i^{out}$ represent the i-th *in* and *out* nodes in the sorting order. Let us examine how much different types of nodes contribute to the flow.

Each initial *out* node t connects to contributes α to the flow regardless of whether $node_i^{in}, node_i^{out}$ are elements of A or B, as s and t are connected by a chain of edges of capacity α.

Every replacement *out* node t connects to contributes potentially less, depending on where it is in the topological ordering of the IFG among the nodes t connects to. Cuts for which $node_i^{in} \in A$ place a restriction on the flow of at most α. Cuts for which $node_i^{in} \in B$ contribute with $\min(\alpha, \sum_{j=1}^{N-i} \beta_j^{(g)})$. For example, if $node_1^{in} \in B$, the cut crosses $N-1$ repair edges. If $node_2^{in} \in B$, the cut might cross one less, if there is a repair edge going between $node_1^{out}$ and $node_2^{in}$, as this does not intersect the cut. If the last replacement node, $node_{N-L}^{in} \in B$, the cut might cross anywhere between $N-1$ and $N-(N-L) = L$

repair edges, depending on how many replacement *in* nodes are elements of B. Figure 2.3 portrays the latter, most restrictive case in terms of max flow, as t only connects to replacement nodes and every replacement node in term connects to the most recent $N-1$ *out* nodes. The last replacement node $node_{N-L}$ has the smallest contribution as it enables a flow of $\min(\alpha, \sum_{j=1}^{L} \beta_j^{(g)})$. Thus, if $\sum_{j=1}^{L} \beta_j^{(g)} \geq \alpha$, as stated in the premise of the theorem, all nodes t connects to supply a flow of α for a total of $(N-L)\alpha$. We have shown that $\text{mincut}(s,t) \geq k$ for a more general family of IFGs. □

We have proved that as long as α packets are transferred from any L–sized set of surviving nodes during a repair, data integrity is maintained. The condition is also necessary for MSR codes. As a consequence, a significant reduction in overall network traffic (γ) can be achieved compared to more traditional erasure codes, such as Reed-Solomon. This result also serves as the base for network-aware functional repair.

2.3 Network–aware repair

2.3.1 Representing the cost of network transfers

We define the cost functions using matrix $\mathbf{C}$, where $c_{i,j}$ denotes the cost to transfer a single packet from $node_i$ to $node_j$ and $\mathbf{C}[j]$ is column j that contains the costs associated with transfers to $node_j$. We introduce two restrictions on $\mathbf{C}$. First, the diagonal elements must be $c_{i,i} = 0$. Second, all other elements $i \neq j$, $c_{i,j} \geq 0$.

$$\mathbf{C} = \begin{pmatrix} 0 & c_{1,2} & \cdots & c_{1,N} \\ c_{2,1} & 0 & \cdots & c_{2,N} \\ \vdots & \vdots & \ddots & \vdots \\ c_{N,1} & c_{N,2} & \cdots & 0 \end{pmatrix} \tag{2.5}$$

We use this general way of modeling costs to make it applicable to different network topologies and traffic patterns. It can be based on any number of measured parameters such as available bandwidth, latencies, number of dropped packets, queuing delays. It can be used, but is not limited, to minimize the total time required for repairing lost data. We make the assumption that the cost of transferring a single packet from $node_i$ to $node_j$ is not dependent on the total number of packets sent between them in the period in which the cost is regarded as accurate. This assumption is valid if the examined period is short or the repair traffic is a negligible fraction of the traffic flowing on the same links.

We evaluate the network–aware cost–weighted repair space of the code as shown on Figure 2.1, where the weighted cost for repairing data on $node_f$ using repair ξ_i is $cost(\xi_i) = \xi_i \mathbf{C}[f]$, where $\mathbf{C}[f]$ is used to denote column f.

2.3.2 A network–aware repair framework

Our proposed framework, defined as Algorithm 2.1, selects the lowest cost repair and is independent of erasure code and network topology. Whenever there is a change in the layout of the data (the initial distribution and any subsequent repairs), the set of feasible repairs $\tilde{\Xi}_f$ is computed for each possible subsequent node failure. The implementation of the $is\_feasible()$ function from Algorithm 2.2 is determined by the erasure code in question and the definition of feasibility. When a node fails, the cost for each feasible repair is calculated based on a cost function reflecting up–to–date network conditions.

Algorithm 2.1 Network–aware repair framework

```
1: ↯ Initial data distribution ↯
2: precompute_feasibility
3: min_cost := ∞
4: repeat
5:     ↯ node_f fails ↯
6:     for ξ_j ∈ Ξ̃_f do
7:         if cost(ξ_i) = ξ_i C[f] < min_cost then
8:             min_cost := cost(ξ_i)
9:             ξ_sel := ξ_i
10:        end if
11:    end for
12:    execute ξ_sel
13:    precompute_feasibility
14: until false
```

The practical applicability of our proposed framework is determined by the complexity of the $is\_feasible()$ function, the size of Ξ_f and $\tilde{\Xi}_f$. The former depends on how feasibility is defined. In our interpretation from Section 2.1, the set of feasible repairs can be easily determined for codes employing exact repair. For codes that use functional repair and randomly selected coefficients, computational complexity is determined by the values

Algorithm 2.2 Precompute feasibility

```
1: procedure precompute_feasibility
2:     Ξ̃_i := ∅
3:     for node_i ∈ Ω_N do
4:         for ξ_j ∈ Ξ_i do
5:             if is_feasible(ξ_j) then
6:                 Ξ̃_i := Ξ̃_i ∪ ξ_j
7:             end if
8:         end for
9:     end for
10: end procedure
```

of parameters N, n, k, α as the rank of a potentially large number of matrices must be checked.

In the following subsections we define specific cost functions for three erasure codes in order to reduce the number of repairs to consider and to be able to characterize the repair space of each code in terms of where the lowest cost feasible repairs are. The codes were chosen to cover both exact and functional repair and both MSR and MBR points on the storage–repair bandwidth trade–off curve. Let us look at finding the minimum cost feasible repair ξ_{min} and its associated cost: $\kappa = cost(\xi_{min}) = \sum_{i=1,i\neq f}^{N} \beta_i c_{i,f}$ after losing the data stored on $node_f$.

2.3.3 Locating potentially lowest cost repairs for Reed–Solomon

Here we examine decoding–based repair for Reed–Solomon (RS) as this can be applied to any linear MDS code.

Definition 2.3 (Decoding–based repair)**.** Decoding–based repair is a form of exact repair, it involves the retrieval of at least k packets and performing the decoding operation to recover the original data. Following this, the same coefficients as used in the encoding operation are used to recreate the lost packets.

We restrict our evaluation to the $\alpha = 1$ case to be in line with how RS is generally used in data centers. Let $c^{(1)}, c^{(2)}, \cdots, c^{(N-1)} : c^{(i)} \in \text{set}(\mathbf{C}[f]) \setminus c_{f,f}$ be a permutation of costs in ascending order and $\beta^{(1)}, \beta^{(2)}, \cdots \beta^{(N-1)}$ the corresponding number of transferred packets. Thus, the cost of the minimal cost repair is shown on Equation (2.6) and the number of feasible repairs to consider given no knowledge of $\mathbf{C}$ is $\left|\tilde{\Xi}_f\right| = \binom{n}{k}$.

$$\kappa_{\text{RS}} = \sum_{i=1}^{k} c^{(i)} \tag{2.6}$$

2.3.4 Locating potentially lowest cost repairs for RBT–MBR

There are two distinct repair strategies to consider in the case of RBT–MBR. Ideally, each surviving node will transfer a single encoded packet ($\beta_i = 1, i \neq f$) as defined in the code construction. The authors of [Shah et al., 2012] coined it repair–by–transfer.

Definition 2.4 (Repair by transfer)**.** A form of exact repair that involves replacing lost packets with identical copies stored on other nodes. No computations are performed on either the parent or the repaired node.

Alternatively, if at least k distinct packets are transferred, the decoding of the embedded MDS code can take place and any missing code words can be re–encoded. Whilst this

second decoding–based repair strategy involves additional bandwidth and computation, it can result in lower transfer costs for some $\mathbf{C}$. Let $c^{(i)}$ and $\beta^{(i)}$ be defined the same way as in the previous subsection. The cost of the optimal repair $\kappa_{\text{RBT-MBR}}$ is specified in Equation (2.7) based on the two repair strategies.

$$\kappa_{\text{RBT-MBR}} = \min\left(\sum_{i=1}^{N-1} c^{(i)}\,,\ \sum_{i=1}^{N-L}(\alpha - i + 1)c^{(i)}\right) \tag{2.7}$$

The first term is the cost of transferring a single packet from each surviving node. The second term expresses retrieving as many packets from the lower cost nodes as possible without getting duplicates. $\sum_{i=1}^{N-L}(\alpha - i + 1) = k$ because the embedded code is MDS and the way RBT–MBR is constructed [Shah et al., 2012]. With no knowledge of $\mathbf{C}$, the number of repairs that are potentially lowest cost is reduced to $\left|\tilde{\Xi_f}\right| = 1 + (N-L)!\binom{N-1}{N-L}$.

2.3.5 Locating potentially lowest cost repairs for RLNC

Let us use Theorem 2.1 to derive the formula for $\kappa_{\text{RLNC-MSR}}$. Let $\beta^{(1)}, \beta^{(2)}, \cdots, \beta^{(N-1)}$ be a permutation of the number of packets transferred from remaining nodes of ascending order and $c^{(1)}, c^{(2)}, \cdots, c^{(N-1)}$ the respective costs from set$(\mathbf{C}[f]) \setminus c_{f,f}$. Taking Equation (2.1) into consideration, we can define a more specific cost function for the optimal repair in Equation (2.8), by only considering repairs $\sum_{i=1}^{N-1}\beta_i \leq k$.

$$\begin{aligned}\kappa_{\text{RLNC-MSR}} &= \sum_{i=1}^{L} c^{(i)}\beta^{(i)} + \sum_{i=L+1}^{N-1} c^{(i)}\beta^{(L)} = \\ &= \sum_{i=1}^{L-1} c^{(i)}\beta^{(i)} + \beta^{(L)}\sum_{i=L}^{N-1} c^{(i)}\end{aligned} \tag{2.8}$$

The first term expresses the cost for the L lowest values of $\beta^{(i)}$, the second term the cost for the rest of the nodes. Each of these must transfer at least $\beta^{(L)}$ to satisfy Equation (2.1). $\kappa_{\text{RLNC-MSR}}$ is minimized if the $c^{(i)}$ are in descending order, i.e. transferring more from cheaper nodes and less from expensive ones. The free variables are thus reduced to $\beta^{(1)}, \beta^{(2)}, \cdots, \beta^{(L)}$. Given that Equation (2.1) should be satisfied with equality for ξ_{min}, this leads to a significant reduction in the number of potential repairs to consider, as shown in Equation (2.9). Furthermore, it determines the positions of the lowest cost feasible repairs in Ξ_f and once $\mathbf{C}$ is known, the optimal repair can quickly be selected.

$$\left|\tilde{\Xi_f}\right| = \left|\left\{\xi : \sum_{i=1}^{L}\beta^{(i)} = \alpha\right\}\right| \tag{2.9}$$

This is an integer partitioning problem on α that is constrained by limiting solutions to those with L additive parts. The number of non–constrained partitions is given by a recurrence formula based on Euler's pentagonal number theorem [Bell, 2005]. The first elements can also be found in the OEIS[‡] as sequence A000041 [Sloane, 1991]. The number of solutions with L parts is equal to the number of partitions in which the largest part is of size L [Alder, 1969]. A similar recurrence formula exists for this constrained version of the problem [Stanley, 2011].

As the bound in Equation (2.1) is sufficient to ensure data survival for all parameters, we may apply the previous results for non–MSR codes as well. However, it is possible to define tighter bounds for these codes [Shah et al., 2010]:

$$\sum_{i=1}^{N-1} \beta^{(i)} \geq \gamma(\beta_{\max}), \tag{2.10}$$

where $\gamma(\beta_{\max}) = \max(\alpha - \beta_{\max}, k \bmod \alpha) + \lfloor k/\alpha \rfloor$.

The authors of [Shah et al., 2010] introduce a cap on the amount of packets any single node transfers, $1 \leq \beta_{\max} \leq \alpha$. They argue against full flexibility ($\beta_{\max} = \alpha$), as it involves transferring at least k packets. However, for some $\mathbf{C}$, transferring k packets is actually the lowest cost repair strategy. An example can be seen on Figure 2.4d. Based on Equation (2.10), we define the costs of optimal repairs for a given $\beta_{\max}$ as

$$\begin{aligned}\kappa_{\mathrm{RLNC}}(\beta_{\max}) = \sum_{i=1}^{\lfloor \gamma(\beta_{\max})/\beta_{\max} \rfloor} & c^{(i)}\beta_{\max} + \\ & c^{(\lfloor \gamma(\beta_{\max})/\beta_{\max} \rfloor + 1)}(\gamma(\beta_{\max}) \bmod \beta_{\max}).\end{aligned} \tag{2.11}$$

A simple way to enumerate as many feasible repairs as possible using these bounds is to look at all values of $\beta_{\max}$. The number of feasible repairs of potentially optimal cost is given by

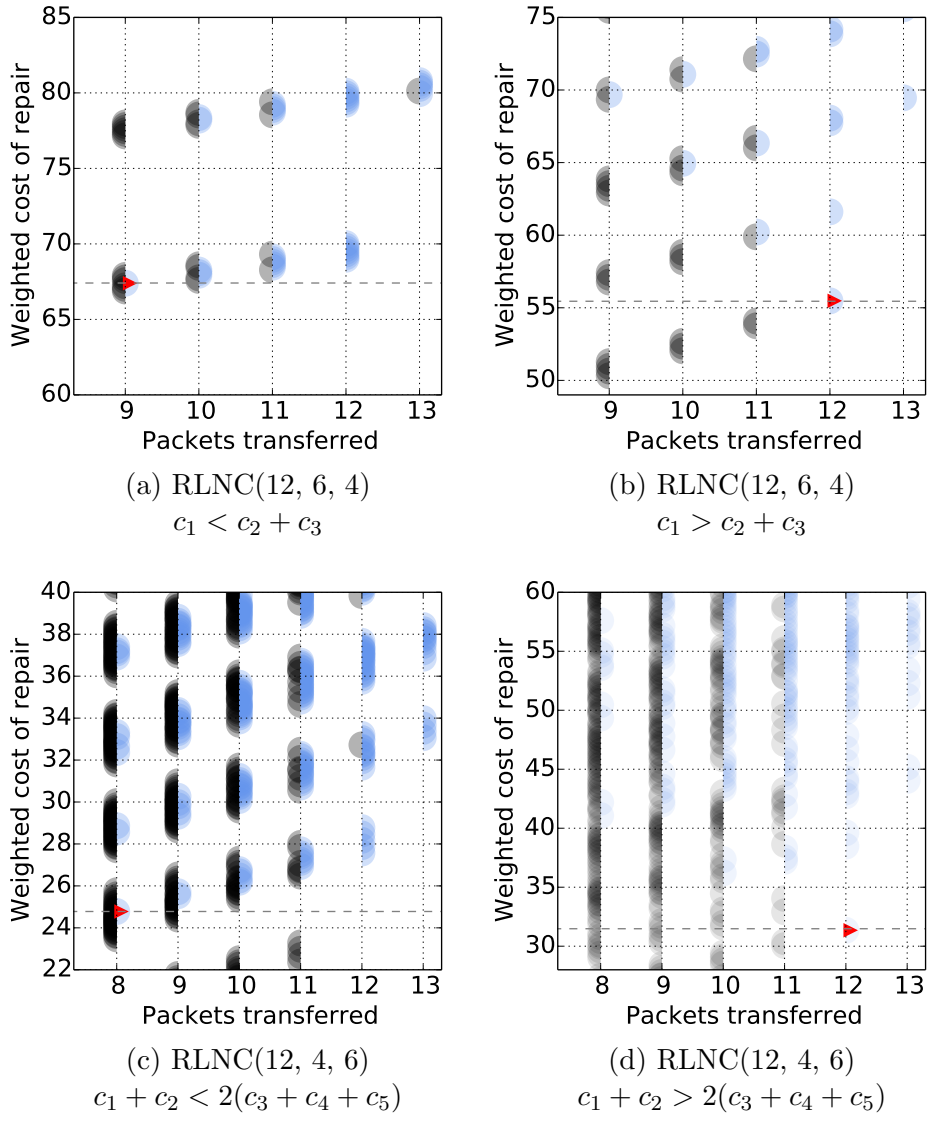
$$\left|\widetilde{\Xi}_f\right| \geq \sum_{\beta_{\max}=1}^{\alpha} \left| \left\{ \xi : \sum_{i=1}^{N-1} \beta^{(i)} = \gamma(\beta_{\max}) \right\} \right|, \tag{2.12}$$

providing a similar set of constrained integer partitioning problems as for the MSR point. In this case the constraints is that the largest part of a partition must be at most $\beta_{\max}$. Due to the equivalence described in [Alder, 1969], it is the same problem as for the MSR point, for multiple $\beta_{\max}$.

[‡]The On–Line Encyclopedia of Integer Sequences: `https://oeis.org/`

2.3.5.1 Case study for RLNC

Let us look at two sets of parameters at the MSR point for which RLNC behaves differently depending on $\mathbf{C}$. We have selected these particular sets because of their low number of potentially minimum cost repairs. This keeps the illustration of the previously presented theoretical results brief. We assume with no loss in generality that the last node, $node_N$ failed and $c_i = c_{i,N}$ are in descending order. Figure 2.4 shows zoomed–in views of the part of Ξ_f containing $\xi_{\min}$.



(a) RLNC(12, 6, 4)
$c_1 < c_2 + c_3$

(b) RLNC(12, 6, 4)
$c_1 > c_2 + c_3$

(c) RLNC(12, 4, 6)
$c_1 + c_2 < 2(c_3 + c_4 + c_5)$

(d) RLNC(12, 4, 6)
$c_1 + c_2 > 2(c_3 + c_4 + c_5)$

Figure 2.4: Case study for RLNC(k, α, N)

First, we look at $k = 12$, $\alpha = 6$, $N = 4$ and require that $L = 2$ failures be supported. Considering Equation (2.8) and assuming repairs do not introduce linear dependence, only 4 of them need to be compared to find $\xi_{\min}$.

$$\xi_1 = \begin{pmatrix} 3 & 3 & 3 & 0 \end{pmatrix},\ \xi_2 = \begin{pmatrix} 2 & 4 & 4 & 0 \end{pmatrix},$$
$$\xi_3 = \begin{pmatrix} 1 & 5 & 5 & 0 \end{pmatrix},\ \xi_4 = \begin{pmatrix} 0 & 6 & 6 & 0 \end{pmatrix}$$

For $c_1 = c_2 + c_3$ all four repairs have the same cost. For $c_1 < c_2 + c_3$, ξ_1, the most balanced repair with the least amount of packets transferred, has the lowest cost as shown on Figure 2.4a. On the other hand, for $c_1 > c_2 + c_3$, $cost(\xi_1) > cost(\xi_2) > cost(\xi_3) > cost(\xi_4)$, i.e. the repair transferring the most amount of packets has the lowest cost as shown on Figure 2.4b. Thus, in these cases a traditional mechanism that only tries to minimize the amount of transferred data will sub–optimally pick ξ_1, giving an error of $cost(\xi_1) - cost(\xi_4) = c_1 - c_2 - c_3$. More importantly, ξ_2 and ξ_3 will not be the lowest cost repairs regardless of $\mathbf{C}$, thus the number of relevant repairs whose feasibility must be checked is further reduced to just those transferring 9 and 12 packets, ξ_1 and ξ_4 in this case.

Second, we look at $k = 12$, $\alpha = 4$, $N = 6$ and require that $L = 3$ node failures be supported. In this case the lowest cost feasible repairs are

$$\xi_1 = \begin{pmatrix} 1 & 1 & 2 & 2 & 2 & 0 \end{pmatrix}, \xi_2 = \begin{pmatrix} 0 & 2 & 2 & 2 & 2 & 0 \end{pmatrix},$$
$$\xi_3 = \begin{pmatrix} 0 & 1 & 3 & 3 & 3 & 0 \end{pmatrix},\ \xi_4 = \begin{pmatrix} 0 & 0 & 4 & 4 & 4 & 0 \end{pmatrix}.$$

The cut–off point between ξ_1 and ξ_4 is $c_1 + c_2 = 2(c_3 + c_4 + c_5)$. Due to the limited number of ways the number 4 can be reduced to additive components, there are no minimal cost feasible repairs with a total of 9 or 11 transferred linear combinations. Thus, there might not be a clear decreasing or increasing order of costs like in the previous example as shown on Figure 2.4c and 2.4d. Therefore, more repairs must be checked.

2.3.6 Evaluation method for network awareness

In this section we evaluate how much each erasure code benefits from being network aware and perform a thorough analysis on the number of computations required to perform the feasibility checks. We compare our proposed framework that guarantees finding the least cost repairs to a naive approach that selects one of the repairs with the lowest traffic but has no knowledge of transfer costs. We perform our analysis using sets of code parameters (N, α, k) that meet the following constraints: $2 \leq N \leq 20$, $1 \leq \alpha \leq 10$, $5 \leq k \leq 32$. We also required that each code must be able to sustain $L \geq 2$ node losses while maintaining data recoverability following each repair. We have selected such a wide range of values to cover as many of the parameter sets that are interesting as possible. We have strived to take into consideration practical aspects as well. For example, we have chosen to have an upper bound on k as it is the main factor that determines the decoding overhead during read operations for non–systematic codes such as RLNC and RBT–MBR. We also require

that codes have a storage efficiency of $N\alpha/k \leq 2.5$. We decided to use 2.5 as a cut–off point to include more cases for RBT–MBR, despite being a relatively high value for a practical system. For Reed–Solomon codes, we only consider $\alpha = 1$ as this maximizes its ability to lose nodes. Recently, Guruswami et al. [Guruswami and Wootters, 2015] proposed subpacketization for RS raising the possibility that repair may benefit from other values of α (or non–integer values of β_i). However, we are unaware of any practical implementation or the exact implications of this construction and have therefore decided to consider RS as used in current systems. For RLNC and RBT–MBR, we restrict our evaluation to sets which have a repair space size for a given failed node of at most 2^{16} and 2^{24} respectively. While one of the key benefits of our proposed approach is that it can handle cases where the repair space is very large by only considering a small fraction of repairs, we would have been unable to compare our solution to the baseline approach without this constraint. Nevertheless, most practically interesting cases for N, k and α are included. 50 sets of parameters meet these constraints for RS, 8 for RBT–MBR and 214 for RLNC.

Each run for each code, costs and set of code parameters consisted of 100 iterations of node loss and recovery. Operations were performed over GF(2^8). Two types of cost matrices $\mathbf{C}$ were considered. First, **I**: one that is based on a static network topology where nodes are grouped evenly in 2 racks. Costs have two types: inter–rack(10x) and intra–rack (1x). We used this model to evaluate the benefits of network awareness assuming a simple, static topology such as that defined in [Gastón et al., 2013]. Second, we used a cost matrix that also portrays current network traffic conditions. The same $\mathbf{C}$ is multiplied entrywise in each round with a different matrix containing values drawn randomly from the following uniform distributions: **II**: $U(0.75, 1.25)$, **III**: $U(0.5, 1.5)$, **IV**: $U(0.25, 1.75)$, **V**: $U(0, 2)$. This allows us to portray the relationship between the variance in network activity and the potential benefit for an erasure code to be network–aware.

2.3.7 Experiments

Figure 2.5 shows how much each erasure code benefits from knowledge of network costs. Because of the different parameter sets, the codes should not be compared against each other directly. However, a general trend can be observed: the larger the variance in the costs, the larger the gain is compared to the naive approach. Thus, a distributed storage system with more dynamic traffic patterns will potentially see a larger benefit from performing network–aware repairs. For Reed–Solomon and RBT–MBR which use exact repair, most cost types show gains across all parameter sets. In the case of RLNC, there is a significant gain overall, but there are sets of parameters which show no gain. RBT–MBR does not benefit at all from being network–aware in a balanced static 2 rack

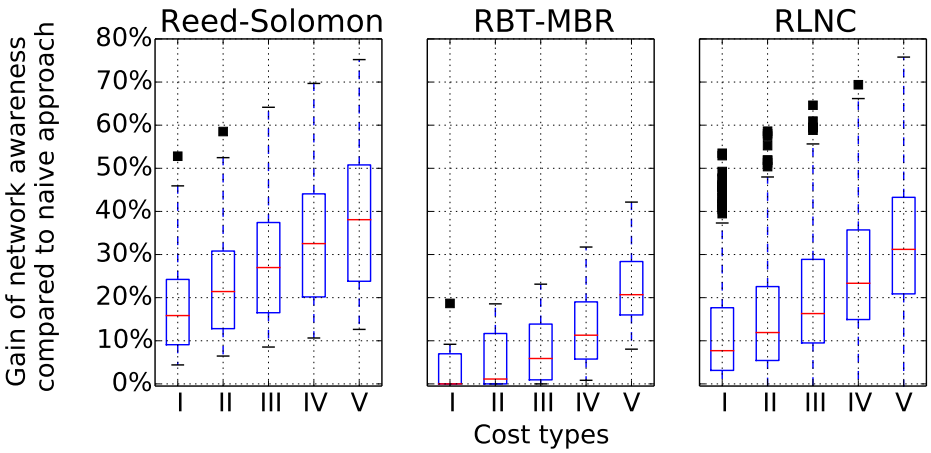


Figure 2.5: Gains of network awareness

scenario regardless of the actual values of the cost. This can be explained by the fact that decoding–based repairs must always access a certain amount of data from outside the rack in a balanced scenario. If the difference between the costs of going to the different inter–rack nodes and intra–rack nodes respectively is the same, a decoding–based repair will entail a cost that is greater than that of performing the baseline approach of repairing by simply transferring data. However, if there is variance between the cost of accessing nodes in the same cost category, we see that repair by transfer is not always optimal and our proposed approach leads to gains.

2.4 An efficient method to check the feasibility of repairs

RLNC encodes and repairs data using randomly selected coefficients. Thus, it is possible that repairs that otherwise conform to the requirements of Equation (2.1) are not feasible due to an unfortunate selection of coefficients. This happens if some of the resulting coefficient matrices become singular after the proposed repair. This lowers the fault tolerance of the system. To avoid this, we propose checking that all coefficient matrices that may be used during data retrieval have full rank before the actual repair takes place. To avoid delaying the repair, the system must perform the checks in advance for all possible single-node loss scenarios. Furthermore, all repairs that have the potential to be of minimum cost must be checked, as described in Section 2.3.2. If checks fail for a repair, a new set of coefficients should be generated and the checks repeated.

We denote the set of matrices that need to be checked with $\mathbf{M}$ and wish to define a mechanism that performs these checks in a computationally–efficient manner. The matrices contain the coefficients associated with data stored on $N - L - 1$ surviving and

one repaired node, α rows from each. Matrices that do not contain repaired rows need not be checked as they have been checked before the repair. Similarly, if the same node fails in successive generations, the checks can be skipped, since no new coefficients are introduced into the system.

Table 2.1: Table of Notation

N	$\triangleq$	the number of nodes
L	$\triangleq$	the number of unavailable nodes the system must support while maintaining data availability
α	$\triangleq$	the number of encoded symbols stored on a node, corresponding to rows in a coefficient matrix
r	$\triangleq$	the number of repairs to check for a possible node failure
$\mathbf{M}$	$\triangleq$	the set of matrices that need to be checked
$\mathbf{D}$	$\triangleq$	the set of matrices that form a decomposition
Ψ	$\triangleq$	a mapping that defines how each matrix in $\mathbf{D}$ is broken down into two smaller matrices
S_i	$\triangleq$	the set of matrices of size $i\alpha \times k$ in upper triangular form (UTF)
S_i^{sel}	$\triangleq$	the set of matrices of size $i\alpha \times k$ selected in decomposition D
s_i	$\triangleq$	an arbitrarily selected matrix from S_i or S_i^{sel}
$s_i^{(k)}$	$\triangleq$	the k–th matrix from S_i or S_i^{sel} given some arbitrarily defined order
$lz(i)$	$\triangleq$	the number of leading zeros in $s_i \in S_i$
$d(a,b)$	$\triangleq$	the number of divisions needed to get $s_i = s_a \oplus s_b$ into UTF if both s_a and s_b are in UTF
$m(a,b)$	$\triangleq$	the number of multiplications (same as the number of additions) needed to get $s_i = s_a \oplus s_b$ into UTF if both s_a and s_b are in UTF
$d_{\text{Gauss}}(k)$	$\triangleq$	the number of divisions needed to get s_k into UTF using Gaussian elimination
$m_{\text{Gauss}}(k)$	$\triangleq$	the number of multiplications/ additions needed to get s_k into UTF using Gaussian elimination

We use an index set to denote which nodes a matrix contains rows from: $s_i = \{index_1, index_2, \cdots, index_i\}$ and $|s_i| = i$. For example $s_3 = \{3, 4, 5\}$ contains rows from $node_3$, $node_4$ and $node_5$. While this representation is easier to follow, an alternative representation using bitmaps should be considered when implementing the algorithms that find good decompositions. Every coefficient matrix of size $i\alpha \times k$ can be represented with a bitmap (row vector with 0 and 1 elements) s_i' of size N based on which nodes its rows originate from. $s_i'[j] = 1$ if it contains rows of coefficients from node j and $s_i'[j] = 0$

otherwise. For example, $s'_3 = (0\,0\,0\,1\,1\,1)$ contains data from $node_3$, $node_4$ and $node_5$. Thus, bitwise operations with low CPU cycle cost can be used instead of set operations. Furthermore, bitmaps fit into a single 32bit or 64bit variable as typically $N << 64$.

2.4.1 Decomposing matrix rank checks into reusable parts

To perform the rank checks efficiently, we propose decomposing the Gaussian elimination performed for the coefficient matrices in $\mathbf{M}$ into smaller steps that can be shared between different checks and across subsequent failure and recovery generations. Once a set of steps and order is identified, it can be used as a schema as long as N, L and α do not change.

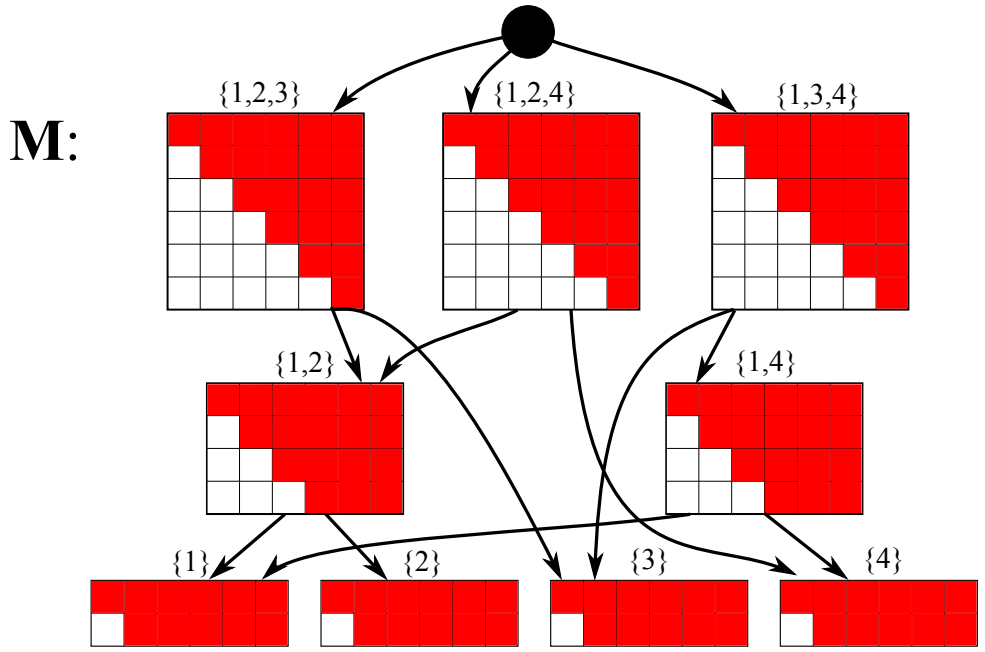


Figure 2.6: An example of a valid decomposition for $\alpha = 2$, $k = 6$, $N = 4$.

Definition 2.5 (Decomposition)**.** A *decomposition*$(\mathbf{D}, \Psi)$ is a set of matrices $\mathbf{D} = \{s_1^{(1)}, s_1^{(2)}, \ldots, s_1^{(k_1)}, \ldots, s_j^{(k_j)}\}$, where s_i is composed of rows from i nodes, $j \leq N - L$, $k_j \leq \binom{N}{j}$, $\mathbf{M} \subset \mathbf{D}$ and a mapping Ψ to match each matrix in $\mathbf{D}$ except those containing rows from a single node to a pair of matrices from $\mathbf{D}$.

A decomposition can also be thought of as a directed acyclic graph that defines the dependencies between matrices as shown on Figure 2.6. The matrices in $\mathbf{D}$ are the vertices and a root element is added that connects to all nodes representing matrices in $\mathbf{M} \subset \mathbf{D}$. Edges go between nodes according to Ψ to denote which pair of matrices should be transformed to upper triangular form before tackling a given matrix. The checks are then a traversal of the graph, and each visited vertex (matrix) is memoized in upper triangular

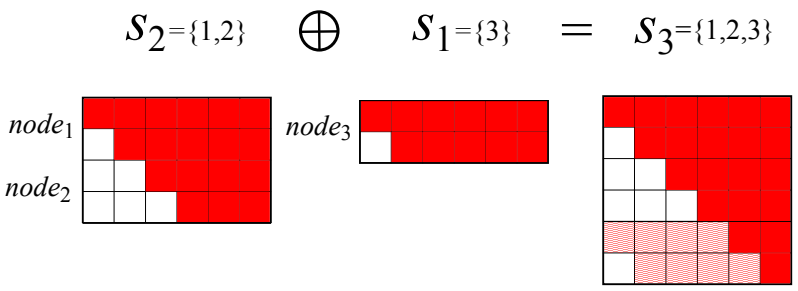


Figure 2.7: An example of merging s_2 and s_1, $\alpha = 2$.

form to reduce the number of operations in subsequent visits. The fewer matrices that need to be visited and the lower the cost of each visit, the fewer overall computations are necessary. Memory requirements only depend on the number of matrices.

We group matrices of identical size to simplify the notation and use S_i to denote the set of matrices of size $i\alpha \times k$ that includes all possible combinations of selecting all rows from i nodes. $S_i^{\text{sel}} = \{s_i | s_i \in S_i \cap \mathbf{D}\}$ is the set of matrices that are selected to be part of $\mathbf{D}$ from S_i. This grouping determines the levels of the topological sorting of $\mathbf{D}$. In order to traverse the graph, we define the merge operation $s_a \oplus s_b = s_i$, where $a + b = i$ and $s_a \cap s_b = \emptyset$ as follows. First, the rows of s_b are appended to the end of s_a. Second, the rows of s_a are used to create leading zeros in the appended rows to get the resulting s_i matrix into Upper Triangular Form(UTF). An overview is shown on Figure 2.7 with 0 elements shown as white boxes, non–zero elements shown with solid red and elements that are going to be transformed to 0s in the second step shown with a wavy red fill. The mapping Ψ associates with each matrix $s_i \in S_i^{\text{sel}}, i > 1$ a pair of matrices s_a and s_b.

A valid decomposition $\mathbf{D}$ is one that can be used to recreate all matrices in $\mathbf{M}$ in upper triangular form using merging. Furthermore, it must also provide a means to build all matrices in $\mathbf{D}$ except for matrices containing rows from a single node. For example, $\mathbf{D} = \{\{1\}; \{2\}; \{3\}; \{4\}; \{1,2\}; \{1,4\}; \mathbf{M}\}$, shown on Figure 2.6, is a valid decomposition for $\mathbf{M} = \{\{1,2,3\}, \{\{1,2,4\}, \{1,3,4\}\}$, but $\mathbf{D}' = \{\{1\}; \{3\}; \{4\}; \{1,2\}; \{2,3\}; \mathbf{M}\}$ (not pictured) is not, as neither $\{1,2\}$, $\{2,3\} \in \mathbf{D}'$ nor $\{1,3,4\} \in \mathbf{M}$ can be built from merging elements of $\mathbf{D}'$. This example highlights that for a decomposition to be valid, it must contain all matrices that contain rows from a single node as well as intermediate matrices.

We now turn our attention to studying the checks required to ensure the feasibility of repairs for RLNC. Since a naive approach involving Gaussian elimination that checks the rank of individual matrices one by one is computationally expensive, we propose decomposing the process into reusable steps.

2.4.2 Characterizing the costs associated with the checks

We would like to use decompositions that result in a small number of computations and matrices to memoize. Unfortunately, the number of valid decompositions is large and with increasing N, we quickly reach a combinatorial explosion. This section focuses on deriving the computational and storage costs associated with a decomposition and motivates the choice of algorithms for selecting effective decompositions.

2.4.2.1 The cost of reaching upper triangular form using basic Gaussian elimination

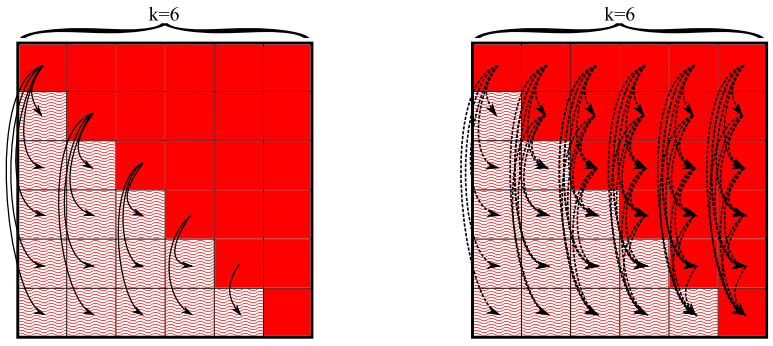
First, let us look at the cost of reaching a UTF in a $k \times k$ matrix using Gaussian elimination. This will act as a baseline for evaluating our proposed solution. We examine divisions and pairs of multiplications and additions, with an example shown on Figure 2.8 for $k = 6$. For $\alpha = 2$, this example corresponds to checking the coefficients associated with data stored on 3 nodes. Several simplifications can made to the general Gaussian elimination algorithm to save on computational cost: the back substitution step can be skipped and it is not necessary to reduce pivot elements to 1 (reduced row echelon form). Furthermore, all operations can be performed solely on the coefficient matrices.

The number of divisions is given by

$$d_{\text{Gauss}}(k) = \sum_{j=1}^{k-1} j = \frac{k(k-1)}{2}, \tag{2.13}$$

while the number of multiplications, which is the same as the number of additions is given by

$$m_{\text{Gauss}}(k) = \sum_{j=1}^{k-1} (k-j)(k-j+1) = \frac{k(k^2-1)}{3}. \tag{2.14}$$



(a) Divisions (b) Multiplications and additions

Figure 2.8: Required operations for reducing a matrix ($k = 6$) to upper triangular form using Gaussian elimination.

2.4.2.2 The cost of merging $s_a \oplus s_b = s_i$

Second, the computational cost of getting the result of merging UTF matrices of size $a \times k$ and $b \times k$ into a UTF, where $a + b = i$ and $i \geq 2$. This includes the practically important case where $a = b = i/2$.

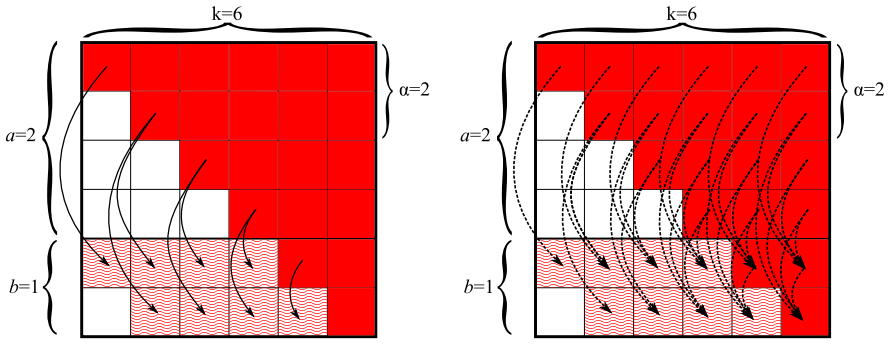
The number of 0 elements in an UTF matrix of size $i\alpha \times k$ (where $i\alpha \leq k$) gives a good indication on the number of operations that can be skipped when performing a merge:

$$lz(s_i) = lz(S_i) = lz(i) = \sum_{j=0}^{i\alpha-1} j = \frac{(i\alpha - 1)i\alpha}{2}. \tag{2.15}$$

The number of divisions is the number of elements that remain to be reduced to 0. The expression can be slightly simplified as shown in Equation (2.16).

$$\begin{aligned} d(a,b) &= lz(i) - lz(a) - lz(b) \\ &= \sum_{j=1}^{b\alpha} a\alpha = ab\alpha^2 \end{aligned} \tag{2.16}$$

Figure 2.9a shows an example for $a = 2, b = 1, \alpha = 2, k = 6$. The arrows start from the elements that need to be divided to create a leading 0 at the location they point to. The number of multiplications is the same as the number of additions required for eliminating



(a) Divisions (b) Multiplications and additions

Figure 2.9: Required operations for reducing a matrix ($k = 6$) to upper triangular form using merging $a = 2, b = 1, \alpha = 2$.

elements:

$$
\begin{aligned}
m(a,b) &= \sum_{j=1}^{b\alpha} \sum_{l=j-1}^{a\alpha+j-2} (k-l) \\
&= \sum_{j=1}^{b\alpha} \sum_{l=j-1}^{a\alpha+j-2} k - \sum_{j=1}^{b\alpha} \sum_{l=j-1}^{a\alpha+j-2} l \\
&= \sum_{j=1}^{b\alpha} k(a\alpha+j-2-j+1+1) \\
&\quad - \frac{1}{2}\sum_{j=1}^{b\alpha}[(a\alpha+j-2)(a\alpha+j-1)-(j-2)(j-1)] \\
&= ab\alpha^2 k - \frac{1}{2}\sum_{j=1}^{b\alpha}(a^2\alpha^2+2a\alpha j-3a\alpha) \\
&= ab\alpha^2 k - \frac{b\alpha}{2}(a^2\alpha^2-3a\alpha) - \frac{2a\alpha}{2}\sum_{j=1}^{b\alpha} j \\
&= ab\alpha^2 k - \frac{\alpha^2}{2}(a^2 b\alpha - 2ab + ab^2\alpha) \\
&= ab\alpha^2[k - \frac{1}{2}(a\alpha - 2 + b\alpha)] \\
&= ab\alpha^2(k - \frac{(a+b)\alpha}{2} + 1).
\end{aligned}
\tag{2.17}
$$

Figure 2.9b shows an example for $a = 2, b = 1, \alpha = 2, k = 6$. The arrows start from the elements that are multiplied with a constant and then added to the location they point to. By comparing Figures 2.8 and 2.9 we can observe that intuitively, not having to eliminate the same elements multiple times (white boxes) is the key idea behind the effectiveness of our proposed technique compared to basic Gauss elimination.

2.4.2.3 The computational cost of decompositions

Having looked at the cost of individual merges, we now turn to the total computational cost of a decomposition that checks a single repair for each node failure: the number of divisions is given by:

$$
\begin{aligned}
DIVS(\mathbf{D}) &= \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{\text{sel}}} d(a,b) + \sum_{s_1 \in S_1} lz(1) \\
&= \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{\text{sel}}} d(a,b) + N\frac{\alpha^2-\alpha}{2},
\end{aligned}
\tag{2.18}
$$

and the number of multiplications/additions is given by

$$MA(\mathbf{D}) = \sum_{i=2}^{N-L} \sum_{s_i \in S_i^{\text{sel}}} m(a,b) + \sum_{s_1 \in S_1} \sum_{j=1}^{\alpha} (k-j+1). \tag{2.19}$$

The second term in both equations is the cost of transforming matrices $s_1 \in S_1$, including the r sets of repaired rows for each of the possible node failures into upper triangular form. If matrices from previous generations are stored, then the summation can simply skip these. If a decomposition avoids memoizing matrices containing repaired rows, as per our goal, the value of r only influences the number of computations by determining the size of S_1 and S_{N-L}, which are given by

$$\begin{aligned} |S_1| &= N(r+1) \\ |S_{N-L}| &= N\binom{N-1}{N-L-1}r. \end{aligned} \tag{2.20}$$

This is why compared to Gaussian elimination the cost does not scale linearly with r, increasing rather more gently.

The number of reusable matrices depends non–trivially on the decomposition. We leave investigating this aspect as future work and present experimental results in Section 2.4.5.

2.4.2.4 The memory requirements of decompositions

Our proposed approach requires memory to store memoized matrices. The number of matrix elements that need to be stored for decomposition $\mathbf{D}$ is given by

$$MEMO_{\text{naive}}(\mathbf{D}) = \sum_{i=1}^{N-L} |S_i^{\text{sel}}| i\alpha k. \tag{2.21}$$

A simple improvement can be achieved by only storing non–zero elements:

$$MEMO_{\text{reduced}}(\mathbf{D}) = \sum_{i=1}^{N-L} |S_i^{\text{sel}}| (i\alpha k - lz(i)). \tag{2.22}$$

2.4.3 Finding efficient decompositions

2.4.3.1 Observations on the relationships between costs

Based on the Equations (2.16) and (2.17), we can make some observations that will help in determining good decomposition strategies. Let $a, b, a', b', c, d \in \mathbb{N}^+$, $s_b \in S_b$, $s_c \in S_c$, $s_d \in S_d$ and $a + b = a' + b' = i$ and $c + d = b$.

1. $m(a, b) = m(b, a)$ and $d(a, b) = d(b, a)$ (because a and b are interchangeable in Equations (2.16), (2.17))

2. $d(a, b) > d(a', b') \Leftrightarrow |a - b| < |a' - b'|$ (consequence of Equation (2.16) and because more imbalanced sets have a larger number of already reduced elements; the larger the matrix, the more reduced elements it has)

3. $m(a, b) > m(a', b') \Leftrightarrow |a - b| < |a' - b'|$ (consequence of Equation (2.17) in which the part in brackets is the same for a, b and a', b')

4. $m(a, b) > m(a', b') \Leftrightarrow d(a, b) > d(a', b')$ (same argument as previous observation)

5. $lz(s_b) \geq lz(s_c) + lz(s_d)$ (consequence of Equation (2.15) and $i^2 > a^2 + b^2$)

6. $d(a, b) \leq d(a, c) + d(a + c, d)$ (consequence of previous observation and Equation (2.16))

Observation 4) is important because a decomposition that minimizes the number of divisions will also minimize the number of multiplications and additions. Observation 2) and 3) have the consequence that a decomposition that decreases i by one (i.e. selecting $a = 1, b = i-1$ or $b = 1, a = i-1$) has the lowest computational cost in that decomposition step. We refer to this method as *decrease and conquer*. Conversely, $a = \left\lfloor \frac{i}{2} \right\rfloor$ or $b = \left\lfloor \frac{i}{2} \right\rfloor$ has the highest number of computations for any given i. On the other hand, it also reduces the size of the matrix by the greatest degree. Thus matrices between $S_{\lfloor i/2 \rfloor}$ and S_i can be skipped and less space is needed. We refer to this approach as *divide and conquer* and propose the following method to deal with odd levels: if i is even, divide the problem into $a = b = \frac{i}{2}$. If i is odd, fall back to the previous approach and decrease the problem to $a = a - 1$, $b = 1$. An alternative divide and conquer decomposition would be to select $a = \left\lceil \frac{i}{2} \right\rceil$ and $b = \left\lfloor \frac{i}{2} \right\rfloor$, then do a second decomposition if $a \neq b$ to cover S_b using S_a and S_1.

There is therefore a trade–off between minimizing the number of levels (and reducing memory requirements in the process) in a decomposition and the cost of moving between levels using merging. However, it is not immediately apparent how the number of matrices in each level ($|S_i^{\text{sel}}|$) changes for different points on the trade–off curve. This metric also

plays a key role in determining the number of computations. In the following subsections, we propose two heuristic–based algorithms to find decompositions for both the decrease and conquer and divide and conquer methods.

2.4.3.2 The first step of a decomposition

As described in Section 2.4, matrices containing rows from $N-L-1$ existing nodes and one hypothetically repaired node must be checked. If we consider that r hypothetical repairs are checked for each possible node failure, $|M| = (L+1)r\binom{N}{N-L-1}$. Instead of starting from these matrices, the first step of a decomposition should be treated differently to ensure that no repaired rows are present in any matrix in S_{N-L-1}. This is because it is not known which node will fail at the time of the checks and hypothetical repaired rows severely limit the reusability of matrices.

To account for this, we propose a simple schema to determine the first step of the decomposition. The matrices in S_{N-L} can be built in the following way: by selecting $S^{\text{sel}}_{N-L-1} = S_{N-L-1}$ (i.e. taking all possible matrices that contain $N-L-1$ non–repaired rows) and adding every possible repaired row to each of them, we arrive at S_{N-L}. This allows the decomposition to start from $\mathbf{M} = S_{N-L-1}$ instead of S_{N-L} and only include coefficients from existing rows to maximize matrix reuse. We will show in Section 2.4.3.3 that this first step is optimal in selecting the minimal number of matrices from S_{N-L-1}.

2.4.3.3 Greedy algorithm for decrease and conquer

We wish to select S_{i-1} in such a way that all elements of S_i^{sel} can be generated by adding an element of S_1 to an element of S_{i-1}. Let $G = (V, E)$ be an undirected bipartite graph with vertices divided into sets $V = X \cup Y$, where $X = \{s_i | s_i \in S_i^{\text{sel}}\}$ and $Y = \{s_{i-1} | s_{i-1} \in S_{i-1}^{\text{all combos}}\}$. There is an edge between a vertex $v_x \in X$ and $v_y \in Y$ if and only if for the corresponding s_i and s_{i-1}, $s_{i-1} \subset s_i$. We wish to cover all vertices $v_x \in X$ using as few vertices $v_y \in Y$ as possible. At each step in the algorithm, the vertex v_y with the highest degree is selected and removed from the graph. All vertices v_x it is connected to are also removed along with any edges containing v_x. The algorithm terminates when there are no more vertices in X.

This greedy algorithm is well known and is analogous to the approach of selecting a covering set in such a way that at every choice, the set that covers the most uncovered elements is selected. This is a $H(\mathbf{n})$–approximation algorithm and it has been proved [Dinur and Steurer, 2014] that no polynomial–time algorithm with a better approximation factor exists for this NP–hard problem. Fortunately, $\mathbf{n} = \max |s_i| = i$, as all matrices from the set used for the cover have exactly i elements. Thus, even though

Algorithm 2.3 Greedy algorithm for decrease and conquer

```
build G(i, 1)
while X ≠ ∅ do
    find v_y, where deg(v_y) ≥ deg(v_j), ∀v_j ∈ G
    S^sel_{i-1} = S^sel_{i-1} ∪ v_y
    Y = Y \ v_y
    for all v_x, where (v_x, v_y) ∈ E(G) do
        X = X \ v_x
        E(G) = E(G) \ (v_x, v_*)          ▷ all edges involving v_x
    end for
end while
return S^sel_{i-1}
```

$|S_i| = \binom{k}{i}$ increases computational costs quickly, the approximation factor increases slowly $\left(\frac{\log i}{\log(i-1)}\right)$ with i and remains acceptable even for large values of i.

We could apply this algorithm for the first step of finding a decomposition and it would select a covering set that is the combination of $N - L - 1$ out of N rows.

Proposition 2.6. *If r repairs are checked for each possible node failure, then $|S_{N-L}| = |S_{N-L-1}|(L+1)r$, which is the same as our previously proposed first step and is in fact the best we can hope to achieve.*

Proof. Let G be a graph with vertices coming from matrices in S_{N-L} and edges between pairs of matrices that differ by α rows as shown on Figure 2.10. It is relatively easy to show that the clique number of G will be $\omega(G) = (L+1)r$ and that each clique can be defined using a submatrix of the rows that are common in each of the matrices in the clique. Thus, the best covering set will be the set of submatrices that define each clique as this covers all matrices using the least amount of smaller matrices. □

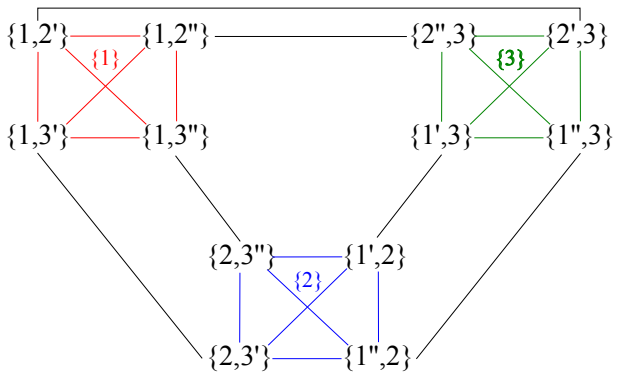


Figure 2.10: The shared rows of matrices in $\mathbf{M}$ that should be checked for $N = 3$, $L = 1$, $r = 2$. Single and double apostrophes denote rows with coefficients from potential repairs. For example, $2'$ and $2''$ are rows resulting from the two different potential repairs of $node_2$.

We have decided to follow the technique proposed in Subsection 2.4.3.2 instead of this algorithm for the first step of a decomposition to ensure that if multiple minimum cost set covers exist (this is the case for $r = 1$), the one that maximizes matrix reuse through memoization is selected.

2.4.3.4 Greedy algorithm for divide and conquer

We propose extending the previous algorithm to deal with the more general case when a matrix of size $i \times k$ is decomposed into two submatrices of size $a \times k$ and $b \times k$, where $a + b = i$. Figure 2.11 shows an example for a state of the algorithm. We keep the graph G to have a pairing that denotes which submatrices cover which matrices. The set X remains the matrices that need to be covered and Y contains all possible submatrices of size $a \times k$ and $b \times k$ that can be built from elements of S_1. There is an edge between a vertex $v_x \in X$ and $v_y \in Y$ if and only if $v'_y \notin Y$, where $v'_y := v_x \setminus v_y$. In other words, v'_y is the relative complement of v_y with respect to v_x, the rows of v_x left uncovered by v_y. We refer to it as its pair.

Unless $a = 1$ or $b = 1$, G has no edges in the beginning because $\forall v_y \in Y \rightarrow v'_y \in Y$. Put differently, in the beginning no single element in Y can cover an element in X. In the special case of $a = 1$ or $b = 1$, G is initialized as described in the previous subsection.

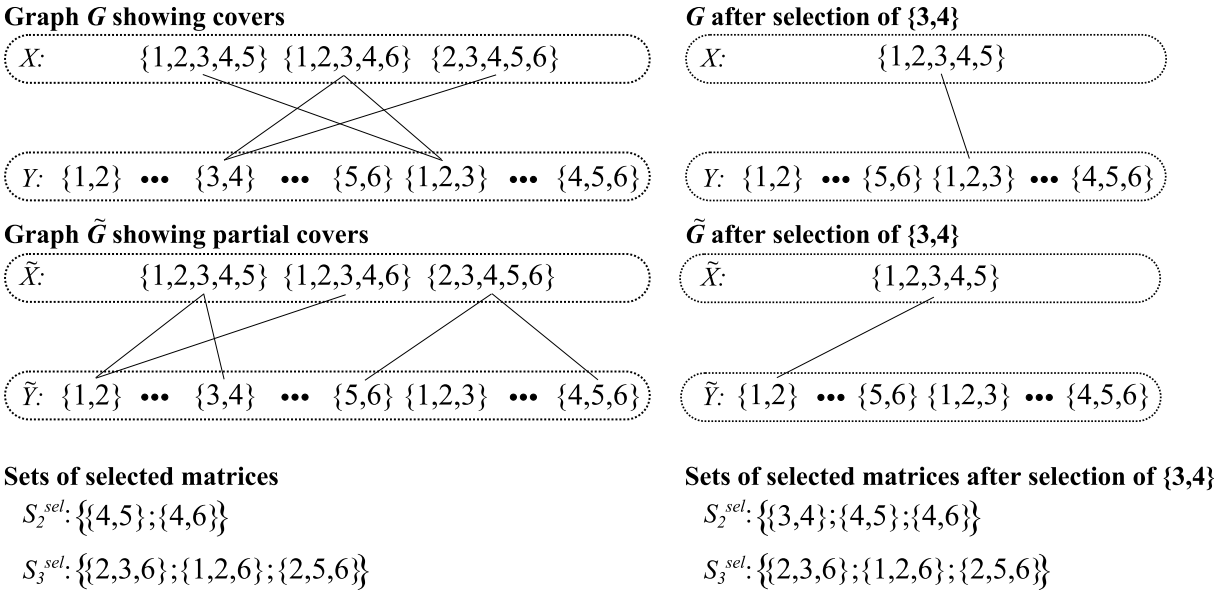


Figure 2.11: Example for a state change of the divide and conquer algorithm showing both graphs and the sets of already selected matrices. Some elements of Y and $\tilde{Y}$ have been omitted (...) due to space constraints. The next step (result shown on the right) selects and adds $\{3,4\}$ to S_2^{sel} as it is tied with $\{1,2,3\}$ in covering the most matrices in G and partially covers more matrices in $\tilde{G}$. The algorithm then removes $\{3,4\}$, $\{1,2,3,4,6\}$ and $\{2,3,4,5,6\}$ from G and $\tilde{G}$ along with all of their edges. It then adds an edge between $\{1,2,3,4,5\}$ and $\{1,2,5\}$ (not shown) in G to reflect that $\{1,2,5\}$ now fully covers $\{1,2,3,4,5\}$.

Algorithm 2.4 Greedy algorithm for divide and conquer

Build $G(a,b), \tilde{G}(a,b)$
while $X \neq \emptyset$ **do**
  $MAX = \{v_y | v_y \in G, deg(v_y) \geq deg(v_j), \forall v_j \in G\}$
  find $\tilde{v_y} \in MAX$, where $deg(\tilde{v_y}) \geq deg(\tilde{v_j}), \forall v_j \in \tilde{G}$
  if $|v_y| = a$ **then**
    $S_a^{\text{sel}} = S_a^{\text{sel}} \cup v_y$
  else
    $S_b^{\text{sel}} = S_b^{\text{sel}} \cup v_y$
  end if
  $Y = Y \setminus v_y$
  $\tilde{Y} = \tilde{Y} \setminus \tilde{v_y}$
  $E(G) = E(G) \setminus (v_*, v_y)$ ▷ all edges involving v_y
  $E(\tilde{G}) = E(\tilde{G}) \setminus (\tilde{v_*}, \tilde{v_y})$ ▷ all edges involving $\tilde{v_y}$
  for all v_x covered by v_y **do**
    $X = X \setminus v_x$
    $\tilde{X} = \tilde{X} \setminus \tilde{v_x}$
    $E(G) = E(G) \setminus (v_x, v_*)$ ▷ all edges involving v_x
    $E(\tilde{G}) = E(\tilde{G}) \setminus (\tilde{v_x}, \tilde{v_*})$ ▷ all edges involving $\tilde{v_x}$
  end for
  for all $\tilde{v_x}$ partially covered by $\tilde{v_y}$ **do**
    $E(\tilde{G}) = E(\tilde{G}) \cup (\tilde{v_y}', \tilde{v_x})$, where $\tilde{v_y} \cup \tilde{v_y}' = v_x$
  end for
end while
return $S_a^{\text{sel}}, S_b^{\text{sel}}$

We introduce a second graph $\tilde{G}$ ($\tilde{X} = X$, $\tilde{Y} = Y$), to have a pairing that denotes which submatrices cover which matrices partially. $v_y \in \tilde{Y}$ partially covers $v_x \in \tilde{X}$ if its pair $v_y' \in \tilde{Y}$. These are submatrices, whose pairs have not yet been selected and thus can only provide partial cover for v_x. $\tilde{G}$ must be initialized to have all partially covering edges, thus in the initial state of the algorithm $E(\tilde{G}) = \{(v_x, v_y) \,|\, v_x \setminus v_y \in \tilde{Y}, v_x \in \tilde{X}, v_y \in \tilde{Y}\}$. In the special case where $a = 1$ or $b = 1$, $\tilde{G}$ will have no edges and may be disregarded as the algorithm falls back to the previously described decrease and conquer algorithm.

Algorithm 2.4 selects submatrices from Y until all matrices in X have been covered. It selects the submatrix v_y that has the highest degree in G, i.e. covers most matrices. Tie–breaks are common and are handled by selecting the submatrix that has the largest degree in $\tilde{G}$, i.e. partially covers most matrices. When a submatrix is selected, it is removed from both G and $\tilde{G}$. Furthermore, all matrices it covers are removed from both G and $\tilde{G}$ along with any edges they are part of. Any matrix $\tilde{v_x}$ it partially covered is updated in G: an edge is added between v_x and v_y' to reflect that v_y' can cover v_x following the selection.

Proposition 2.7. *For every partially covered v_x by the selected v_y, there will always be exactly one v'_y that will cover v_x.*

Proof. If we supposed that no v'_y exists, then it must have been previously selected. However, in this case v_x would have already been fully covered by v_y. We have reached a contradiction. Furthermore, it is unique, because no two permutations of the same elements exist in Y. □

2.4.4 Performing invertability checks given a decomposition

A decomposition is only dependent on N, L, α, parameters that typically do not change during the lifetime of a system. Therefore, several decompositions can be computed in advance to cover the likely parameter set values before the system comes online. Thus, even if finding a good valid decomposition is computationally expensive, it does not negatively influence the general repair performance of the system. This subsection examines how a decomposition can be applied.

2.4.4.1 The benefits of performing checks in a top–down manner

Given a valid decomposition $\mathbf{D}$, either a bottom–up or a top–down approach can be used to do the checks. Bottom–up checks start with matrices from S_1^{sel} and then reduce each matrix $s_j \in S_j^{\text{sel}}$ level–by–level to an upper triangular form and memoize it after merging two smaller matrices based on Ψ. This approach has the benefit of avoiding recursive calls, but will only provide relevant information on the rank of a matrix $s_m \in \mathbf{M}$ after all smaller matrices $s_j \in S_j$, $j < m$ have been reduced to an upper triangular form. Conversely, a top–down approach starts with matrices $s_m \in \mathbf{M}$ and attempts to merge $s_m = s_a \oplus s_b$, where the choice of s_a and s_b is defined by Ψ. If either s_a and/or s_b are not yet in UTF, the algorithm is called recursively on s_a and/or s_b and so on. Once a matrix is reduced into UTF, it is memoized so it can be reused for other matrices from $\mathbf{M}$. Thus, the invertability of some s_m will be known earlier than using a bottom–up approach. This can be used to provide probabilistic statements on the overall result of the checks before all matrices are checked.

This is most important in situations where the checks are time constrained. Either there are not enough free computational resources in the system and the checks must be postponed or two node failures occur in quick succession and a repair must be started before the checks for the second failure have time to complete. We might still be able to make a more informed decision on what repair to select based on the matrices that have been computed so far. We can increase the amount of useful information at any given point in time by using a top–down approach and ordering the matrices in M in a certain

way. One possible ordering is to take one repair for each node failure first. Once one feasible repair is found for every failure, we can check one more repair for each failure and so on. This minimizes the time until we have at least one feasible repair for every possible failure scenario. Alternatively, if it is acceptable for the system to temporarily go below the predetermined L number of concurrent node failures it must sustain, we may order the matrices differently. In this case, we may also start by looking at a single repair for each node failure first. However, instead of completing all checks for that repair, we would do one check at a time, moving on to check matrices associated with another node failure. With each pass, the degree of confidence with which we can state that a repair is feasible for any L node losses increases. If a check fails, we can generate new coefficients and redo the checks that have already been performed for the failed check. We leave the formal problem definition and evaluation for the problem of ordering matrices in M as future work.

2.4.4.2 Memoizing matrices across generations

A further reduction in computations can be achieved if matrices from a decomposition are stored and reused across multiple generations. Any matrix that contains rows from the recently lost node should be discarded, but all others can be reused in the subsequent generation. The proposal to start a decomposition from $\mathbf{M} = S_{N-L-1}$ described in subsection 2.4.3.2 also encourages matrix reuse across generations as it ensures that only matrices containing rows from actual nodes (as opposed to rows from hypothetical repairs) are memoized. We leave the characterization of the choice of decomposition on the expected number of matrices that can be reused as future work and present simulation–based results to show the effectiveness of this enhancement. We also note that the checks for the initial data distribution are more expensive as they cannot reuse matrices from previous generations and no matrices can be skipped as there were no previous failures.

2.4.5 Experiments

To test the effectiveness of the proposed decomposition techniques, we have implemented both divide and conquer and decrease and conquer algorithms in Python. We removed the constraint on the size of the repair space to include a total of 629 RLNC codes. We can apply the decrease and conquer approach to all of these. The divide and conquer approach falls back to decrease and conquer in all but 348 cases. The figures have had the 281 duplicate cases removed for clarity.

2.4.5.1 The benefits of performing decompositions

Figure 2.12 shows the benefits of using our proposed technique for $r = 1$. By decomposing the rank checks and reusing intermediary results, a reduction of up to 87% is achieved in the number of multiplication/addition operations compared to doing Gaussian eliminations on the matrices in $\mathbf{M}$. The gains grow in significance with the number of operations for both types of decomposition, making checks feasible for a wider range of parameters. Divisions show a similar trend, though in a few cases we actually see an increase in the number of computations and results show more variance.

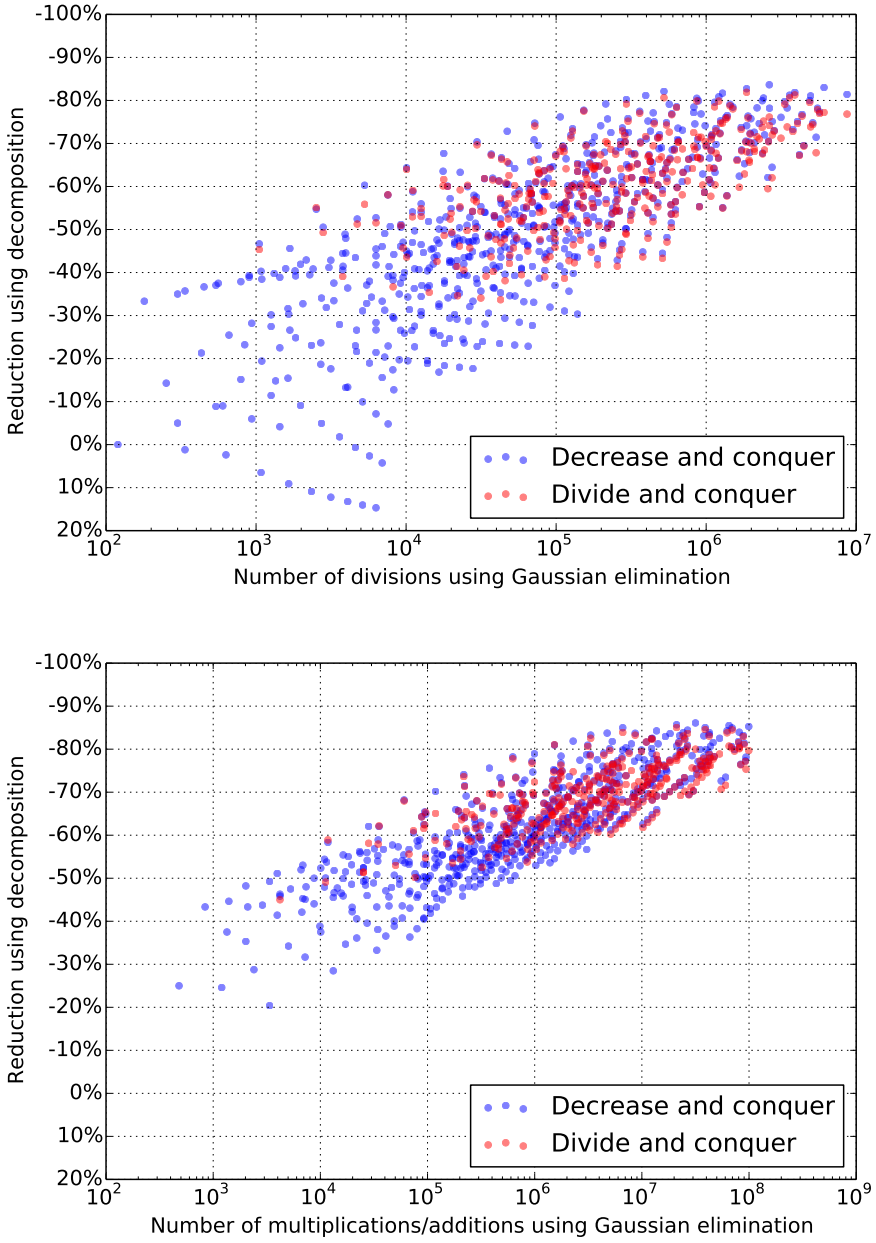


Figure 2.12: Reduction in the number of operations compared to Gaussian elimination

Looking at different values for r on Figure 2.13, we can see how the difference between the computational cost difference increases as r grows. This is as expected based on Equation (2.20). All other figures show simulations for $r = 1$.

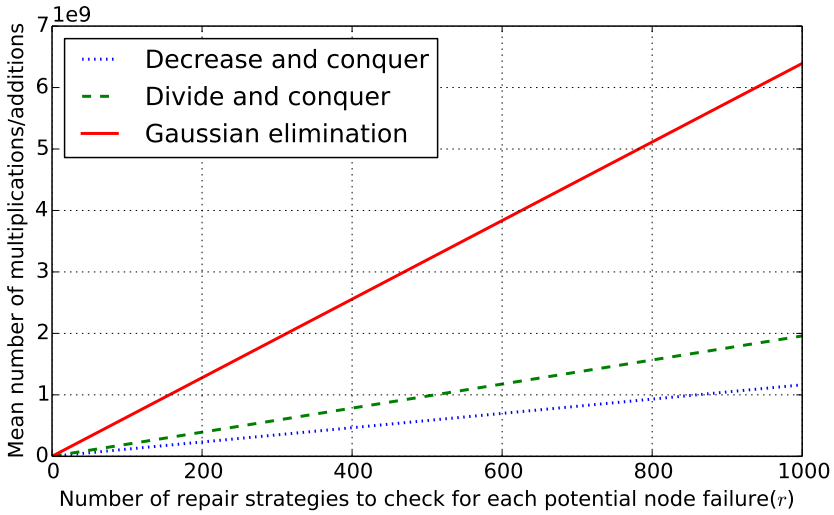


Figure 2.13: Checking the feasibility of multiple repair strategies for each node failure

2.4.5.2 Comparing divide and conquer to decrease and conquer

Section 2.4.3 briefly described the trade–off between the number of memoized matrices and the number of operations. Figure 2.14 illustrates a direct comparison between the two proposed methods on this aspect. Divide and conquer requires up to 37% less storage than decrease and conquer. The downside is an increase of up to 33% in the number

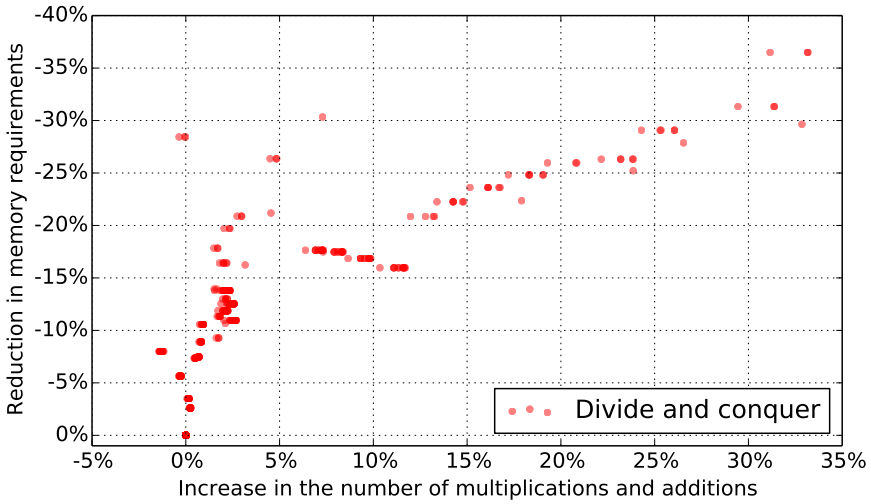


Figure 2.14: Comparing divide and conquer to decrease and conquer on storage and computation costs

of multiplications/additions. The memoized matrices for the examined codes typically require around 200kB – 300kB of storage, with a maximum of 6.2MB. This is negligible compared to the data associated with the coefficients. Thus, most practical systems should employ decrease and conquer to reduce the amount of computations as much as possible. Division operations show a similar trend and have been left out for brevity.

2.4.5.3 Memoizing across generations

Finally, we look at the benefit of reusing matrices across generations of node failure and recovery. We expected to show further large gains in computation cost by extending our solution with this simple technique because most matrices remain valid through at least two subsequent generations. However, results on Figure 2.15 only show relatively modest gains of between 5%-15%. This can be explained by looking more closely into which matrices can be reused. While around $\frac{N-1}{N}$ of the smallest matrices contain rows that are present in two subsequent generations, the ratio is smaller for larger matrices. Furthermore, most of the computations can be associated with the larger matrices. Nevertheless, the modest additional storage cost makes reusing matrices across generations a good choice to further reduce computational costs.

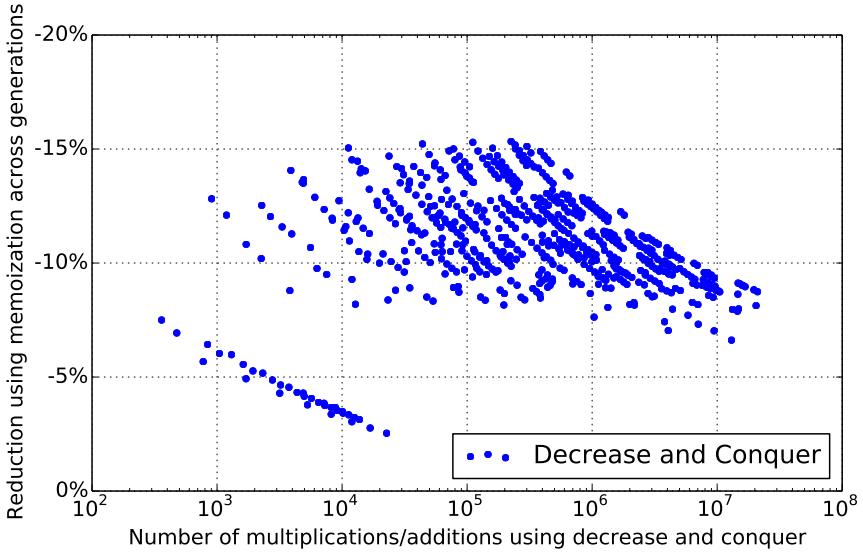


Figure 2.15: Benefits of reusing matrices across generations

2.5 Conclusions

Increasing the proportion of erasure coded data in complex data center applications is a daunting task both in terms of system design and theoretical models. While the economics of large–scale data storage put pressure on service operators to adopt erasure coding in order to decrease the $/GB cost while maintaining a high level of reliability, many obstacles remain. The goal of this chapter has been to help solve some of these practical and theoretical challenges.

First, we looked at what are the theoretical limits of repair efficiency for MSR codes. We presented a simple formula that can be easily used to find the potentially minimal cost feasible repairs for network codes.

Second, we answered a key question concerning the design of erasure coding for data centers: knowledge of network conditions benefits a wide range of erasure codes. We have proposed a framework that checks the feasibility of repairs in advance and provided analytic results on reducing the space of relevant feasible repairs. We hope to make further advances in this area for specific network topologies by looking at specific types of cost matrices.

Third, we have presented a set of techniques to increase the practical value of our work for erasure codes employing randomly selected coefficients, such as RLNC. Our proposed decomposition of matrix rank checks shows significant reductions in the number of computations needed and has negligible storage costs. We presented two algorithms to perform the decomposition and found that in practice a decrease and conquer type solution works better. We wish to continue this work by formalizing the problem of ordering the matrices that are to be checked in such a way that useful information on their rank is gained as soon as possible.

Chapter 3

Distributed cloud storage using RLNC

3.1 Introduction

Cloud storage is widely adopted as it offers a cost–effective solution to storing enterprise data, with the advantages of increased reliability as well as arguably decreased technical complexity and business agility compared to on–site, personalized storage solutions. Its adoption among end users is growing as well [Gartner, 2012] thanks in part to the free storage space offered by major IT players, e.g., Amazon, Google, Microsoft, Apple, and specialized cloud storage companies, e.g., Dropbox, Box, SugarSync, as well as the additional free storage offered with new smart mobile devices.

We have looked at applying network coding to store data safely and efficiently in data centers in the previous chapter. We argued that it is flexible in the sense that it can be used to achieve different points on the storage efficiency–repair efficiency curve. Applying it over an aggregated cloud storage systems reveals additional benefits to its flexibility that are due to the more dynamic nature of this scenario. First, its MDS–like properties should help mitigate the effect stragglers have on retrieval performance. Second, its ability to create new coded packets as a combination of existing ones through recoding can be used for more than just repair, and could serve as a highly tunable mechanism to adapt the distribution of data. Third, its rateless nature makes it easy to create new data to account for a reduction in the number of clouds in the system. We have chosen to study RLNC over other codes used for storage [Huang et al., 2012b, Sathiamoorthy et al., 2013b, Lei et al., 2013, Le and Markopoulou, 2012] to highlight the practical benefits of these three characteristics.

3.1.1 Structure and overview of the contributions of this chapter

Our first contribution is an in–depth analysis of the benefits of using network coding to aggregate multiple cloud providers, compared to the single cloud approach. We show through theoretical results that spreading the content over multiple clouds provides higher reliability and availability and also requires a malicious attacker to break into multiple clouds to access a user's content. We formalize the different requirements users might have and define the optimal data distribution strategy for MDS codes as an Integer Programming(IP) problem.

Our second contribution is a more detailed look at retrieval performance. We show through measurements using four cloud providers (Box, Dropbox, Google Drive and OneDrive) that download speed can be significantly improved when using multiple clouds judiciously. We also included a detailed comparison to widely–used replication–based codes. We propose a simple scheduling algorithm for replication and discuss why an RLNC–based approach leads to better and more consistent performance.

As our third contribution, we address the problem of dynamically adapting the data distribution in the face of changing provider performance characteristics to maintain optimal download times. Central to the solution is the efficient exchange and delivery of additional data without incurring in high coding/decoding/transmission overheads. We address these questions by proposing a robust and efficient solution with an implementation that employs five cloud storage providers, namely the previously mentioned four and SugarSync. At the core of our scheme lies the idea of maintaining high reliability of *critical data storage* using a dense* code, while using an efficient sparse recoding mechanism for providing different clouds with *performance enhancing storage*. The former type corresponds to the minimal storage needed to guarantee reliability of the data, which will only require a reconfiguration if a cloud joins, leaves, or is in outage. The latter is meant to handle the dynamics in download speed of the various cloud providers and, thus, requires changes in a shorter time–scale. To strike a good balance between network use for download of data and network use for adaptation to changing conditions, we propose a sparse code. This results not only in low network use for the latter, close to optimal performance in the former, but also a reduced encoding and recoding computational effort. We define this problem formally and refer to it as the *recoding bandwidth problem*. We also devised and implemented a dense recoding mechanism and two simple replication–based approaches to use as comparison to our sparse solution. Sparse recoding shows performance very close to the dense approach while using up to 9x less adaptation traffic.

*A coded packet is said to be dense when it is generated from a linear combination of original or other coded packets using a large number of non–zero coefficients. Conversely, a sparse packet is a linear combination of only a few non–zero coefficients.

We briefly touch on handling changes in the number of clouds and show measurement results on the overall robustness of the system.

This chapter is organized as follows, Section 3.2 describes the structure of the system as well as the theoretical results on the main characteristics of multi–cloud network coded storage systems. Section 3.3 focuses on characterizing the retrieval performance of the system. Section 3.4 introduces the recoding bandwidth problem and proposes four possible solutions to solve it. It also looks at how to adapt to a change in the number of clouds. Finally, Section 3.5 presents the conclusions.

3.2 Optimizing the data distribution scheme

We present a system that employs commercially available clouds to store files reliably using RLNC. An example is given on Figure 3.1 for four clouds. Given the similar challenges, our solutions can be adapted for use with other types of storage nodes as well. It is comprised of a client application that uploads and downloads data to the storage nodes and handles all computations related to encoding and decoding. The storage nodes have no other

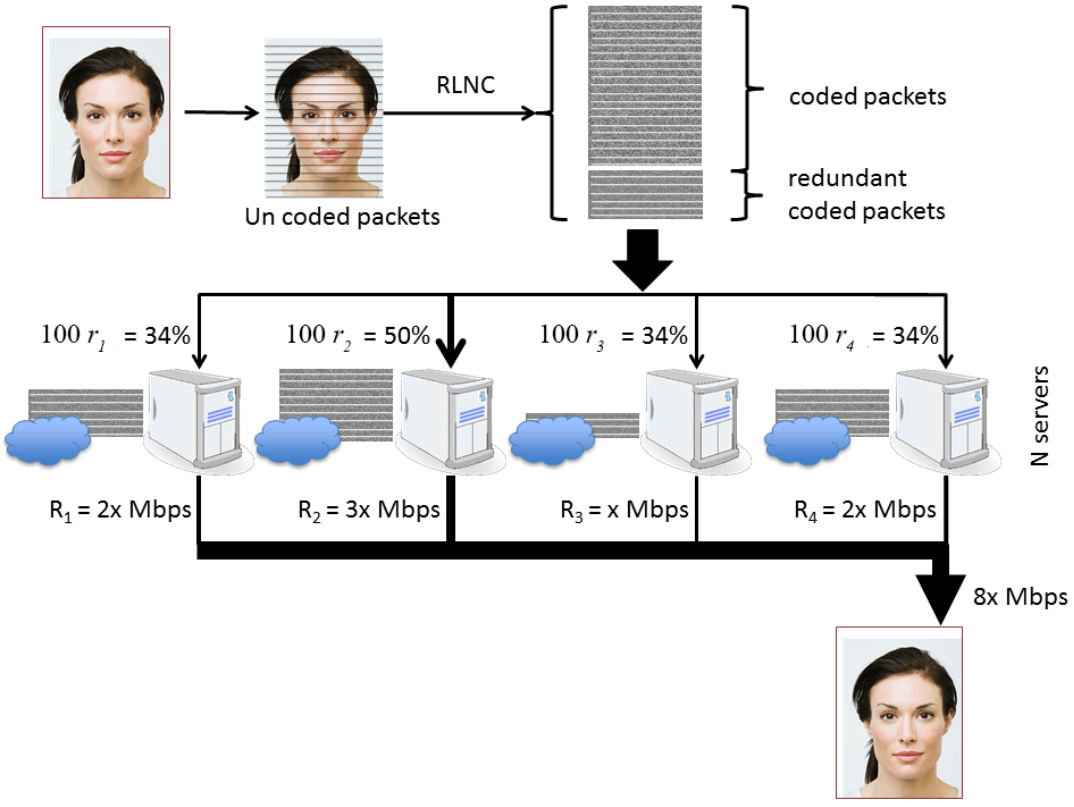


Figure 3.1: Main idea of distributed clouds with network coding. For example, $N = 4$ clouds available with different download data rates(R_i), x $\in \mathbb{R}^+$. The system allocates different amounts of data to decrease retrieval time. It stores a minimum of 34% of the original data on each cloud to ensure data availability if at most one cloud is not reachable.

functionality besides storing the data, which makes employing recoding techniques that involve creating new linear combinations on the nodes impossible. This is a limitation of several commercial clouds. Thus, any changes to the data must be directed by either the client application or a long–running service. We have built a model without this limitation and explored other types of recoding separately in the following chapter.

A total of n encoded packets are distributed by the client a priori to the clouds, with c_i storing α_i. At least k must be retrieved to recover the file. Let us derive the performance measures of our approach. We consider a set of N cloud providers $\mathcal{C}$, where $\mathcal{C} = \{c_1, c_2, ..., c_N\}$ and c_i represents the i–th cloud provider. We consider that a single user uses this set of clouds to upload and download a file of size F using network coding. The download rate from the i–th cloud is R_i.

3.2.1 Reliability

If we consider that a cloud provider c_i, has a probability p_i of being unavailable (e.g., service is down, data is lost), then the probability of all data being unavailable to the user given that it has used L redundant clouds is

$$\mathcal{P}\left(L, N, \{p_i\}\right) = 1 - \sum_{a_1+...+a_N \geq N-L} \prod_{i=1}^{N} (1-p_i)^{a_i} p_i^{1-a_i}, \tag{3.1}$$

where $a_i \in \{0, 1\}$ indicates that cloud i is down or not for $a_i = 0$ and $a_i = 1$, respectively and $0^0 = 1$. For the case of $p_i = p, \forall i = 1, ..., N$, this simplifies to

$$\mathcal{P}(L, N, p) = 1 - \sum_{t=0}^{L} \binom{N}{t} (1-p)^{N-t} p^t. \tag{3.2}$$

3.2.2 Download Speed

In order to maximize the download speed from the clouds, we need to consider that the amount of data stored need not be the same as provided by the reliability criteria. The latter is an indication of the minimum amount of data that can be stored, but it does not consider the benefits of downloading more data from faster clouds. We consider that each cloud c_i provides a download rate of R_i and that F is large enough so that the download time is much larger than the individual round trip times from each server. Our goal is then to request a fraction r_i from cloud c_i such that all clouds complete the delivery of the data simultaneously. Thus, the fraction of the file requested from cloud c_i is

$$r_i = \frac{R_i}{\sum_{i=1}^{N} R_i}. \tag{3.3}$$

If we consider there is asymmetric round trip delays until the packets are received, where D_i represents the round trip delay from cloud c_i, then

$$r_i = \frac{R_i}{\sum_{i=1}^{N} R_i} \left(1 + \frac{\sum_{c_j \in \mathcal{C}} R_j (D_j - D_i)}{F} \right). \tag{3.4}$$

If both reliability and download speed are important factors in the design, then the fraction of the data stored, P_i, stored in cloud c_i needs to be

$$P_i = \max \left(\frac{1}{N-L}, r_i \right). \tag{3.5}$$

3.2.3 Privacy and Security

We can consider different conditions for data privacy. The most stringent is to force attackers to break into at least $N - L$ clouds to obtain the data assuming that the link to the user is encrypted during download/upload. This condition is of course fulfilled if we only store enough for guaranteeing reliability. If we store additional data for boosting download speed performance, then we need to set a different condition. A simple way of thinking about it is that any combination of the fractions of data of a subset of $N-L-1$ clouds cannot amount to 1, i.e., enough degrees of freedom to decode.

Of course, $P_i < 1/(N - L - 1), \forall i$ is a sufficient, but not necessary condition to preserve privacy. A necessary condition for preserving privacy is stated in the following. Let us define $\mathbf{C}_m$ as the set of distinct subsets of $\mathcal{C}$ missing exactly m cloud elements each. For example, $\mathbf{C}_1 = \{\mathcal{C} \setminus c_1, \mathcal{C} \setminus c_2, ..., \mathcal{C} \setminus c_N\}$, and $\mathbf{C}_2 = \{\mathcal{C} \setminus \{c_1, c_2\}, \mathcal{C} \setminus \{c_1, c_3\}, ..., \mathcal{C} \setminus \{c_{N-1}, c_N\}\}$

Then, as long as

$$\max_{C \in \mathbf{C}_{L+1}} \sum_{c_i \in C} P_i < 1, \tag{3.6}$$

the condition for security is preserved. This provides an interesting result, namely, that there is room for providing higher download speeds without compromising security albeit with some additional storage cost. From a practical perspective, a simple approach to test the condition is to sort P_i's in increasing order. Then, we can add the $N - L - 1$ greatest values. If the result is lower than 1, then privacy is preserved. Otherwise, it is compromised.

3.2.4 Storage Costs

For the case in which reliability is the only concern, the overall storage size H, that is needed for the distributed approach when a file of size F bytes is used equals

$$H(N, L, F, P) = \frac{N}{N-L} \cdot F = F + F\frac{L}{N-L}, \tag{3.7}$$

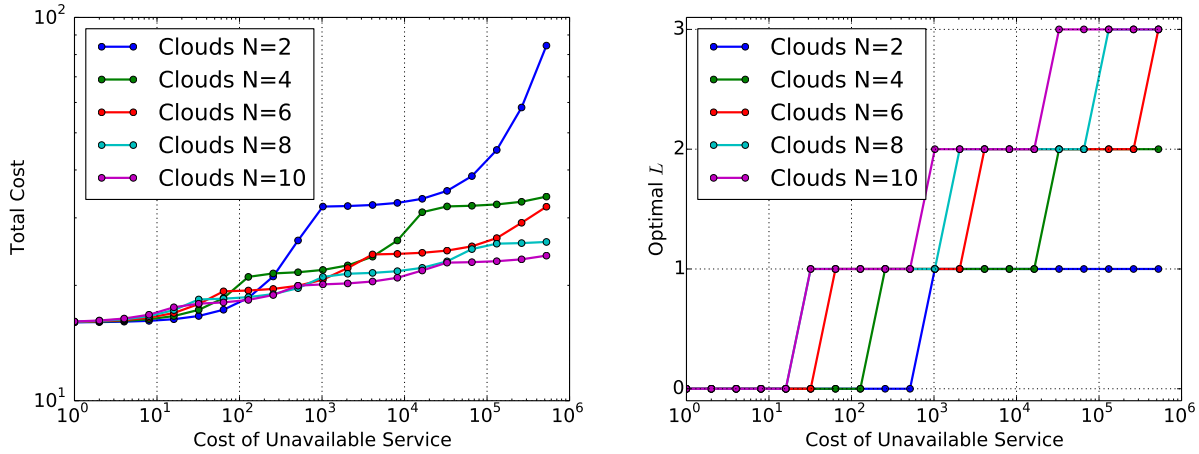
where we assume that each cloud will contain a fraction $P = 1/(N-L)$ of the data. Clearly if $N >> L$, the storage cost becomes negligible. This is an important distinction from replication–based solutions, where the storage cost associated with increasing redundancy grows linearly with L, the number of cloud outages the system can sustain. If more factors are important, then

$$H(N, L, F, \{P_i\}) = F \sum_{i=1}^{N} P_i. \tag{3.8}$$

3.2.5 Overall costs

Let us consider that we have a storage cost of $H = \frac{N}{N-L} \cdot F$ and a cost of having the data unavailable U_c and both can be expressed as a monetary cost. The optimal choice for L will be given by the problem

$$\min_L (H(N, L, F, P) + U_c \cdot \mathcal{P}(L, N, \{p_i\})). \tag{3.9}$$



(a) Overall Cost

(b) Optimal number of redundant clouds

Figure 3.2: Cost of aggregated cloud storage for $F = 16$ and unitary storage cost

Figure 3.2a and Figure 3.2b show that the overall cost can be maintained low by using a higher number of cloud providers, even if the number of redundant ones is high. For example, $N = 10$ provides not only a low overall cost but also allows for a configuration that is stable for a wider regime of U_c.

3.2.6 Data distribution scheme as an integer programming problem

Based on the previous constraints, let us formally introduce an integer programming problem that aims to minimize storage cost while meeting reliability requirements and ensuring maximal retrieval performance.

Consider an initial state of the system, where the data distribution is symmetric $\alpha_1 = \alpha_2 = \ldots \alpha_N = \lceil k/(N-L) \rceil$. If the download rates (R_i) differ significantly from each other, then the performance of the system is suboptimal. More precisely, if for any i Equation (3.5) is not satisfied, then c_i is not used to the entirety of its capacity from a performance point of view. Therefore, if it is possible to ascertain the values of R_i for all i, the values of r_i can be calculated using equation (3.3) and the initial distribution can be modified to satisfy Equation (3.5). This adaptation maximizes the retrieval performance of the system.

Let κ_i be the cost of storing one packet on cloud c_i. We formulate the optimal data distribution problem as an IP optimization problem, using the previous equations in such a way that storage cost is minimized subject to availability and retrieval performance constraints, as given by

$$\begin{aligned} &\min \sum_{i=1}^{N} \alpha_i \cdot \kappa_i \quad \text{such that} \\ &\qquad \alpha_i \geq \max\left(\frac{R_i}{\sum_{i=1}^{N} R_i}, \frac{k}{(N-L)}\right) \\ &\alpha_i, N, L \geq 0, \\ &\alpha_i, N, L \in \mathbb{N} \end{aligned} \tag{3.10}$$

In practice N and L are usually given values, and the values of R_i can be measured. The problem could be further specified by using a more complex model for costs or by introducing constraints to reinforce a given level of security.

In the measurements from Section 3.3, we use $\alpha_i \geq k/(N-L)$ to ensure reliability rather than the formula used to define the IP problem. Data is stored symmetrically and the motivation for an adaptive solution is given. We then use the results from this section as the basis for performing the adaptation of the data distribution in Section 3.4, providing a practical method to solving the IP problem.

3.3 Characterizing retrieval performance

Data retrieval performance is a crucial factor in determining the perceived quality of experience. While analytic results suggest that aggregating multiple services reduces retrieval time, it is hard to quantify by how much. On another note, MDS–like codes such as RLNC greatly simplify packet scheduling compared to a replication–based code. We seek to answer how much this affects average retrieval performance and whether it influences the robustness of the system. We dedicate this section to understanding the key factors that influence retrieval performance, comparing a traditional replicated and an RLNC–based erasure coded system. We analyze the real–world performance characteristics of four leading cloud storage providers through measurements carried out over the span of several months.

3.3.1 Testbed Setup

To evaluate and showcase the previously presented analytic results, we have created an implementation to conduct measurements in a real–world environment. This subsection discusses the implementation details as well as the technical specifications of our testbed. The implementation was done in C++ using KODO [Pedersen et al., 2011], a fully fledged network coding software library and the Qt framework. It was compiled with GCC 4.6.2 set to O2 level optimization. We chose four cloud storage services based on their market penetration: Box, Dropbox, Google Drive and Skydrive. All provide free storage space and have publicly available, well documented REST–based APIs.

Measurements were carried out at the Budapest University of Technology and Economics using a high–speed fiber optic connection to the Internet. This academic network setting represents an ideal scenario where the most important factor influencing retrieval performance is the bandwidth and response time of the cloud providers. Several provisional measurements were conducted using servers located in different cities in Europe (Amsterdam, Rome, Madrid, Paris, London, Oslo) to assess available network bandwidth and typical latencies. This was necessary because the location of the servers used by some of the cloud services has not been disclosed. Ping times were under 55ms and available download bandwidth over 94Mbps in all cases. A machine equipped with an AMD Athlon II X3 450 CPU at 3.2GHz, 4GB of RAM and a clean installation of Windows 7 was used. Some of the measurements were repeated on an identical machine in Berlin, using an ADSL connection. Results were mostly identical, apart from a small number of cases where the available bandwidth limited retrieval performance and have therefore not been included for brevity.

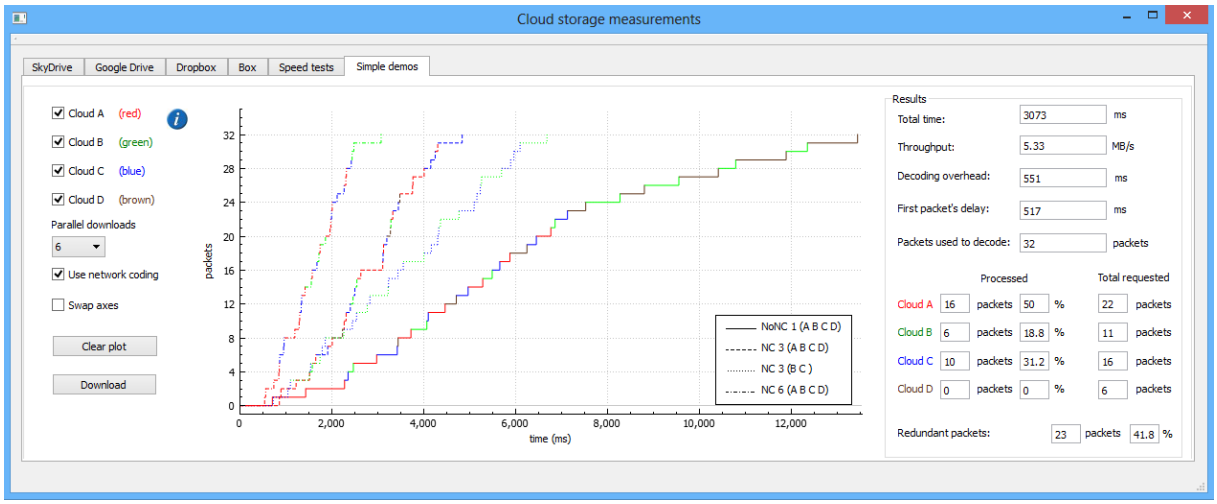


Figure 3.3: The user interface of the measurement application

We have created a demo application with a simple user interface as shown in Figure 3.3. Measurements can be performed and results compared with any combination of the four clouds, number of maximum parallel downloads and type of erasure coding (RLNC or replication–based). It measures retrieval time as well as the decoding overhead, the delay of the first packet and the necessary packets used to download the data. It shows the number of requested and processed packets for each provider and counts the number of packets that contain redundant data.

3.3.2 Measurement Campaign and Metric

We distributed and retrieved a 16MB file containing random data. A simple replication–based approach was used as a baseline to compare our proposed solution that employed RLNC. A subset of packets covering the entire data was distributed to all providers in both cases. For the RLNC approach, we used a generation size of $k = 32$ and operated over GF(2^8), resulting in packets of size 512k (and a few extra bytes overhead for the coding coefficients). Decoding was done once enough packets were downloaded. For the replicated approach, we used the same packet size of 512k. Each provider was artificially limited in the number of parallel downloads from it. This constraint was achieved by using a sliding window technique. Initially, a set number of packets was requested from each provider with a new request to that provider being issued once one was served.

We employed several metrics, the two more important ones are presented here: the time taken to retrieve a 16MB file and the number of useful packets received from each provider. The first one is the variable we chose to optimize for, whilst the second provides insight into the processes taking place. Our goal is two–fold, show that distributing data to several cloud storage providers shortens retrieval time and that employing RLNC

provides consistent results in a changing environment. Each measurement was repeated one hundred times.

3.3.3 Packet scheduling

The distribution of data constrains how it can be retrieved, and what type of scheduling techniques can be applied. We describe how scheduling packets differs for RLNC and replication and how stragglers affect performance.

Definition 3.1 (Straggler)**.** A straggler is a requested packet that takes significantly longer to download then expected.

3.3.3.1 RLNC

MDS codes and codes that have MDS–like properties, such as RLNC, do not require complex scheduling to achieve good retrieval performance. As long as a cloud provider has packets to serve, it can be used to its maximum capacity regardless of which packets are stored on it.

Because every packet has the same usefulness, optimal retrieval times can be reached by ensuring that cloud providers store enough packets. If a packet takes longer to download then normal, other packets will naturally take its place. Thus, a low number of stragglers do not affect performance significantly.

3.3.3.2 Replication

Replication–based schemes are more constrained by the data distribution in what packets can be retrieved from which storage cloud. Furthermore, stragglers are harder to deal with, making packet scheduling a significantly more complex problem. As a general rule, care must be taken to not download the same packet from different providers. However, if a packet takes significantly longer to download than is expected, then perhaps this rule should no longer be applied and the packet should be downloaded from a different provider in order to keep overall download time as low as possible. Unfortunately, requesting redundant packets to deal with these stragglers may even increase download time. Redundant requests use valuable bandwidth and space in the request queue of a provider. It is also not obvious which provider should be used to make the redundant request.

Generally, download scheduling algorithms for replicated content can be devised which perform well in more or less static environments. However, it is more difficult to create one solution which adapts well to changing conditions. We have used a simple algorithm that tries to deal with short–term changes in cloud performance by making the clouds

race against each other. The clouds start at different points in the original file. Let us order the clouds in decreasing order based on their performance: $c_1, c_2, \ldots, c_N$, where c_i precedes c_j if and only if $R_i \geq R_j$. Let us create pairs from the fastest and slowest clouds in the list, removing them in the process. If there is an odd number of clouds, the last cloud will not be paired and will receive half the work of the other pairs. Each pair will be tasked with supplying a $\frac{2k}{N}$ sized segment of the file. The last cloud or pair of clouds may need to provide slightly more packets to cover the whole file. In each pair, the faster cloud requests packets in ascending order, while the slower cloud does the same in descending order. Thus, if a cloud's rate becomes lower than long–term measurements suggest, its pair will try to compensate. Furthermore, it ensures that scheduling is optimal if each pair of clouds takes the same amount of time to download its part of the file. An example for 4 clouds is shown on Figure 3.4.

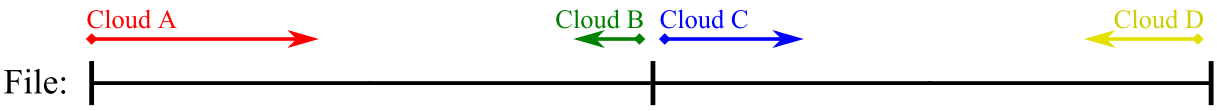


Figure 3.4: An illustration of scheduling for 4 replicated clouds.

This simple algorithm only works with a replication factor of at least 2 and if the data is distributed in the same way the scheduling necessitates. Thus, it only works well if the long–term rates the clouds provide are known and more or less static. Finally, it does not deal with stragglers as it is not clear what is the best approach, nor whether new requests lower retrieval time or actually introduce an extra delay into the system.

3.3.4 The benefits of aggregated cloud storage

We begin with the measurements of the download times for the individual clouds. We do not wish to disclose which provider is which, so we shall refer to them as Cloud A, Cloud B, Cloud C and Cloud D. Unless otherwise noted, figures show results for RLNC.

An improvement proportional to $1/l$ can be observed in Figure 3.5 as the number of parallel downloads increases. This is to be expected, assuming that the providers do not artificially throttle the requests. The curve flattens around six, so there is next to no incentive to use more than six parallel downloads per provider. Another important observation is that Cloud A is significantly faster then the other three regardless of the limitation on the number of parallel downloads. This observation shows that not all clouds have the same characteristics, contributing to the later gains of network coding which we see throughout this chapter.

We continue by showing the benefit of increasing the number of clouds in a distributed storage system in terms of retrieval time. An important aspect is that the limitation on

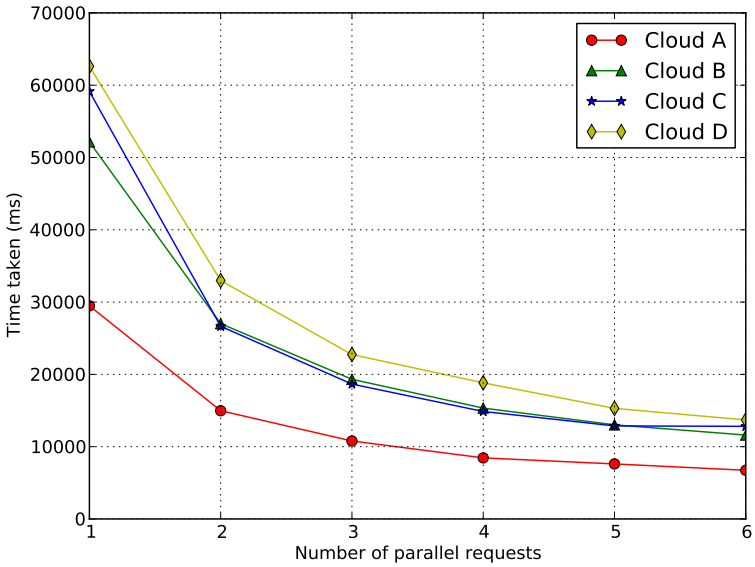


Figure 3.5: Download time for individual cloud providers

the number of parallel downloads is per provider, so using several of them means that the actual number of parallel connections in the system is multiplied by the number of providers. This is one of the reasons why employing more providers gives a clear advantage over using just one.

Figure 3.6 shows the measured gains when using more than one cloud service to retrieve files. While Cloud A is significantly faster than the other three, the limitations on available storage space would still mean in practice that a multi–cloud approach is needed

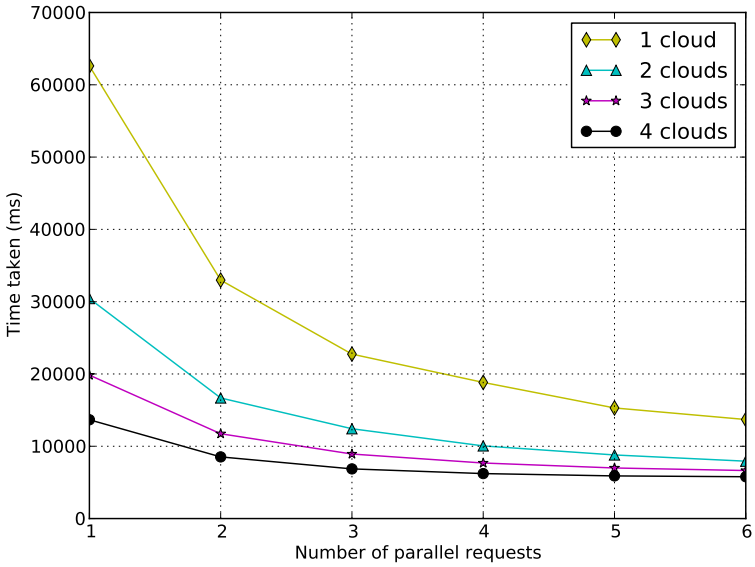


Figure 3.6: Download time when using multiple cloud providers

to achieve a certain download time given an amount of storage space and a required level of reliability. Subsection 3.2.2 gives a closed formula for calculating the optimal fraction of data to be distributed to each cloud under such heterogeneous conditions. This basically means that the faster clouds need to be able to provide proportionally more storage space. Figure 3.7 illustrates this by showing the time in milliseconds (ms) when each encoded packet was received and processed. In this example, six parallel requests per provider were allowed and decoding finished at 5660ms. Cloud A has a significantly shorter round–trip time when retrieving the data compared to the other providers and manages to download significantly more useful packets than the other providers. If less packets had been stored on Cloud A, decoding would have been delayed. Also interesting is the differing characteristics of the individual clouds. Even though Cloud B and Cloud C generally manage to download the same amount of packets, the arrival patterns shows great differences, perhaps due to the method used to process the requests by the providers.

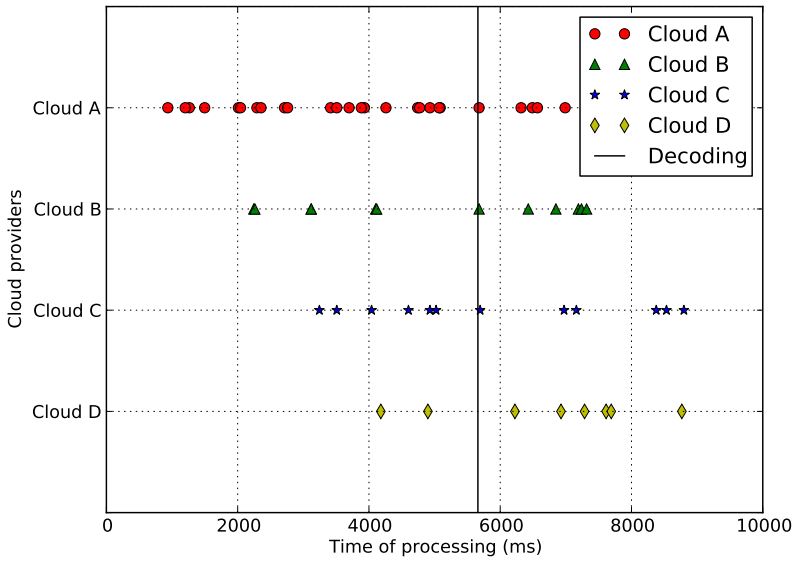


Figure 3.7: Packet processing time

3.3.5 Number of redundant packets

Figure 3.7 also illustrates the disadvantage of using a higher number of parallel downloads. At the time of decoding, there are $l-1$ packets in the download window for the provider that received the last useful packet and l for all others. These extra packets arrive after decoding is complete and therefore must be discarded. Generally, if we consider a set of N clouds with l parallel downloads per provider, the number of redundant packets is:

$$n_{\text{red}} \leq Nl - 1. \tag{3.11}$$

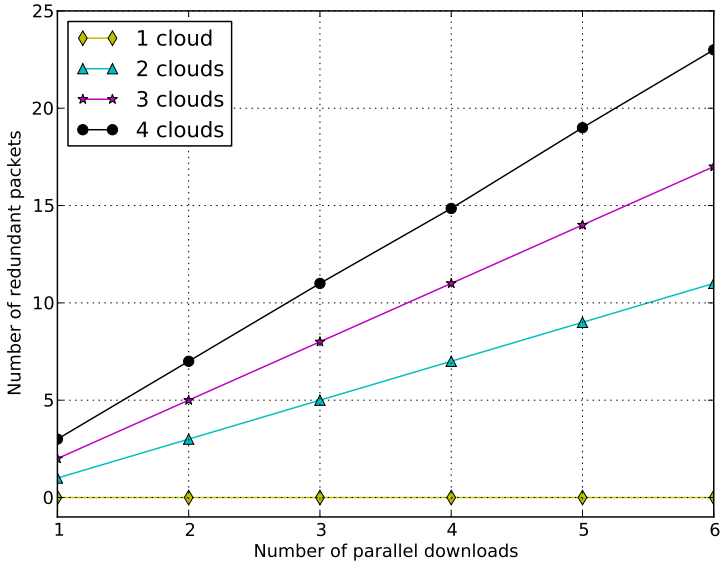


Figure 3.8: Number of redundant packets

As shown in Figure 3.8, this upper bound is in fact achieved in practice in every case. This worst–case scenario happened because there was no limitation placed on the stored data, each cloud contained packets with data covering the entire file. If such a limitation was in place, the number of redundant packets could have been less, because once all the packets from a provider had been requested, the size of the download window for that provider would have decreased.

To minimize the number of redundant packets necessitates either a smarter scheduling algorithm, which tries to predict how many packets will be received from each provider beforehand and makes the requests accordingly or a reduction in the number of parallel downloads. Considering the download times presented in Figure 3.6, a strong case can be made for using no more than two or three parallel downloads, as using more has a negligible effect on download speed whilst having a great effect on the increase of redundantly downloaded packets.

3.3.6 Comparing RLNC to replication

We examined the number of packets received from each provider as we think this should be a primary factor when determining the distribution of packets among the clouds. Providers that are able to supply packets faster should get a bigger cut to minimize the required download time, as discussed in Subsection 3.2.2. We have found that there is a significant variance in the distribution of the number of useful packets retrieved from each provider as shown in Figure 3.9. It shows results for 100 measurements using six parallel

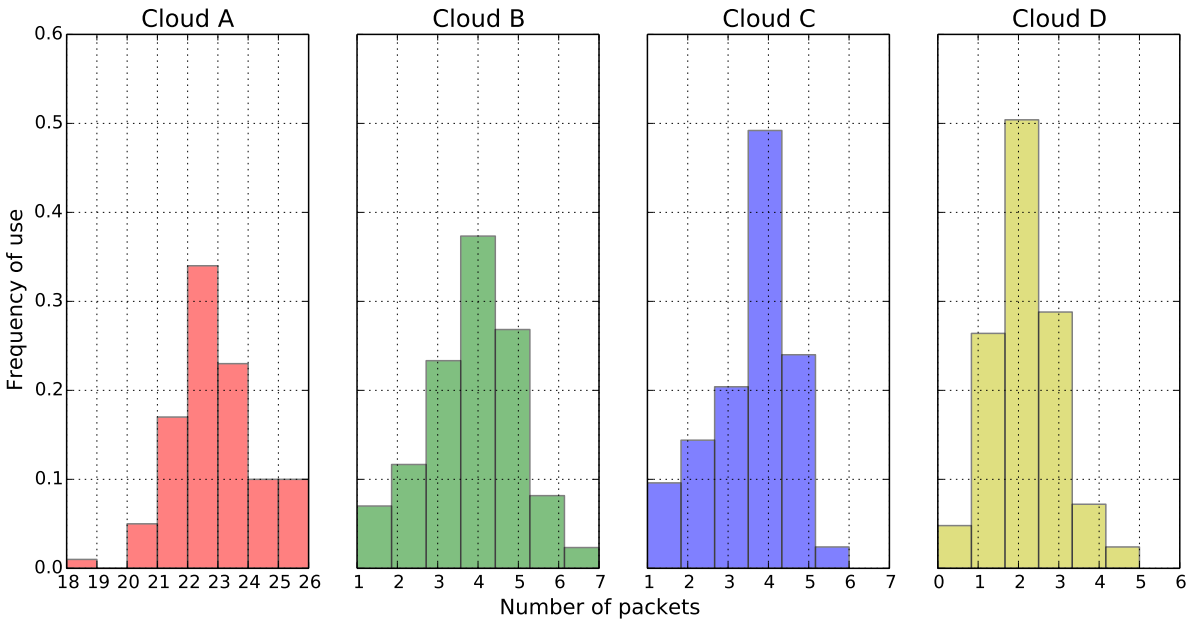


Figure 3.9: Distribution of received packets from four clouds

downloads. Cloud A retrieves most packets before the others. Even so, all four clouds still show a significant spread. This is an indication that even during error-free operation, the ratio of the download times of the providers changes, necessitating a distribution and retrieval schema that is able to adapt to the dynamically changing conditions. When considering a conventional replicated approach, it is imperative that the algorithm doing the distribution and retrieval takes into account what data is stored where. The location

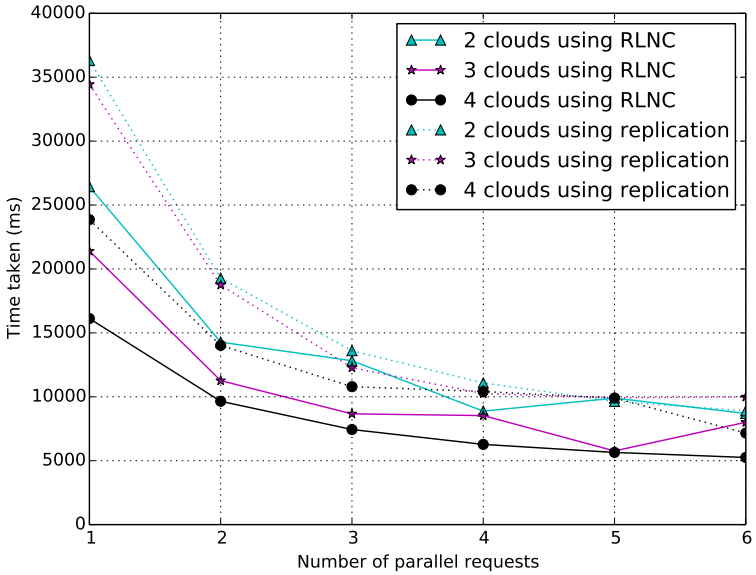
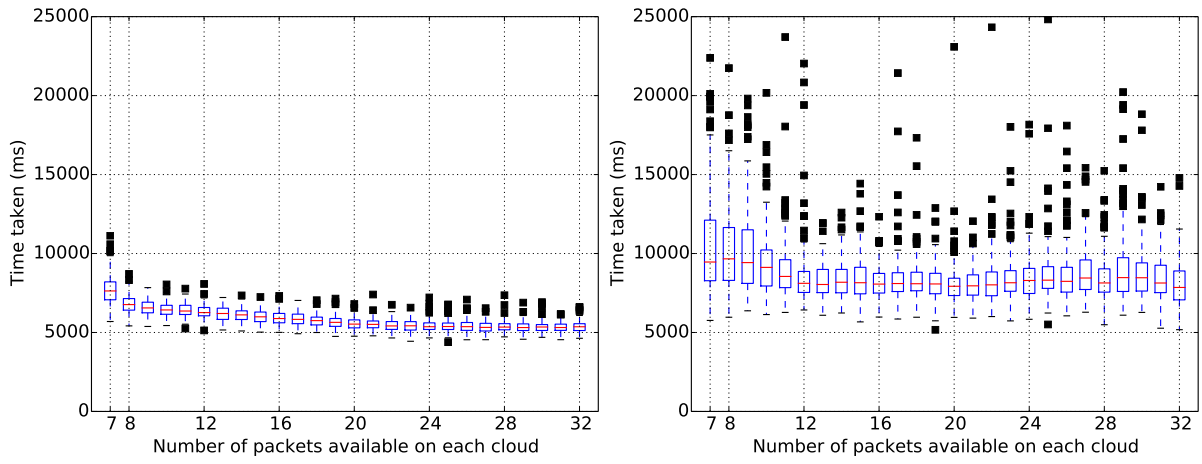


Figure 3.10: Comparing RLNC and replication on download times, using different numbers of clouds

of each packet in the original file should also be stored. It is easy to see the advantage of an approach that employs network coding, since there is no need for this bookkeeping. It is not important which packet is placed on which cloud, only the number of the packets on each cloud. This simplicity ensures that as long as the distribution of the packets is proportional to the ratio of the download speeds of the providers, the time taken to download a file will be optimal.

With a network coded approach there are no duplicate packets, each contains unique, useful data (if we assume all packets are linearly independent) and the order with which packets arrive is irrelevant. This enables it to outperform a replication in most cases as shown in Figure 3.10. When employing a single cloud, a replication is slightly faster as it does not have a decoding overhead. However, once at least two clouds are used, the network coded approach achieves shorter download times on average.

Looking closer at the distribution of download times, the reason behind the performance difference becomes obvious. Figure 3.11 shows how the two approaches compare for 6 parallel requests per provider. We introduced a further limitation in the number of packets stored on each cloud, shown on the horizontal axis. The median values are close to the means illustrated in the previous figure. RLNC outperforms replication in terms of the lowest measured values by a small margin. In terms of worst case times however, the difference is very significant, replication needs up to 3 times more time to recover the same data. The large variance shown by the replicated approach is an indication of how vulnerable it is to variances in the time taken to download individual packets. Stragglers have a very large impact on overall performance. Compared to this, the same variance has a much smaller effect on the overall performance of RLNC.



(a) RLNC (b) replication

Figure 3.11: Distribution of download times for RLNC and replication for 4 clouds

3.3.7 Coding overhead

We have shown that employing network coding can significantly reduce the time taken to retrieve the original file. This is achieved despite the overhead associated with decoding, which was around 657ms, ±5% for the given configuration. Opting for a smaller field, generation or symbol size lowers this. For example, the decoding overhead of GF(2) is less than 100ms, but the number of linearly dependent packets is higher, which in turn delays the time when decoding is finished. There is a trade-off between minimizing the decoding overhead and network usage due to additional packets being requested. This problem has been studied previously in [Heide et al., 2011b].

3.4 Adapting the data distribution

Having studied several aspects related to storing data on commercial clouds in a distributed fashion using RLNC, we now turn our attention to adapting the distribution of packets among clouds. First, we propose a method to efficiently react to short–term changes in cloud retrieval performance. Second, we address longer–term changes by providing a method to maintain data reliability at a predefined level as clouds are removed or added to the system or during temporary outages.

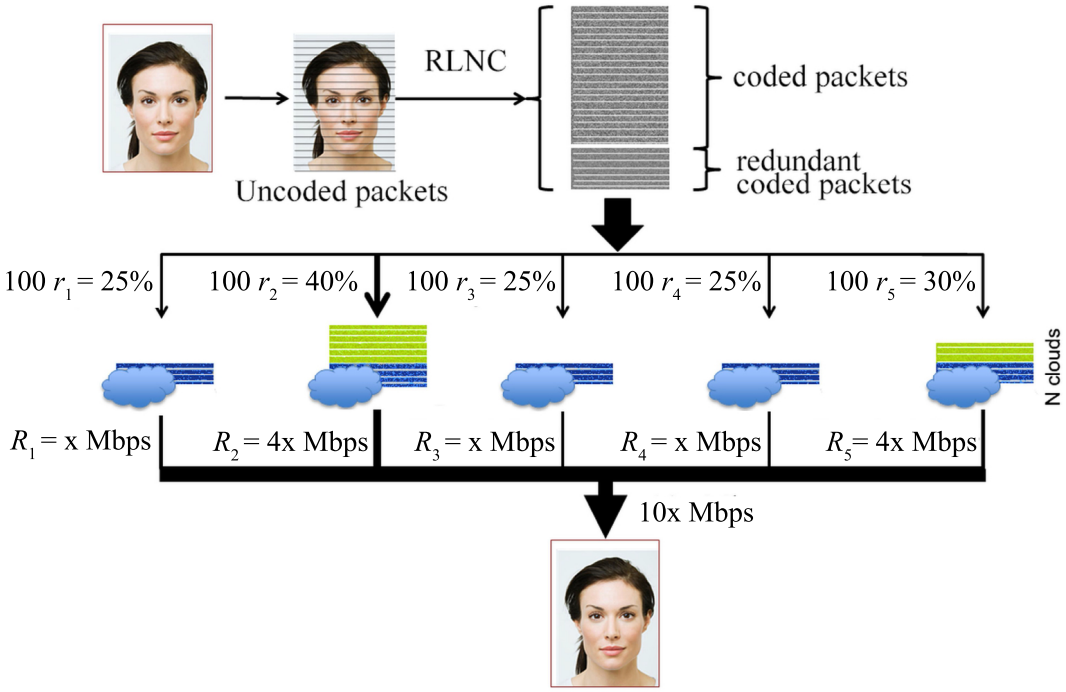


Figure 3.12: The allocation of critical (blue) and performance enhancing storage (green) depending on the different retrieval speeds.

We distinguish between two types of packets based on their creation and purpose, as shown on Figure 3.12.

Definition 3.2 (Critical data storage)**.** Critical data packets are distributed during the initial upload phase with the purpose of ensuring reliability. They are kept unchanged unless the value of N or L changes.

These are shown in dark blue on each cloud. Since they are critical in terms of data reliability checks, such as the one described in Section 2.4, can be applied to ensure that the encoding matrix associated with this data has the MDS property.

Definition 3.3 (Performance enhancing storage)**.** Performance enhancing packets are created on the fly with the purpose of minimizing retrieval times. They are added and removed continuously based on the relative performance of the clouds to each other.

The distribution of data should be continuously adapted to match the changing download rates that the clouds provide. This involves deleting data on clouds that have become slower and creating new data on ones that have become faster. Because this adaptation should be done frequently, new coded packets should be created without transferring a lot of data. However, the less data is transferred, the less information the new packets will contain. Performance enhancing packets, shown in light green on figures, must find the right balance between reducing retrieval times and the cost of the adaptation in terms of network transfer.

Despite our efforts, adaptive techniques use a significant amount of storage and upload/download bandwidth. Therefore, frequent performance optimization should only be performed for frequently accessed *hot data*. Very rarely accessed archival *cold data* should not be optimized for performance, it is sufficient to ensure reliable storage. We argue that the hotter the data, the more frequent the adaptation should be.

3.4.1 Recoding bandwidth problem

This subsection focuses on the resource allocation and adaptation process needed to optimize performance in terms of download time, while keeping the adaptation traffic low. This is in fact somewhat similar to the repair bandwidth problem [Dimakis et al., 2007], which deals with minimizing the number of packets that must be transferred to keep the reliability criterion met in case one of the storage nodes becomes unavailable. We formulate it by introducing three metrics which should be jointly minimized:

Definition 3.4 (Adaptation cost - B)**.** B is the number of packets transferred for adaptation.

Definition 3.5 (Linear dependence during data retrieval - X)**.** X is the number of extra packets needed during data retrieval to reconstruct the original file.

Definition 3.6 (Retrieval performance - T)**.** T is the time required to retrieve and reconstruct the original file.

We chose a set of coefficients for RLNC that ensured the MDS property for the critical data. However, performance enhancing packets are not held to the same standard and may be linearly dependent from other coded packets[†]. The time necessary to retrieve the original file is increased because linearly dependent packets do not contribute to the reconstruction of the original information and use valuable bandwidth. Due to the complexity of defining an optimization problem with this new structure, specifically the difficulty of expressing X as a function of the input parameters, we focused on heuristics.

3.4.2 Proposed solutions

We propose and evaluate four different methods to solve the recoding bandwidth problem. Specifically, we consider the problem of creating a single new packet at a time. To create multiple packets, the presented techniques can be applied multiple times sequentially. (I) and (II) minimize adaptation cost B, (III) minimizes both the number of linearly dependent packets X and retrieval performance T, but at great cost in B. (IV) uses a novel sparse recoding technique that offers a fair compromise between reducing the value of all three metrics. An initial idea could be to create new performance enhancing packets by simply moving a packet from a slower cloud ($P_i > r_i$) to a faster node ($P_j < r_j$). This however is unfeasible because it would break the reliability criterion as the slower node should already be on the limit in terms of the number of critical packets it stores. Our proposed methods avoid this by not changing critical storage.

Definition 3.7 (Inter–cloud copy (I))**.** Inter–cloud copy is an adaptation method where a single packet from a slower node ($P_i > r_i$) is copied to a faster node ($P_j < r_j$).

Since it does not involve removing packets, it does not break the reliability criterion. It transfers only a single packet ($B = 1$). If the old and new packets are used together for data retrieval, it will increase both X and possibly T.

Definition 3.8 (Intra–cloud copy (II))**.** Intra–cloud copy is an adaptation method that replicates an existing packet on the same node.

[†]In this case the encoding matrix used during retrieval will contain at least one submatrix of size $k \cdot k$ that does not have full rank.

This has no cost on bandwidth ($B = 0$), however it is expected to behave worse than (I) due to the redundancy that is introduced among packets which have a high probability of being used for data retrieval[‡]. Furthermore, the same effect could be achieved by using parallel requests to packets during retrieval.

Definition 3.9 (Dense recoding (III))**.** Dense recoding is an adaptation method, where a new packet is generated as a linear combination of k existing packets from other nodes.

This approach, combined with similar checks to those described in the previous chapter, eliminates linear dependence (X), thus improving retrieval performance (T). The downside is that the cost of $B = k$ is very high.

Definition 3.10 (Sparse recoding (IV))**.** Sparse recoding is an adaptation method that uses recoding similarly to (III), but creates linear combinations of much fewer packets. We are concerned with the case of combining a single packet from each of the other nodes.

We give both analytic and measurement results to show that it performs close to optimal in terms of X and T, but at the greatly reduced cost in $B = N - 1$, instead of k compared to dense recoding.

The effectiveness of (I), (II) and (IV) can be significantly improved by introducing a procedure to select which packets to use when creating the new ones. We propose basing this procedure on the individual probabilities of each packet being used during retrieval. The general idea is to combine packets that are unlikely to be used together during retrieval, therefore minimizing linear dependence and improving performance. We introduced (I) and (II) to have a baseline to compare against and do not recommend their use in actual systems.

3.4.2.1 Defining a packet request order

We propose introducing a node–based scheduling scheme into the content retrieval phase by setting up a preferred order of performing the packet requests. This allows estimating the likelihoods of a packet being used for retrieval. Let $pac(i, j)$ denote packet number j stored on node c_i.

Definition 3.11 (Packet request order)**.** The order in which packets are requested is defined by relationship $\preceq$, such that $pac(i, l) \preceq pac(i, m)$ if and only if $l < m$.

Packet $pac(i, l)$ is requested no later during the content retrieval phase than $pac(i, m)$ if and only if its arbitrarily assigned number is smaller. Note that this ordering is valid for a single node and there is no defined order between packets from separate nodes as these behave as separate entities.

[‡]This is a consequence of the fact that the probabilities associated with packets on faster nodes of being used for data retrieval are greater.

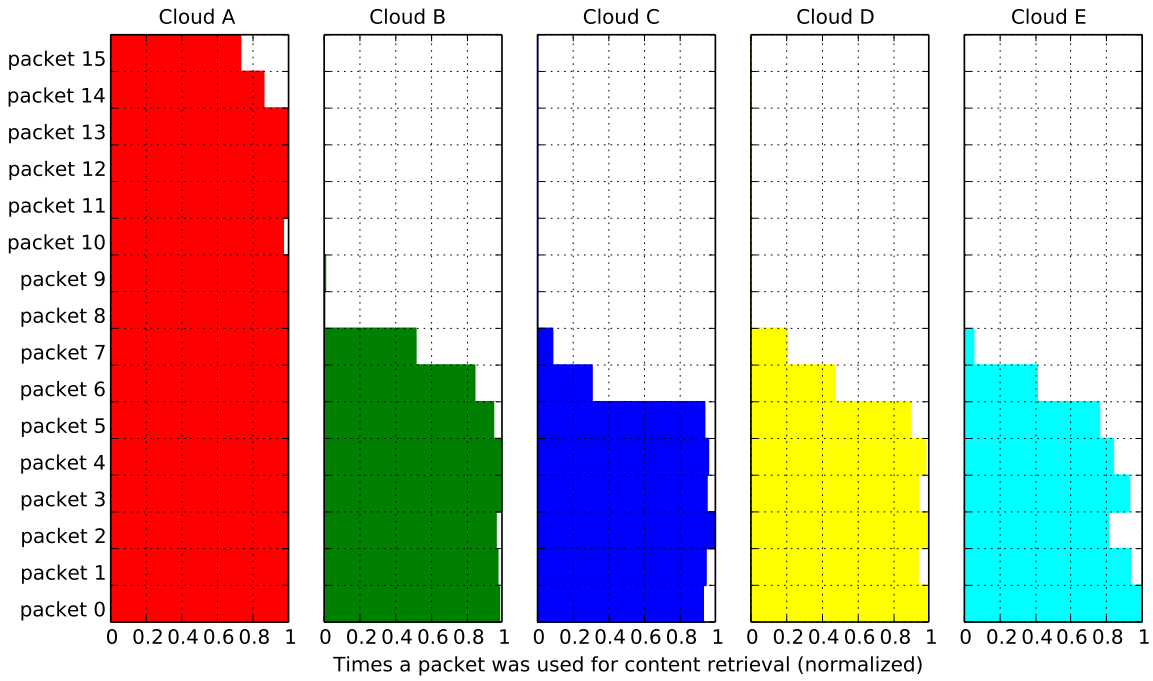


Figure 3.13: Distribution of packets received from each provider over a period of 24 hours using an adaptive approach

Let $X_{pac(i,j)}$ be the indicator variables with Bernoulli distributions signifying whether a packet is used during content retrieval:

$$\text{Let } \begin{cases} X_{pac(i,j)} = 1 & \text{if the packet is used for data retrieval} \\ X_{pac(i,j)} = 0 & \text{if the packet is not used} \end{cases} \tag{3.12}$$

The probability of being used for data retrieval will then be a non-increasing function of j. Thus,

$$P(X_{pac(i,0)} = 1) \geq P(X_{pac(i,1)} = 1) \geq \cdots \geq P(X_{pac(i,\alpha_i - 1)} = 1). \tag{3.13}$$

Figure 3.13 shows the measured likelihood for all packets stored in the system.

3.4.2.2 Defining a recode order

To decrease linear dependence during data retrieval (X), performance data should be generated by using that part of the critical data, which has a low probability of being used in conjunction with the newly generated data later, during retrieval. In general, the critical packet with the lowest probability should be used first, followed by the one with the second lowest and so on. When selecting packets for solutions (I), (II) and (IV), the recode order should therefore be the exact opposite of the request order with

one important additional criterion. Namely, we only define the order for packets from the critical data. These are the packets that should be used for solutions (I), (II) and (IV), because they are linear combinations of all uncoded packets and therefore contain the maximum amount of information. In contrast to this, performance enhancing packets contain less information due to their sparse nature and therefore have a higher probability of introducing linear dependence. By using exclusively critical packets as a source for the adaptation, no more performance enhancing packets can be created on any cloud than the number of critical packets stored on each cloud using solutions (I), (II) and (IV). This limits performance in highly unbalanced systems. A possible solution is to switch to dense recoding once this limit is reached.

Definition 3.12 (Recode order)**.** The order in which packets are recombined during recoding is defined by relationship $\dashv$, such that

$$pac(i,l) \dashv pac(i,m) \text{ if and only if } l > m, \text{where } 0 \leq l, m < \left\lceil \frac{k}{N-L} \right\rceil.$$

For inter–cloud copy we propose selecting a single packet from one of the other nodes based on the recode order. It is not clear which node to use, therefore we propose selecting one at random from the set that does not include the node that will contain the newly generated packet. Intra–cloud copy follows the recode ordering as is. For dense recoding,

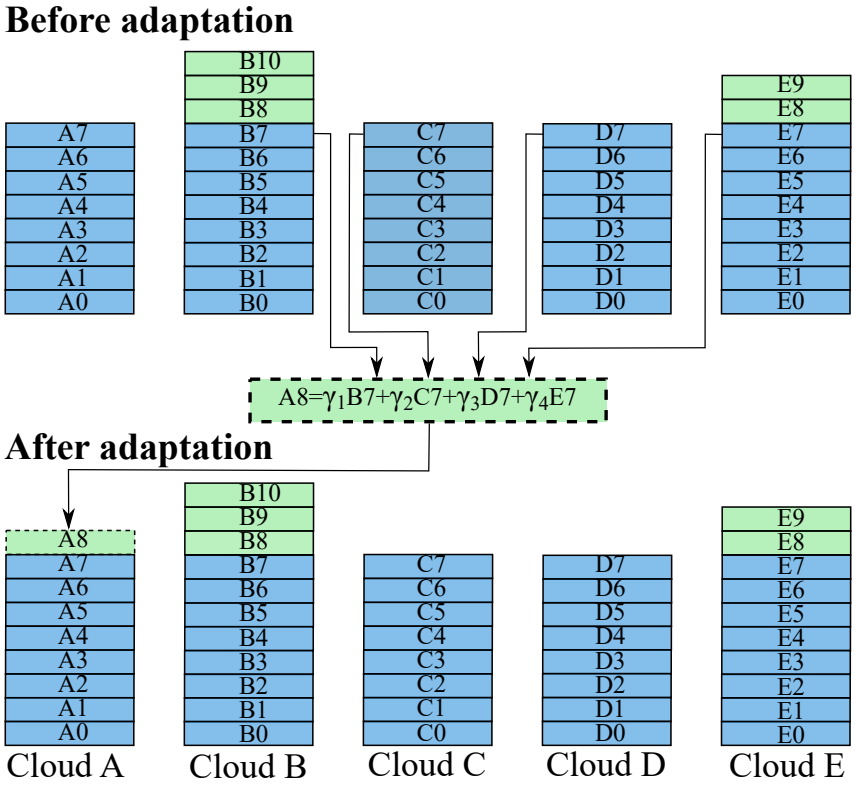


Figure 3.14: An example of a new packet being generated with sparse recoding.

there is no need to choose packets, because it does not matter which k are used due to the MDS property.

On the other hand, the effectiveness of sparse recoding benefits greatly if the packets are selected based on the recoding order. As illustrated in the example on Figure 3.14, for creating performance enhancing packet number r on a cloud, we propose gathering only one critical packet from all other clouds, specifically number r counting from the end of the critical packets. In the example, the retrieval performance of Cloud A has just increased, thus a new performance enhancing packet should be added to it. Our proposed sparse recoding solution generates the new packet A8 by creating a linear combination of packets B7, C7, D7 and E7 using random coefficients(γ_i). This is then uploaded to Cloud A. Similarly, if Cloud A becomes even faster in the future, packet A9 would be generated by combining packets B6, C6, D6 and E6. Our sparse recoding schema requires only $N - 1 = 4$ packets for each new performance packet as opposed to the number necessary to decode the original content, 32, for the most dense recoding.

3.4.3 Experimental results

We have implemented all four methods and performed measurements to determine the effectiveness of each techniques. We added SugarSync to the list of cloud storage providers, increasing the count to five: Box, Dropbox, Google Drive, OneDrive and SugarSync. The plots show the mean values for download time for a 16MB file, the mean cost of adaptation in packets in a measurement round, and the percentage of linearly dependent packets during retrieval based on 1000 measurements for each parameter set. We have introduced a storage constraint (an upper bound on α_i) to evaluate how each solution reacts to a broader range of scenarios. On figures, the numbers next to the markers indicate this

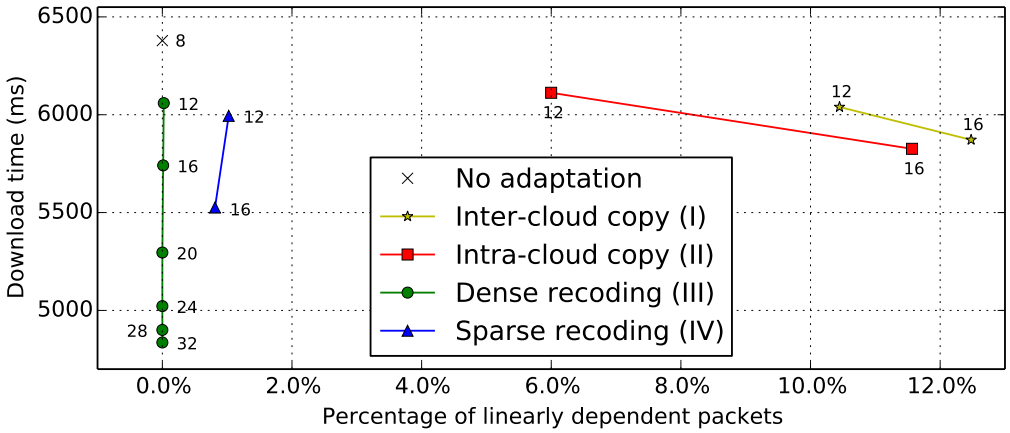


Figure 3.15: Comparing retrieval performance in relation to percentage of linearly dependent packets.

constraint. Solutions (I), (II) and (IV) are constrained to a maximum of 16 as described in Subsection 3.4.2.2.

Figure 3.15 shows a correlation between download performance and linear dependence in the case of solutions affected by the latter – (I), (II), and (IV). (I) and (II) retrieve a high number of linearly dependent packets, which decreases their performance. Figure 3.16 compares the trade–off between download performance and the cost of adaptation. Inter–cloud and intra–cloud copy use very little adaptation bandwidth, but have a higher download time . All solutions outperform the non–adaptive approach. Dense recoding shows the best performance overall, but needs a large number of packets for the adaptation. Sparse recoding uses significantly fewer packets and still achieves slightly better performance for the scenarios with storage constraints 12 and 16. This is due in part to the lower decoding complexity of a sparse code. Interestingly, there is more adaptation taking place in scenarios with a storage constraint below 20. This is more pronounced for dense recoding due to its very high adaptation cost per packet: 9x more than for sparse recoding, 32x more than for inter–cloud copy. The effect can be explained by our unbalanced measurement setup, where one of the clouds is significantly faster than the others. In scenarios with a more restrictive ($\alpha_i = 12$ and $\alpha_i = 16$) storage constraint, the fluctuation of the dynamics of the cloud providers has a more pronounced effect. It is not so noticeable for less restrictive scenarios ($20 \leq \alpha_i \leq 32$) where the fast cloud dominates and the performance of the slower ones no longer dictates generating performance enhancing packets for them. Most of the adaptation cost associated with these scenarios comes from filling up the fast cloud with performance enhancing packets in the initial phases of adaptation.

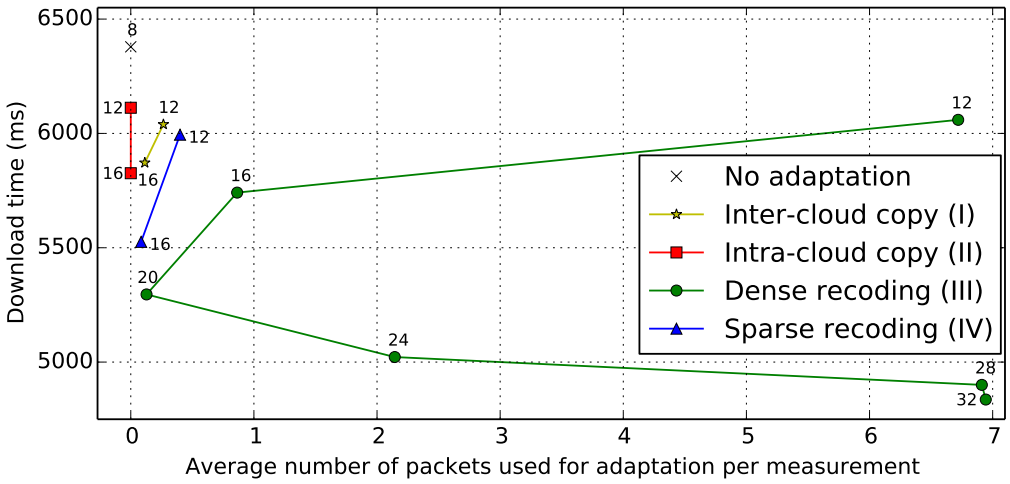


Figure 3.16: Comparing retrieval performance in relation to adaptation cost.

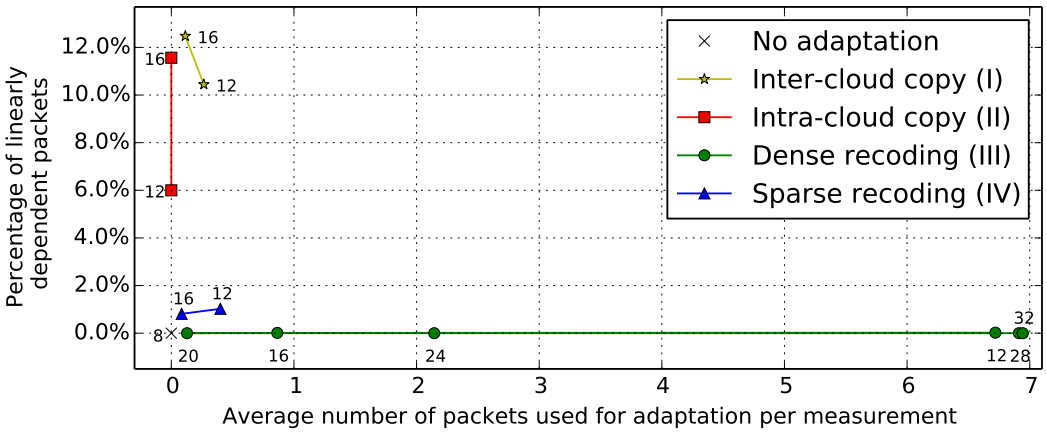


Figure 3.17: Comparing adaptation cost in relation to the percentage of linearly dependent packets.

Finally, Figure 3.17 illustrates that sparse recoding generates significantly less linearly dependent packets compared to (I) and (II) and comes very close to dense recoding. This is due in part to the recode ordering.

We include results which illustrate the robustness of the system in handling dynamic conditions when faced with large–scale changes in individual cloud performance. Figure 3.18 presents a period of 24 hours using the unconstrained dense recoding technique. The upper subplot shows retrieval times, the lower the amount of packets downloaded from

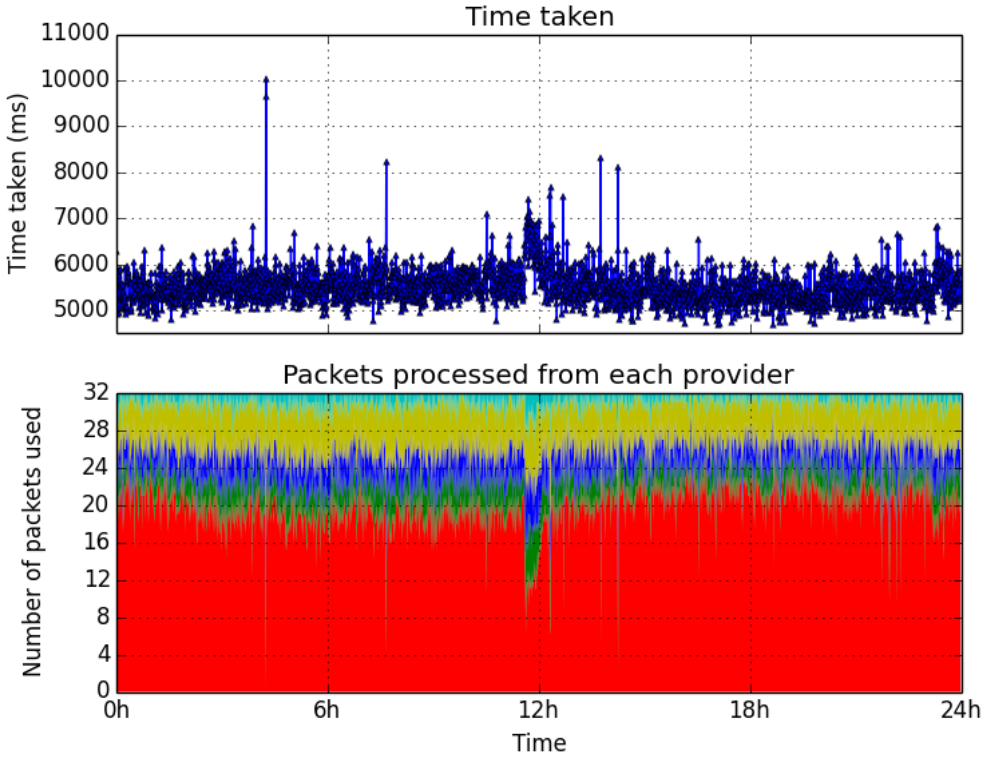


Figure 3.18: Performance and distribution of retrieved packets over 24 hours.

each provider. The different colors mark the different providers with the bottom one being noticeably faster. A significant event occurred around the 12 hour mark to which the system adapted by removing performance data from the previously fast cloud and adding some to the second fastest. Over time, the distribution reverted to the pre–event state.

3.4.4 Handling changes in the number of cloud providers

Cloud computing providers usually guarantee a high level of availability in their SLAs. Despite this, outages are a relatively common occurrence. Therefore, it is advisable to prepare for such events by storing data redundantly. Although $L = 1$ ensures reliable operation during an outage, a second outage would compromise data availability. To prevent this from happening, the number of critical data packets needs to be adapted to the decreased number of clouds. Furthermore, reliability may also be compromised in more common circumstances: subscription ending, billing issues or simply deciding to discontinue the use of a service.

To adapt the level of redundancy in the system to the decreased number of storage clouds, new coded packets need to be generated and added to the critical storage. Using

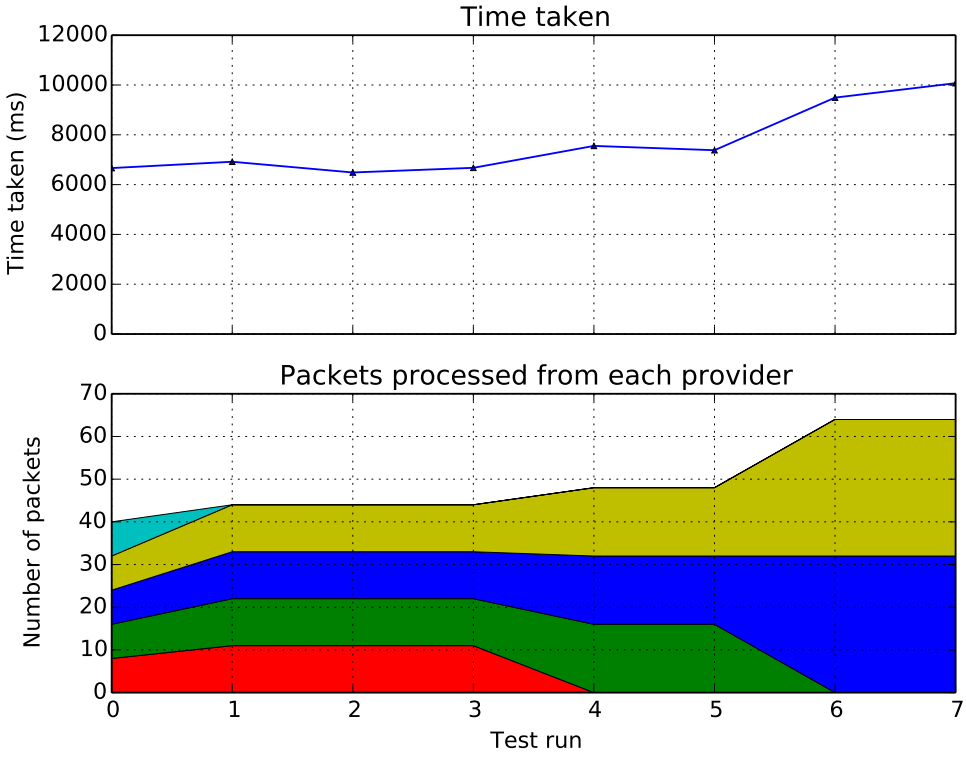


Figure 3.19: Maintaining a reliability of $L = 1$, while the number of cloud providers is reduced.

RLNC, this can be achieved with recoding. It is not necessary to first recover the original data as recoded packets can be generated from any number of existing coded packets. However, to maximize the probability that they carry useful information, k should be used. Furthermore, because this modifies the critical data storage, additional checks need to be performed to ensure recoverability. Figure 3.19 shows how the system adapts the distribution of data as it goes from 5 clouds to 2, such that $\alpha_i \geq \lceil k/(N-L) \rceil$ is maintained after each change in N. The total amount of stored data increases as shown in the lower subplot while performance decreases, as shown in the upper subplot.

A new cloud joining the system can be handled in a simpler fashion: part of the critical data storage from the old providers can be copied to the new one. There are always enough packets to redistribute, because the more clouds the systems contains, the less redundant data it needs to store to be able to handle the loss of one cloud. As with the other case, additional checks need to be performed for the new decoding matrices. An example is shown on Figure 3.20.

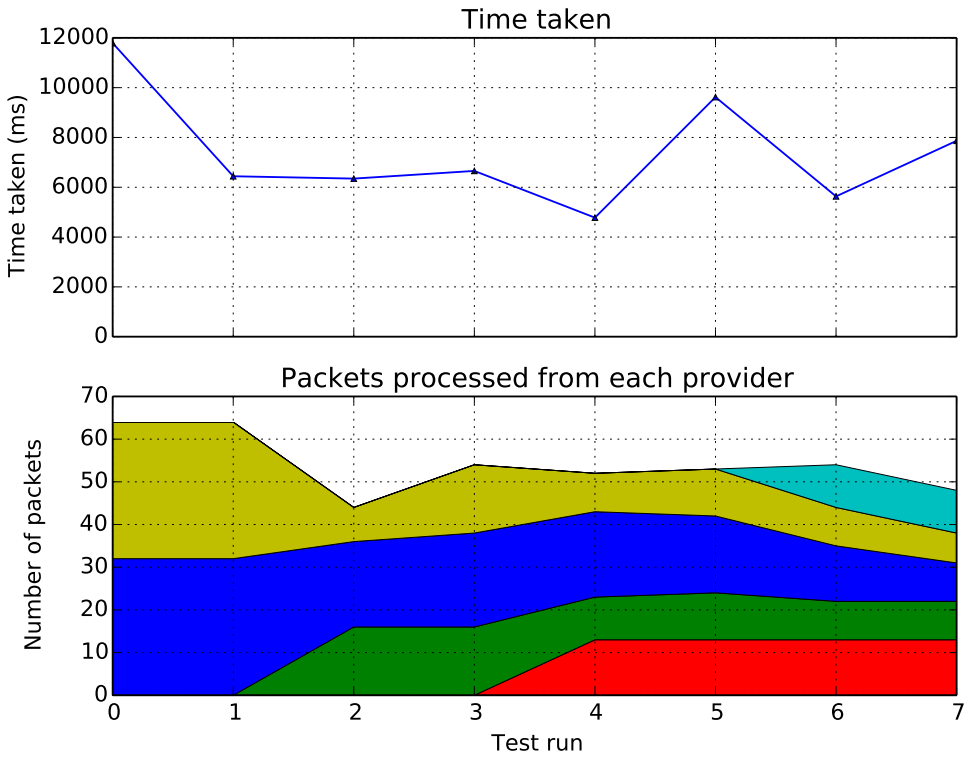


Figure 3.20: Maintaining a reliability of $L = 1$, while the number of cloud providers is increased.

3.5 Conclusion

Using multiple cloud providers has many benefits in terms of reliability, retrieval performance, as well as privacy and security. We characterized the relationships between these and showed using measurements that by applying RLNC on top of the cloud storage results in predictable performance that is superior to that of a non–encoded, replicated solution. Most of the gains are a result of the MDS–like property of RLNC, which simplifies and enhances data distribution and retrieval. This is also desirable from a system design perspective, as some bookkeeping relating to the data distribution can be avoided. If encoding parameters are chosen judiciously, the extra computational time of using erasure coding is compensated by decreased retrieval times. This holds true even for relatively costly dense versions of RLNC.

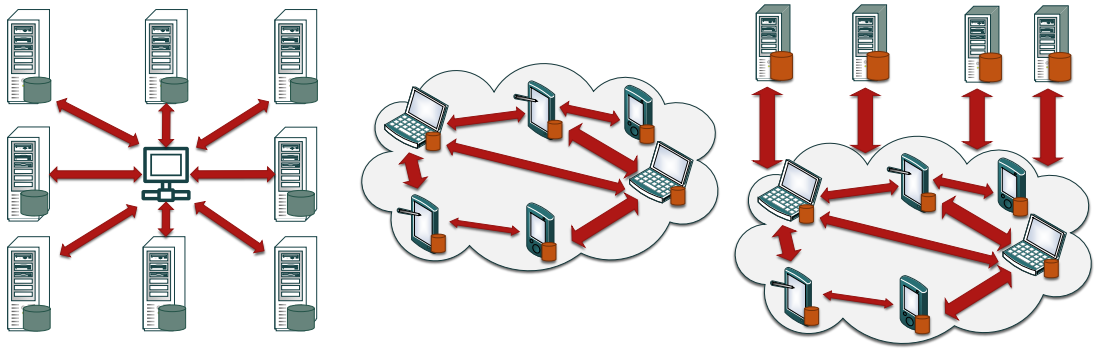
This chapter also covered adapting the data distribution scheme to reflect changes in cloud performance in order to decrease retrieval times. We have proposed a sparse recoding technique that has low adaptation cost while maintaining close to optimal performance. Finally, we showed that the rateless nature of RLNC makes maintaining a level of redundancy in the face of a changing number of clouds simple. While many of the benefits of using RLNC over cloud storage generalize to all MDS codes, its rateless nature and efficient recoding set it apart by enabling a distributed storage system to adapt to changing conditions. Furthermore, its random selection of coefficients carries advantages in terms of privacy and security.

Chapter 4

Erasure coding for fog computing

4.1 Introduction

Almost all current online services are centralized and are supported by a distributed storage systems that features a central storage controller that manages the distribution, retrieval and maintenance of data. However, many envision a move towards more decentralized systems. Whether in their pure form, as a fully decentralized P2P system shown on Figure 4.1b, or as an edge caching mechanism for cloud–based services shown on Figure 4.1c, mobile storage clouds move the data closer to where it is produced and consumed. Because they are based on a P2P topology, they scale better, make better use of existing infrastructure, reduce latency, improve throughput and remove the single point of failure characteristic of a centralized topology. As such, we see them as one of the key enablers of fog computing architectures.



(a) Centrally controlled storage

(b) Decentralized mobile cloud

(c) Decentralized mobile cloud augmented with high–availability nodes

Figure 4.1: Distributed storage architectures

Offsetting the benefits somewhat, mobile storage clouds present added challenges as the number of nodes changes continuously and is hard to predict. Erasure codes can play a crucial role, but must meet stringent requirements, regenerating data onto new nodes without a central arbitrator and providing slack to account for the large variance in node numbers. Furthermore, such systems can only sustain themselves if they possess adequate bandwidth to fill new nodes.

4.1.1 Structure and overview of the contributions of this chapter

This chapter has three main contributions. First, it introduces and contrasts different types of data regeneration techniques for network coded distributed storage. This is compared to replication and Reed–Solomon–based techniques through a large number of experiments that focus on how well they maintain the integrity of the mobile clouds. A distinction is made between centrally controlled and fully decentralized approaches. Besides looking at how well single node failures can be recovered from, our results deal with data regeneration in two types of concurrent node failure patterns.

Second, a sufficient set of conditions is derived for quasi–infinite longevity of data when using network coding using realistic constraints.

Third, we explore user behavior in a mobile P2P BitTorrent system to analyze how much aggregated bandwidth and storage may be offered by a mobile cloud. We use these results to make rough calculations on whether such a system can be self–sustainable. We show that mobile storage clouds are feasible as long as they meet certain requirements and briefly look at how high–availability nodes improve the system.

Finally, we propose a two–phase technique to estimate the number of nodes in the system based on a simple idea: by aggregating device level estimations centrally, local knowledge can be used to improve accuracy.

Section 4.2 introduces three different approaches to regenerating data in a distributed storage system. Section 4.3 looks at defining a set of sufficient bounds to ensure data survival for functional repair. Section 4.4 compares our proposed approaches using different node failure patterns. Section 4.5 looks at whether mobile storage clouds are feasible and Section 4.6 introduces a simple technique to predict node availability. Finally, Section 4.7 presents the conclusions and lays out potential directions for future work.

4.2 Reconstruction strategies

We use the same general notation and model as in the previous chapters, with some changes. As before, our proposed system is made up of N initial nodes, each storing the same α amount of data. When a data creator uploads a file to the system, it is first

divided into k pieces. These are encoded to create n coded pieces, which are then evenly distributed to the nodes. When a data collector tries to retrieve the original file, it contacts a number of nodes and gathers at least k coded pieces to be able to decode. Unlike the system described in the previous chapter, nodes in decentralized systems are not simply storage locations, but intelligent entities, which can perform computations on the data. It is distinguished from the last two chapters in that there is no coordinating entity to oversee the operations of the system. Thus, the reconstruction strategies described in this section differ from the repair mechanism defined in Chapter 2 in that nodes must function autonomously, possessing a limited view of the network.

As in Chapter 2, the passage of time is simulated using a discrete sequence of rounds (generations). During each generation, the number of nodes may vary as they join and leave the system. When a node joins the system, it is filled with new pieces using encoded pieces from the remaining nodes. We refer to the subset of remaining nodes that are used in this reconstruction process as parents and use d to denote their number. Data stored on nodes that leave the system is considered lost in Section 4.2, 4.3 and 4.4. Rejoining nodes appear in the model in Section 4.5. Unlike the previous two chapters, the number of available nodes in a given generation may surpass their initial number as we look at different node patterns.

This section introduces the different strategies that deal with how L nodes are filled by their d parents. The strategies will differ in the success probability p_S^g and the related reconstruction traffic.

Definition 4.1 (Success probability(p_S^g))**.** After filling up the new L nodes, data integrity is checked over all N nodes. If successful, the procedure starts from the beginning until g generations are completed. The tests are repeated several times to derive the success probability, which is defined by the ratio of successful tests to the overall number of tests carried out.

Definition 4.2 (Reconstruction traffic(γ_S))**.** γ_S is the number of packets transmitted for a specific strategy S to fill a newly joined node.

Two constraints have been considered. First, the amount of data stored on each node is limited. Second, there is a limit on the number of parents that can be contacted by a joining node. This is a common bottleneck in current storage systems and an active research area [Dimakis et al., 2011, Hu et al., 2012b, Lei et al., 2013]. Ultimately, our goal is to minimize the amount of storage over all nodes and network traffic while being still able to retrieve the original data.

4.2.1 RLNC–based recovery strategies

4.2.1.1 Post–recoding approach

The d parent nodes convey all available $d\alpha$ information to the new L nodes. Each node out of L recodes over all $d\alpha$ received packets storing only α and discarding the remaining $(d-1)\alpha$. Due to the large number of received packets and the possibility to recode, each of the L nodes will store different coded versions and the likelihood to store redundant information across the new nodes is reduced with an increased field size. The resulting traffic γ_{post} per generation equals

$$\gamma_{\text{post}} = d\alpha L. \tag{4.1}$$

4.2.1.2 Pre–recoding approach

In order to reduce the traffic involved in filling the new nodes, pre–recoding is introduced. Each of the d parent nodes recodes over its own α packets sending only $\lceil \alpha/d \rceil$ to each new node. For each new node, different coded versions are generated. The receiving node stores the received coded versions without any further actions. The overall traffic γ_{pre} is drastically reduced in comparison with the previous approach, as given by

$$\gamma_{\text{pre}} = \left\lceil \frac{\alpha}{d} \right\rceil dL = \alpha L + \theta dL, \tag{4.2}$$

where $\theta \in [0, 1)$. While the traffic is reduced significantly by a factor of up to d, the diversity in the recoded packets is small. Therefore, we propose a hybrid approach that combines the benefits of the former two.

4.2.1.3 Hybrid approach

While the post–recoding approach sends the maximum traffic and achieves the maximal mixing of the coded packets, pre–recoding sends the minimum traffic with limited mixing capabilities across different nodes, i.e., only recoding within each existing node. As the name implies, the hybrid solution provides a simple mechanism to trade–off traffic and coding diversity.

In the hybrid approach, each parent recodes over its own packets sending $\lceil \lambda\alpha/d \rceil$ to each of the L nodes. The values for λ are between 1 and d, where $\lambda = 1$ corresponds to the pre–recoding and $\lambda = d$ to the post–recoding approach. Each new node receives $\lceil \lambda\alpha/d \rceil d$, over which it recodes again in order to store α packets, discarding the unused

packets. The overall traffic γ_{hybrid} for the pre–recoding approach is given by

$$\gamma_{\text{hybrid}} = \left\lceil \frac{\lambda\alpha}{d} \right\rceil dL = \lambda\alpha L + \theta L d, \tag{4.3}$$

where $\theta \in [0, 1)$.

4.2.2 Reed–Solomon coding approach

In order to compare our network coding approaches with the state of the art, we use Reed–Solomon (RS) coding. There are two ways of using Reed–Solomon in this context and the first one results in more traffic but makes sure the integrity of the information is intact, while the second is prone to information loss but uses less traffic and is comparable in its constraints to the proposed network coding approaches.

4.2.2.1 Fully controlled Reed-Solomon

Each of the L nodes has to retrieve any k segments from the remaining P parents, decode the received segments if possible, and store the exact same information that was lost beforehand. This approach needs some overlay control entity to organize that the L nodes are filled in the correct way. The resulting traffic $\gamma_{\text{RS,control}}$ in this case is

$$\gamma_{\text{RS,control}} = kL, \tag{4.4}$$

without taking into account for the extra signaling that would be needed among the overlay control entity and all N nodes.

An optimization in the amount of traffic generated is to retrieve all k packets in one of the L nodes, create the missing $L\alpha$ packets, then distribute the missing ones to the other $L - 1$ nodes. In this case the traffic is reduced to

$$\gamma_{\text{RS,control}} = k + \alpha(L - 1). \tag{4.5}$$

If a newly joined node is unable to decode, it will replicate the rarest packet in the system from those it receives.

4.2.2.2 Random Reed–Solomon

In contrast to the controlled approach, we can forward randomly selected pieces to the new nodes. Following the pre- and the post–recoding approach for network coding, Reed–Solomon can also perform a traffic aggressive or a traffic careful policy, sending either $d\alpha$ or only α segments to each of the L new nodes, respectively. The resulting traffic $\gamma_{\text{RS,random}}$

here is therefore similar to the traffic of the post and pre–recoding approach of network coding given in Section 4.2.1.1 and 4.2.1.2, respectively, that is

$$\gamma_{\text{RS,random}} = \lambda d \alpha L, \tag{4.6}$$

with λ being either 1 (large traffic) or $1/P$ (low traffic).

4.2.3 Replicated approach

Since replication is still the most common form of storing data in a distributed storage system, we included this approach as a comparison.

4.2.3.1 Centrally controlled replication

In centralized systems, a control entity can keep track of what is lost and replicate those packets. Newly joined nodes replicate the rarest pieces in the system from those served by their parents. This is similar to BitTorrent [Legout et al., 2006] and has the purpose of avoiding the loss of any given packet. Let $\gamma_{\text{repl,control}}$ be as defined in Equation (4.6).

4.2.3.2 Random replication

This uncoded approach also uses random filling as described for Reed–Solomon. Care is taken to not replicate the same packet on a newly joined node twice. The approach is used for comparison and not as a recommendation for use in a storage system. Let $\gamma_{\text{repl,random}}$ be as defined in Equation (4.6).

4.3 Ensuring data availability using RLNC

This section looks at determining lower bounds on the number of parents that are needed to maintain cloud integrity for network coded approaches and $L = 1$. We limit our evaluation to hybrid recoding since it is a generalization of both pre–recoding and post–recoding. To have results that are closer to real–world systems and be more in line with other efforts, we use β to denote the number of packets a parent sends instead of fine–tuning the λ parameter. Since we advocate the use of Random Linear Network Coding, we introduce the simplification of using an infinitely large finite field. Thus, this evaluation focuses on avoiding linear dependency introduced by the reconstruction process, rather than by the unfortunate selection of coding and recoding coefficients. We later show using experimental results that even moderately sized finite fields, such as $GF(2^8)$, are suitable for practical systems. Unlike the results in Chapter 2, we look at the case of homogeneous

β, that is $\beta_i = \beta$, as our model does not allow any coordination between the nodes. This constraint, among other practical considerations, also makes it unfeasible to apply the checks to the coding and recoding coefficients described in the same chapter.

4.3.1 System convergence

We wish to show that a storage system using RLNC can ensure data survival with a high probability after a large number of failures given a judicious selection of values for some key parameters. Using the model based on information flow graphs, the criterion for this is to have a maximum flow with value k (or k edge–disjoint paths) between the source and the sink. In other words, the data collector must be able to gather as many linearly independent packets from the surviving storage nodes as the number of packets that the source introduced into the network (k). First, the initial data distribution must be performed such that this property is satisfied. Second, subsequent node loss and recovery processes must ensure that the property is kept for a large number of generations. We denote $M^g \subseteq \Omega_N^g$ as a randomly selected set of non–failing nodes in generation g. It is the smallest possible set that stores k packets (to make possible a flow with value k between the source and the sink without the traversal of other nodes of generation g).

$$m = |M^g| = \left\lceil \frac{k}{\alpha} \right\rceil \tag{4.7}$$

Definition 4.3 (Robust Data Recoverability (RDR))**.** A system has the RDR property if and only if data can be recovered from any set M^g of non–failing nodes in generation g, where $|M^g| = \lceil \frac{k}{\alpha} \rceil$.

For RLNC this is analogous to having a submatrix of rank k of the matrix composed of the coefficients used to (re)encode the data for all selections of nodes of size m in each generation. The RDR property is a generalization of the Maximum Distance Separable (MDS) property. We introduced RDR to handle cases where $\alpha \nmid k$, common for RLNC.

To have this property, the initial distribution of data must meet some conditions. More importantly, the storage system must have a robust reconstruction transition between generations, as there is no way to rebuild lost data paths. Newly introduced reconstruction edges to the recovery device have the goal of increasing the interconnectedness of nodes. They build redundant paths to be used in case another node fails in the future.

The following proposition states that any given non–adaptive reconstruction mechanism either maintains the RDR property indefinitely or loses it after some generations and never recovers it.

Proposition 4.4. *Considering a storage network with a fixed set of values for k, N, α and a reconstruction transition with fixed values for β and d that maintains the RDR property*

between at least one pair of consequent generations l and $l+1$, then this transition will also maintain the RDR property for any generation $g \in \mathbb{N}^+$.

Proof. We divide the proof into several parts:

- For $g < l$: Based on the network construction that provides a topologically sorted form, if layer l exhibits the RDR property then all layers $g < l$ must also have this property as the premise of the proposition can be applied recursively until generation 1.

- For $g > l+1$: Let us assume that generation g is the first generation after $l+1$, for which there exists a selection of nodes M^g that do not have k edge–disjoint paths pass through them. We will show by contradiction that such a selection cannot be made and thus generation g also has the RDR property.

  M^g can be selected in $\binom{N}{m}$ ways. Fortunately, it is enough to consider two distinct cases.

  - M^g does not include the newly recovered node.

    In this case, all nodes in M^g were already present in the previous generation, therefore we can easily find the corresponding set of nodes M^{g-1} that includes the same storage nodes. These can support k edge–disjoint paths because generation g is the first to not have the RDR property. The α paths between the m pairs of nodes in generation M^{g-1} and M^g will ensure k edge–disjoint paths pass through M^g. We have arrived to a contradiction.

  - M^g includes the newly recovered node.

    Surviving nodes already ensure at least $k-\alpha$ edge–disjoint paths pass through nodes in the corresponding M^{g-1} set in generation $g-1$. Let us assume that the recovered node does not provide the minimal number of α additional paths necessary for the system to keep the RDR property in generation g. This implies that there are less than α edges between the recovery node and nodes from generation $g-1$ outside of M^{g-1}, i.e. $(p-(m-1))\beta < \alpha$. The original assumption of the proposition is that d, k, α, β and thus m have a fixed value. Therefore, this bound must also have had to have been in place in generations l and $l+1$. This would mean that the transition to generation $l+1$ would have lost the system the RDR property because the M^{l+1} set that included the node recovered in generation $l+1$ would not have had k edge–disjoint paths pass through it either. Again, we have arrived to a contradiction.

Because we arrived at contradictions for both categories of cases, we can conclude that the system must be able to support robust data recoverability in generations following generation $l + 1$ as well.

□

An important consequence of Proposition 4.4 is that the system becomes memory–less if the RDR property is maintained because the ability of a reconstruction transition to maintain the RDR property is only influenced by the state of the system in the pre–transition generation.

4.3.2 Criteria for maintaining the Robust Data Recoverability property

In the previous section, we have shown that given a correct set of values for the parameters of the system, the RDR property can be maintained for a large number of generations indefinitely, if we assume the use an infinitely large finite field. Here we give the criteria for the parameters as a set of inequalities.

Our model can be defined using the previously introduced parameters: $N, \alpha, k, d, \beta \in \mathbb{N}^+$. We derive the key constraints for these by examining each state and state change of the system. From the initial state it is possible to conclude that to be able to store the data, we must have at least $N \geq \lceil \frac{k}{\alpha} \rceil$ nodes. However, an extra node is required to be able to handle a loss.

$$N \geq \left\lceil \frac{k}{\alpha} \right\rceil + 1. \tag{4.8}$$

Because the original data is divided into k pieces, there is no reason to store more than $\alpha \leq k$ packets on a single node (there is a k sized cut between the source and the rest of the network).

The defining state changes is the transition between any two consecutive generations l and $l+1$. A node fails in generation l and a recovery node is filled with data in generation $l + 1$ to functionally repair the lost data. To be able to contact enough parent nodes, the recovery node must have access to at least these d nodes. Therefore,

$$N \geq d + 1, \tag{4.9}$$

where the $+1$ is the recovery node in generation $l + 1$. Furthermore, each node stores α packets, therefore it should receive at least that many to make the recoding of α linearly independent packets possible

$$\alpha \leq d\beta. \tag{4.10}$$

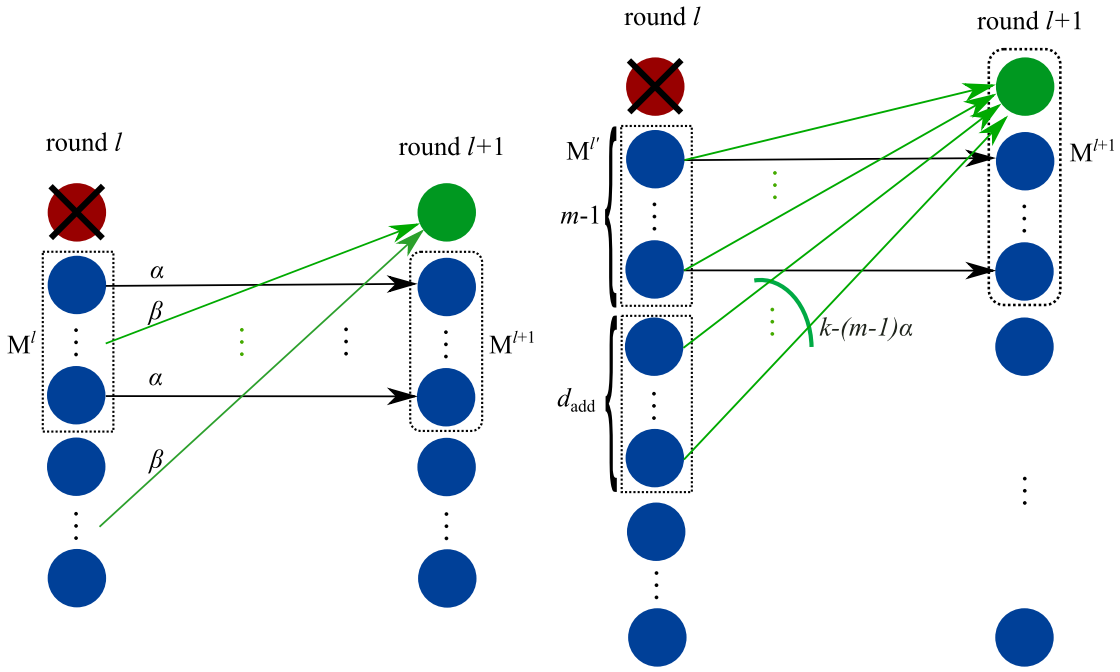
Let us look at how to ensure that the system maintains the RDR property in generation $l+1$, i.e. there exists a network flow with value k between the source and the sink that only traverses nodes from a selected M^{l+1}. This is the same as having k linearly independent packets transferred on k edge–disjoint paths. We assume that this property holds for generation l and let us examine the transition to $l+1$. We denote the set of nodes representing the surviving devices from generation l which are also elements of M^{l+1} with $M^{l'}$ as illustrated on Figure 4.2a.

We can choose m nodes out of N in $\binom{N}{m}$ ways and the parent nodes for the recovery node can be chosen in $\binom{N-1}{d}$ ways. However, there are only three distinct cases to analyze:

- Case I. M^{l+1} does not include the newly joined recovery node.

  In this case, there are no additional constraints on any of the parameters, as the property will hold true regardless of which nodes were used to fill the recovery node. This is because M^{l+1} includes the same devices as $M^{l'}$ for which the property was true.

- Case II. M^{l+1} includes the newly joined node, which was repaired using all $m-1$ nodes in $M^{l'}$ except itself, i.e. $d \geq m-1$.



(a) **Case I.** M^{l+1} does not include the newly joined recovery node.

(b) **Case II.** M^{l+1} includes the newly joined node, which was repaired using all $m-1$ nodes in $M^{l'}$ except itself: $d \geq m-1$.

Figure 4.2: An illustration of cases I. and II.

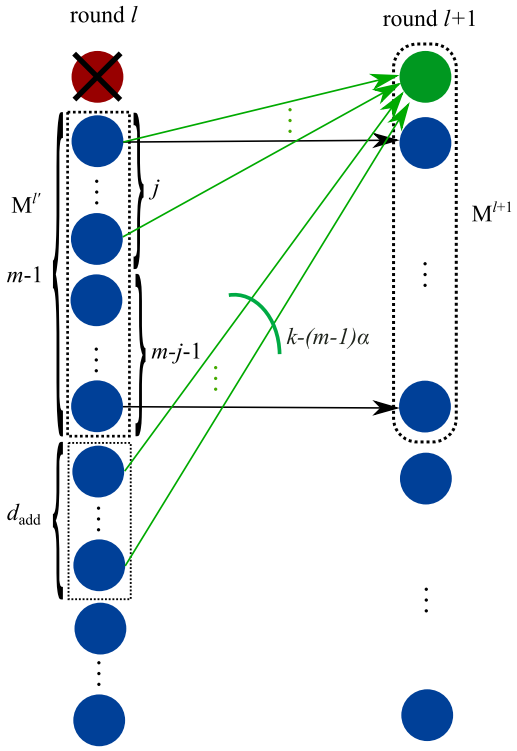


Figure 4.3: **Case III.** M^{l+1} includes the newly joined node, which was repaired using $j = 0, \cdots, m-2$ nodes from M^l.

Each of these parent nodes will supply α edge–disjoint paths. To be able to have a total of k edge–disjoint paths crossing M^{l+1}, additional paths must traverse the recovery node. However, these should not be the same paths as supplied by the parents, as those would not necessarily be edge–disjoint. This can be achieved using additional parents (d_{add}), which results in

$$d_{\text{add}} = \left\lceil \frac{k-(m-1)\alpha}{\beta} \right\rceil = \left\lceil \frac{k-(\lceil \frac{k}{\alpha} \rceil - 1)\alpha}{\beta} \right\rceil \tag{4.11}$$

Considering the additional parents from inequality (4.11) results in

$$d \geq m - 1 + d_{\text{add}} = \left\lceil \frac{k}{\alpha} \right\rceil - 1 + \left\lceil \frac{k-(\lceil \frac{k}{\alpha} \rceil - 1)\alpha}{\beta} \right\rceil \tag{4.12}$$

- Case III. M^{l+1} includes the newly joined node, which was repaired using $j = 0, \cdots, m-2$ nodes from M^l.

  A total of $j\beta$ paths are created between $M^{l'}$ and the newly joined node. With reasoning similar to II., the number of additional paths that must be created between parents not in $M^{l'}$ and the recovery node is the same as described in Equation (4.11).

This gives the following lower bound for the number of parents:

$$d \geq j + d_{\text{add}} = j + \left\lceil \frac{k - (\lceil \frac{k}{\alpha} \rceil - 1)\alpha}{\beta} \right\rceil \tag{4.13}$$

This is a less strict condition for d than Inequality (4.12) because $j < m-1 = \lceil \frac{k}{\alpha} \rceil - 1$ by definition. Furthermore, this case is only possible if: $N \geq (m-j-1)+d+1$ (to have enough nodes in M^l not taking part in the repair) for nodes, which is a more strict condition for N then the one in Inequality (4.9).

Having studied all types of transitions from generation l to $l+1$, we have identified the sufficient conditions to ensure that any randomly chosen M^{l+1} surviving nodes can be used to create a flow with value k in generation $l+1$ if the property was true for generation l. A constraint on the number of nodes can also be expressed using Inequalities (4.9) and (4.12).

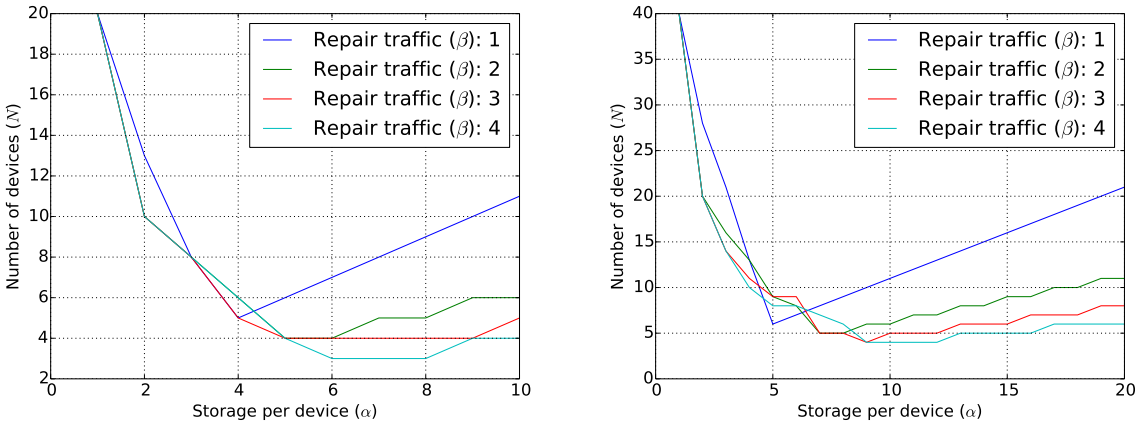
$$N \geq \left\lceil \frac{k}{\alpha} \right\rceil + \left\lceil \frac{k - (\lceil \frac{k}{\alpha} \rceil - 1)\alpha}{\beta} \right\rceil \qquad \forall k, \alpha, \beta \in \mathbb{N}^+ \tag{4.14}$$

Our initial expectation is that these or a subset of these also defines the necessary conditions. We plan to investigate this in the future. Finally, the last state change is the data retrieval itself. However, having already established the constraints to allow the recovery of the data using any m nodes for the repair transition, no new constraints need to be added.

4.3.3 Discussion

In this section we evaluate a storage system with parameters that satisfy all previously presented constraints. First, we examine the number of storage devices N_{suf} that are sufficient to maintain the RDR property between subsequent generations. We derive these results from Equations (4.8) and (4.14). Figure 4.4 shows that by increasing the storage space α on each device, the amount of devices needed initially declines. The reason behind this reduction is that the required edge–disjoint paths used by the repair process can be provided by fewer parent nodes as expressed in Inequality (4.12). However, N_{suf} increases after a point, as the amount of parent nodes required to fill the recovery node increases with α as expressed by Inequality (4.10). The relationship between the parameters of the system shows similar trends for both $k = 10$ and $k = 20$, as well as other values we have examined but not included to save space.

Figure 4.5 shows the same relationship for a wider range of values for repair traffic (β). A similar trend can be observed as in Figure 4.4 for values to the right of the $\beta = \alpha$ plane.

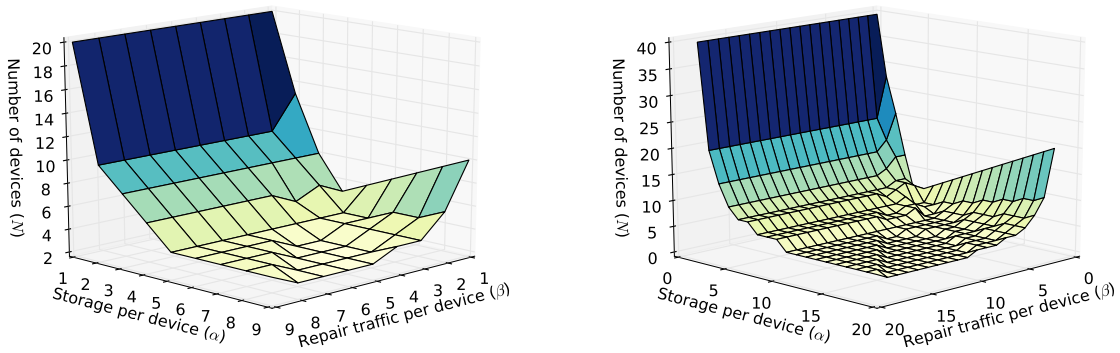


(a) $k = 10$ (b) $k = 20$

Figure 4.4: The minimum number of storage devices needed to safely store data, with constraints on per device storage (α) and repair traffic provided (β) for selected values.

Values to the left of the $\beta = \alpha$ plane are for cases where $\beta > \alpha$. Clearly, such systems have no advantage compared to systems where $\beta = \alpha$, as recoding cannot produce more linearly independent packets to send to the recovery device than was stored on the parent device.

The set of constraints offers a wide range of values for the parameters for which data integrity is guaranteed. This is due to the effectiveness of recoding in the reconstruction process and makes RLNC–based systems cost–effective.



(a) $k = 10$ (b) $k = 20$

Figure 4.5: The minimum number of storage devices needed to safely store data, with constraints on per device storage (α) and repair traffic provided (β) for a wider range of values.

4.4 The performance of erasure coded repair

Section 4.2 presented three different approaches for handling storage in both centrally controlled and decentralized systems and Section 4.3 established lower bounds on the parameters of functional repair. This section looks at how well the codes perform in practice, given the same types of restrictions. Besides single and multiple concurrent node losses, it also covers a sinus–wave–like node change pattern, more typical of real systems.

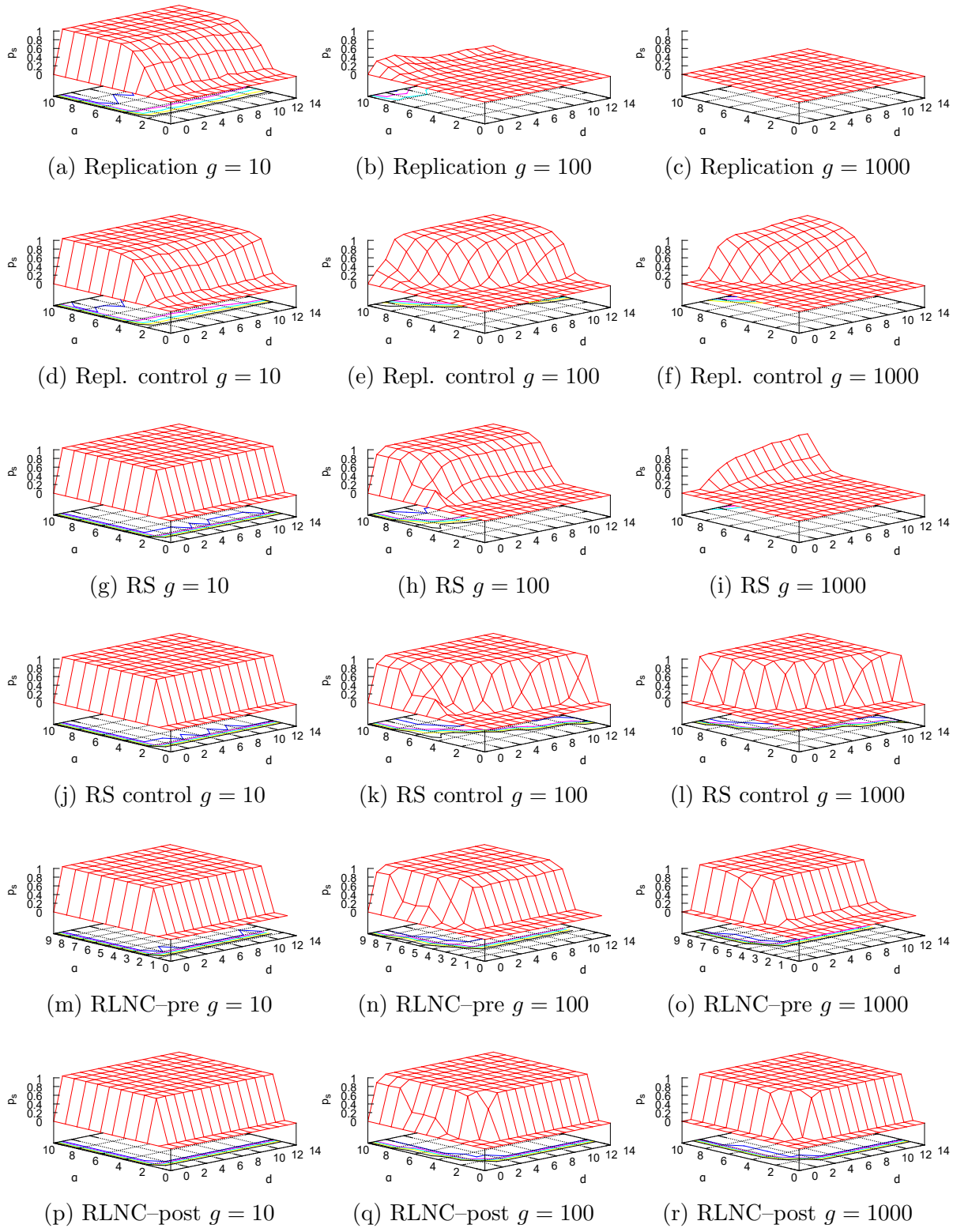
4.4.1 Single node failure

First, we look at how systems behave when faced with the loss of a single node, which should give a good general indication of the resilience of the compared strategies and is the most common case in data centers [Ford et al., 2010]. In Figure 4.6 cloud integrity is given for different coding strategies in relation to the number of available parents and the number of storage place per node α. The first, second, and third column show cloud integrity after 10, 100, or 1000 generations of node failure, respectively. Results except those involving pre–coding use $\lambda = d$.

The first two rows show the replicated approach. Cloud integrity can only be guaranteed for a large value of α, which means high costs in maintaining the cloud infrastructure. In the decentralized case, the number of parents has a smaller impact and for larger generations the probability to retrieve the data is going towards zero. When employing a centrally controlled approach, replication can maintain cloud integrity, albeit at a high cost in storage and network bandwidth.

The third row shows the random RS approach. For 10 generations the cloud integrity can be ensured by a small number of parents ($d \geq 1$) and a small number of storage ($\alpha \geq 2$). For 100 generations the cloud integrity is significantly decreased and can only be ensured by a large number of parents ($d \geq 3$) and a very large number of storage per node ($\alpha \geq 8$). For 1000 generations the cloud integrity can not be guaranteed anymore. However, if a controlling entity coordinates the operation of the cloud system, cloud integrity is maintained for a wider range of parameters than replication. This is shown in the fourth row.

In the final two rows, the post and pre–recoding network coding approaches are presented. Both outperform replication and Reed-Solomon, despite being fully decentralized with no central coordinating controller. Even for large generation numbers such as 1000, both approaches are still robust. The difference for 10 generations is not visible and both clouds are intact for $d \geq 1$ and $\alpha \geq 2$. For 100 generations the pre–recoding approach



(a) Replication $g = 10$ (b) Replication $g = 100$ (c) Replication $g = 1000$

(d) Repl. control $g = 10$ (e) Repl. control $g = 100$ (f) Repl. control $g = 1000$

(g) RS $g = 10$ (h) RS $g = 100$ (i) RS $g = 1000$

(j) RS control $g = 10$ (k) RS control $g = 100$ (l) RS control $g = 1000$

(m) RLNC–pre $g = 10$ (n) RLNC–pre $g = 100$ (o) RLNC–pre $g = 1000$

(p) RLNC–post $g = 10$ (q) RLNC–post $g = 100$ (r) RLNC–post $g = 1000$

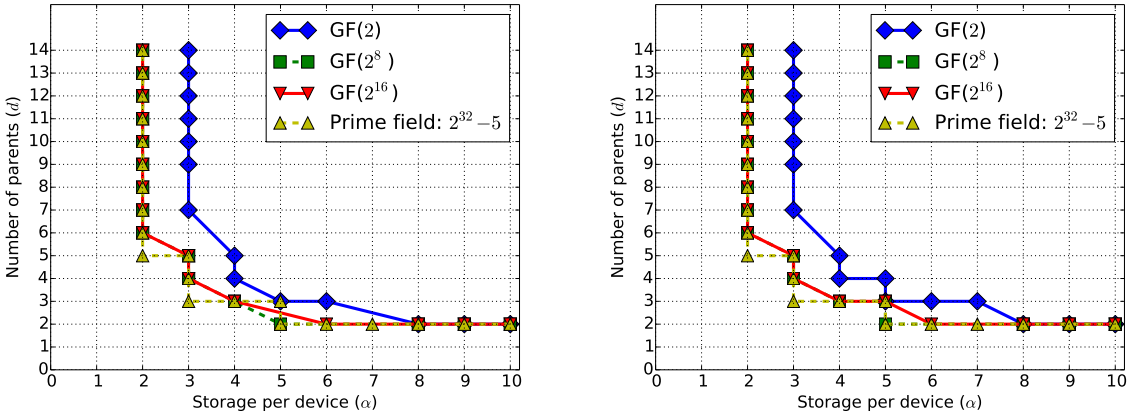
Figure 4.6: Probability of successfully recovering the data after a given number of generations with constraints on storage (α) and the number of parents (d). $N = 15$.

can assure the integrity with $d \geq 2$ and $\alpha \geq 7$ or with $d \geq 4$ and $\alpha \geq 4$ (optimal working point with respect to storage).

The post–recoding approach can ensure integrity with $d \geq 2$ and $\alpha \geq 5$ (optimal working point with respect to traffic) or with $d \geq 5$ and $\alpha \geq 2$ (optimal working point with respect to storage) after 100 generations.

4.4.2 The impact of field size on the effectiveness of recoding

To verify the assumption we made based on prior work [Heide et al., 2011a] that recoding does not introduce a significant amount of linear dependence for RLNC given a large enough field for calculations, we have conducted further simulations with different fields. We chose four fields of practical interest. GF(2) allows for very efficient computations due to the addition operation corresponding to the logical XOR operation and multiplication to logical AND respectively. Both use a small number of cycles on modern CPUs. GF(2^8) is a natural choice for use in erasure codes as its elements can be represented in a byte on most modern hardware architectures. The relative large size of the field is sufficient for use with non-deterministic coding schemes yet small enough that a multiplication and addition table can fit in memory for many systems. GF(2^{16}) has significantly more elements but is slower in terms of encoding and decoding performance. Finally, we have included the field based on the prime number $2^{32} - 5$. It has the benefit of having a very large number of elements as well as relatively low encoding and decoding complexity because addition and multiplication can be performed using efficient integer addition, multiplication and the modulo operation. However, it requires some additional steps to map data prior to encoding, decoding [Paul Crowley, 2006, Pedersen et al., 2013].



(a) $p_S^{1000} = 0.95$ (b) $p_S^{1000} = 0.99$

Figure 4.7: Impact of field size on RLNC with post–recoding: the minimal values for d and α for which data survival is ensured with a given probability p_S^{1000}. Multiple fields are illustrated over which operations are performed.

As in Section 4.3.3, we fixed the values of the parameters $N = 15, k = 15, \beta = \alpha$ and explored different values for α and d. We looked at the success probability p_S^{1000} of being able to recover the data after 1000 generations of simulation.

Figure 4.7 shows a top–down view of the results for RLNC from Figure 4.6. It illustrates the smallest values of α and d for which data survival is ensured with probability p_S^{1000}. GF(2) performs worse than the larger fields, however the difference is relatively small. The larger fields perform almost identically, with the prime field only surpassing GF(2^8) and GF(2^{16}) in a few cases. These results confirm that GF(2^8) is sufficient to ensure data survival for RLNC with high probability over a large number of failure and recovery generations.

4.4.3 Multiple node failures

While concurrent failures in data centers account for only a small part of failures [Ford et al., 2010], it is important to plan for them nevertheless. Furthermore, we expect concurrent node failures to be the norm in mobile clouds.

4.4.3.1 Concurrent node failure

First, let us look at the case where the nodes are lost and rebuilt concurrently, shown on Figure 4.8.

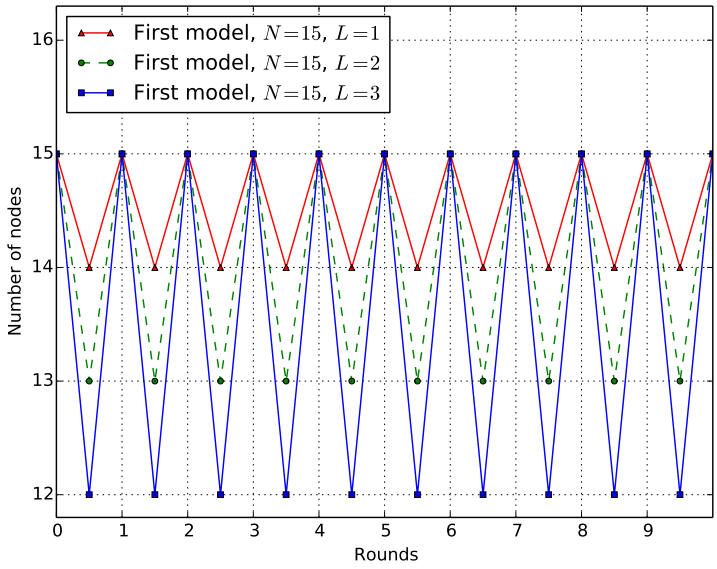
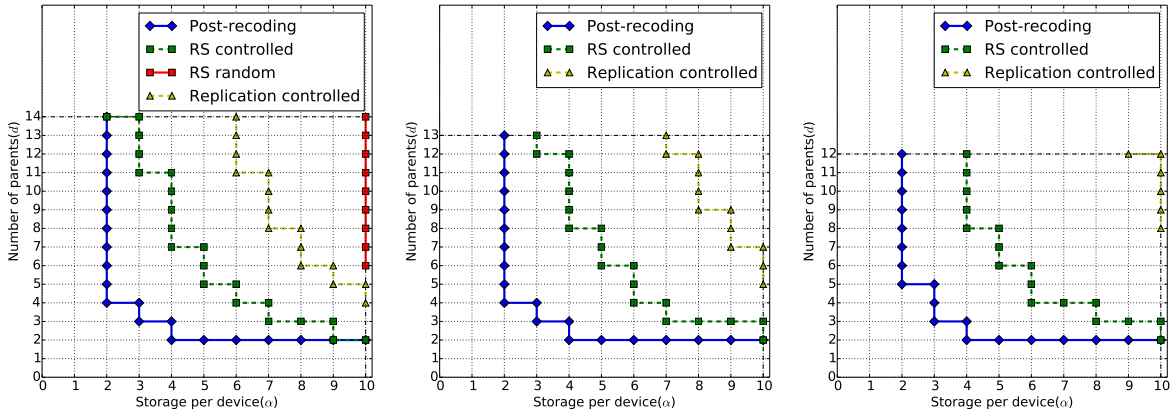


Figure 4.8: First model for node failure patterns, $L = \{1, 2, 3\}$

Figure 4.9 is a top–down view of the last column of Figure 4.6 for $L = \{1, 2, 3\}$. The lines define the plateaus where cloud integrity is maintained with a probability of



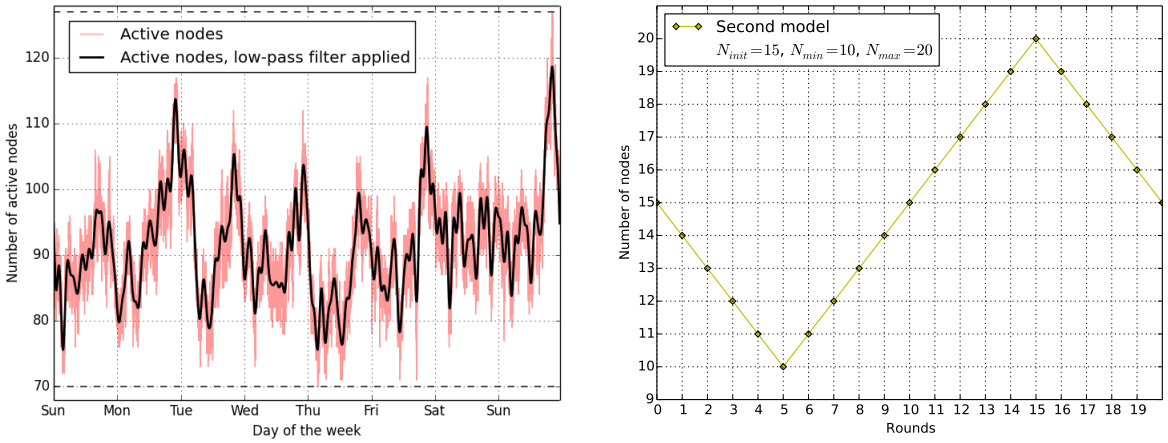
(a) $L = 1$ (b) $L = 2$ (c) $L = 3$

Figure 4.9: First node variance model: the limit of maintaining data integrity after 1000 generations with probability of at least $p_S^{1000} \geq 0.95$. $L = 1, 2, 3$ randomly selected nodes are removed and new ones are added in every generation. $N = 15$.

at least $p_s^{1000} \geq 0.95$ after 1000 generations. We have omitted random replication due to its unsatisfactory performance and pre–recoding for conciseness. There are only subtle differences compared to the single node loss model for erasure coding based approaches, each requiring slightly more storage and/or repair traffic as L increases. Replication performs significantly worse with higher losses, the probability that the last replica of the least common part is lost tends to 1 as the number of generations increases in traffic and storage constrained scenarios.

4.4.3.2 A more realistic failure pattern

Second, let us look at a more realistic failure pattern, motivated by user behaviour in mobile P2P systems with a global user base. Figure 4.10a shows the number of online users of DrTorrent [Andras Bori, 2014], a popular Android BitTorrent client, for a one week period between 23/03/2014 and 30/03/2014. This is a typical example chosen from a set of several months that show similar patterns. To be able to better visualize short–term seasonal trends, we employed a Fast Fourier Transformation and removed all frequencies above 1/343716 Hz (around 4 days) as a low–pass filter. The estimated mean value is $\hat{\mu} = 91.47$ and the standard deviation is $\hat{\sigma} = 8.07$. The sample minimum is $N_{\min} = 70$ on Thursday at 3:42 GMT, the sample maximum is $N_{\max} = 127$ on Sunday 20:13 GMT. There is a distinguishable periodicity that can be attributed to the day–night cycle and the geographic location of users. The number of connected nodes increases during the day until around 20:30 GMT and decreases again until around 03:30 GMT. Other, much larger systems show similar usage patterns. The Steam gaming service publishes the number of concurrent connected users on its website [Corporation, 2017].



(a) Variance in the number of active nodes based on DrTorrent trace

(b) Pattern used in simulations

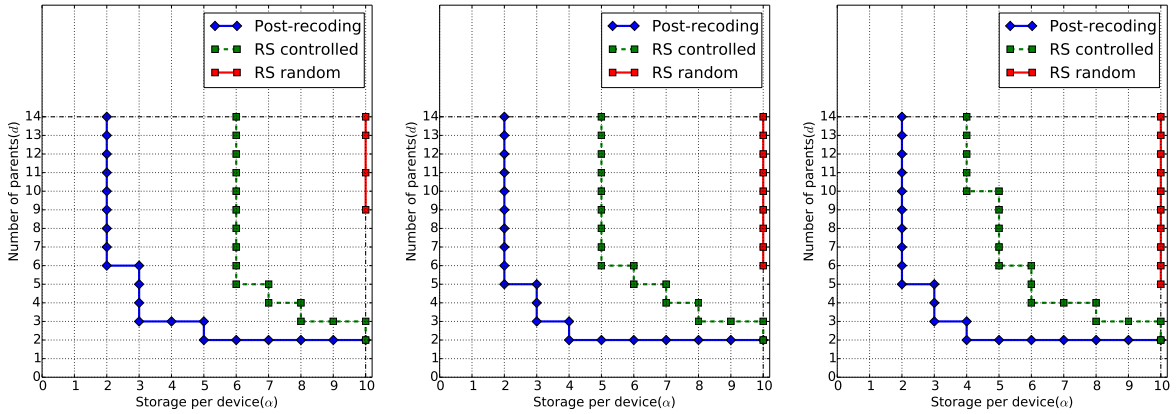
Figure 4.10: Second time–varying model for node variance

Based on these general patterns, we have chosen to use a discrete triangular signal with a step of 1 in each generation for our second model as shown in Figure 4.10b. Compared to our first model, it is a significantly more challenging pattern due to the large fluctuation in the number of nodes. In the descending phase, N falls considerably below the original N_{init} to N_{min}, potentially approaching the theoretical limit of data integrity of $n \geq k$. In the ascending phase, the ability of reconstruction techniques to spread information evenly on new nodes is tested as the number of nodes rises to N_{max}.

We have used the same values for this model as in the previous subsection, most significantly $N_{\text{init}} = 15$. Because the measurement data had a much higher mean value of $\hat{\mu} = 91.47$, we downscaled the variance of the system accordingly. We looked at three sets of parameters: $N_{\text{min}} = 9$ and $N_{\text{max}} = 21$, $N_{\text{min}} = 10$ and $N_{\text{max}} = 20$, $N_{\text{min}} = 11$ and $N_{\text{max}} = 19$.

The much larger differences between N_{max} and N_{min} create a different type of challenge for the various encoding techniques. Figure 4.11 shows the plateaus where $p_S^{1000} \geq 0.95$. A crucial difference compared to previous results is that decentralized Reed–Solomon does slightly better than centralized replication in the scenarios with more variance in the number of nodes. Replication–based codes and decentralized RS both only ensure data integrity if α is large enough to lower the probability of loosing all replicas of a given piece. The number of available parents has a reduced influence on the results.

Post–recoding does best for this model and is barely affected by the increase in node number variance. The recoding mechanism is highly effective in spreading the redundancy among the new nodes evenly. This is crucial as the original nodes eventually drop out.



(a) $N_{\min} = 9, N_{\max} = 21$ (b) $N_{\min} = 10, N_{\max} = 20$ (c) $N_{\min} = 11, N_{\max} = 19$

Figure 4.11: Time–varying second model: the limit of maintaining data integrity after 1000 generations with probability of at least $p_S^{1000} \geq 0.95$. The number of nodes in the system varies in accordance with the second triangular node variance model. $N_{\text{init}} = 15$.

Centralized RS behaves reasonably well, however it needs significantly more storage and network traffic to achieve the same degree of data integrity.

Since the initial publication of our work, some of our experimental results have been validated by theoretical results. [Abdrashitov and Medard, 2015] examined how the rank of the encoding matrix changes over generations of reconstructions.

4.5 The feasibility of mobile clouds

4.5.1 Conditions for a self–sustaining mobile storage cloud

To maintain data availability, the system must fill newly joined nodes (N_{J} denotes their number) with data during the reconstruction process. To do this, it must have parent nodes with a sufficient combined available upload bandwidth. We have aggregated this requirement for the entire mobile cloud for a given period of time:

$$C_h(\Delta t) \leq \frac{B(\Delta t)}{\gamma}, \tag{4.15}$$

where C_h is the rate at which new nodes join the system, B is the total amount of upload bandwidth available from potential parents and γ is the reconstruction traffic for one newly joined node. Δt is the period of time of the observation.

The rate at which new nodes join the system (C_h) provides a means of quantifying how dynamic the system is. We have chosen this metric, because the number of joining nodes determines the amount of reconstruction traffic. Nodes that rejoin already posses

the data, therefore it is enough to only consider newly joined nodes:

$$C_h(\Delta t) = \frac{N_{\mathrm{J}}(\Delta t)}{\Delta t}. \qquad [nodes/\Delta t] \tag{4.16}$$

The aggregated upload capability of the system (B) can be estimated by simply summing together the individual upload bandwidth (B_i) of the nodes, assuming that all connected nodes can be accessed from all joining nodes:

$$B(\Delta t) = \sum_{N_i \in N(\Delta t)} B_i. \qquad [bits/\Delta t] \tag{4.17}$$

If there are difficulties in measuring the individual bandwidth of the nodes, but the connection types are known, the total available bandwidth can be estimated by using fixed values based on the connection type:

$$B(\Delta t) = N_{\mathrm{WiFi}}(\Delta t) \cdot B_{\mathrm{WiFi}} + N_{\mathrm{cell}}(\Delta t) \cdot B_{\mathrm{cell}}. \quad [bits/\Delta t] \tag{4.18}$$

Finally, the amount of data that needs to be transferred to fill one joining node during reconstruction depends on several factors. There is an inherent trade–off between the amount of data that is transferred to a new node and the amount of new, useful information it can build and store from this as shows in the measurements in Section 4.4.

The type of coding is a critical factor in ensuring data availability, as it determines how well the data is spread among the nodes. Some coding techniques can also be customized to be traffic–heavy or a traffic–light. Using RLNC, we vary the parameter λ, with $\lambda = 1$ corresponds to traffic–light pre–recoding and $\lambda = P$ to traffic–heavy post–recoding approach.

The maximum amount of data that can be transferred to every newly joined node limits the total amount of storage the system can use:

$$\alpha_{\max}(\Delta t) \leq \frac{\gamma_{\max}(\Delta t)}{\lambda}. \tag{4.19}$$

This is a potentially important practical limitation, therefore $\gamma_{\max}$ is an important metric of the system:

$$\gamma_{\max}(\Delta t) = \frac{B(\Delta t)}{C_h(\Delta t)}. \tag{4.20}$$

4.5.2 Managing variations in the number of nodes

It is a natural property of the mobile cloud that nodes leave and join at arbitrary points in time. The reconstruction process maintains data availability despite this, however it

is only effective as long as there are enough new nodes to balance out leaving nodes. The system must therefore have some slack to account for time spans when the number of leaving nodes is higher than the number of joining nodes. Having redundancy in the erasure code achieves this slack. In the case of MDS codes, this is another representation of the rate of the code:

$$R = 100\frac{n-k}{k}. \qquad [\%] \tag{4.21}$$

The necessary condition for maintaining data availability for MDS–like codes such as RLNC with an initial node count of N is given using the condition in (4.23). To be able to recover the data after loosing N_L nodes and N_J joining nodes, the coded pieces stored on any $N - N_\mathrm{L} + N_\mathrm{J}$ sized subset of nodes must contain at least the same amount of data as the original file. This condition is only sufficient, if the reconstruction process guarantees that the MDS property of the code is maintained. Section 4.3 presented a set of sufficient bounds for RLNC. We introduce

$$R' = \frac{R}{100} + 1 \tag{4.22}$$

to simplify the formulas and to express the requirement on node dynamics:

$$R' \geq \frac{N}{N - N_\mathrm{L} + N_\mathrm{J}}. \tag{4.23}$$

By employing a number of *always on nodes* (N_A) with very high availability, this requirement can be relaxed. These could be conventional cloud storage providers or servers that include some guarantees in their service level agreements. We have chosen to model these nodes as somewhat separate to the mobile cloud and not include them when calculating R and R' :

$$R' \geq \frac{N}{N - N_\mathrm{L} + N_\mathrm{J} + N_\mathrm{A}} \tag{4.24}$$

4.5.3 User behavior in a mobile P2P system

4.5.3.1 Trace analysis

Let us look at a real–world scenario to characterize node dynamics. DrTorrent is a popular BitTorrent client for mobile devices running the Android operating system with over 50000 downloads as of August 2014. The app collects anonymous usage information regarding the network and battery status of devices and the torrent files being downloaded and shared. The creators of DrTorrent have published a paper [Csorba et al., 2013] detailing some initial long–term trends. We have been given permission to use the collected information to evaluate the feasibility of our proposed system. As both BitTorrent ap-

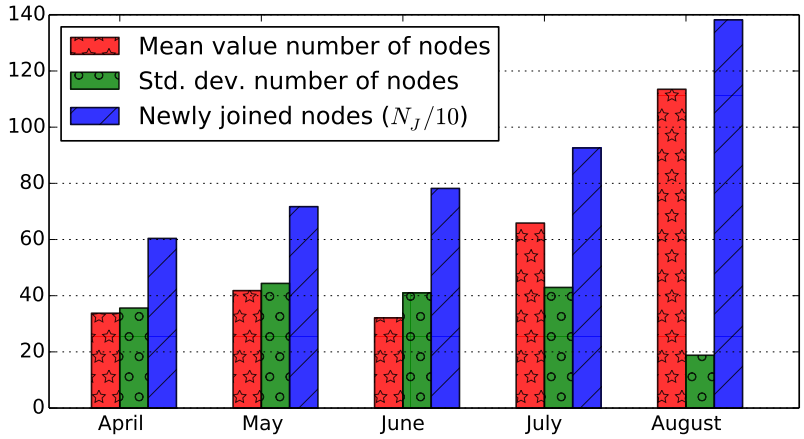


Figure 4.12: The evolution of concurrently online nodes and newly joined nodes in the middle of 2014. To be able to show the large number of node joins (almost 1400 in August), the values have been divided by 10.

plications and mobile storage clouds are P2P storage systems, we believe the traces are relevant and suitable for evaluation. Besides the common underlying architecture, both are community–driven and characterized by a constant stream of users joining and leaving. Both use the same resources: the storage and available network bandwidth of user devices. In BitTorrent systems a certain number of seeders and leachers with a high completion ratio are needed to be able to retrieve information. Likewise, mobile clouds must contain some nodes that remain connected even when not actively retrieving information. These underlying similarities motivate using the DrTorrent traces.

As the popularity of DrTorrent increased, the variance in the number of connected nodes stabilized somewhat as shown on Figure 4.12. We have therefore decided to use the latest results and limit our evaluation to the month of August 2014. Figure 4.13 shows the number of active nodes during this period with samples taken every second. The short–term variance is relatively low and shows a periodicity typical of global systems with a large number of users. This can be attributed to a certain degree to the day–night

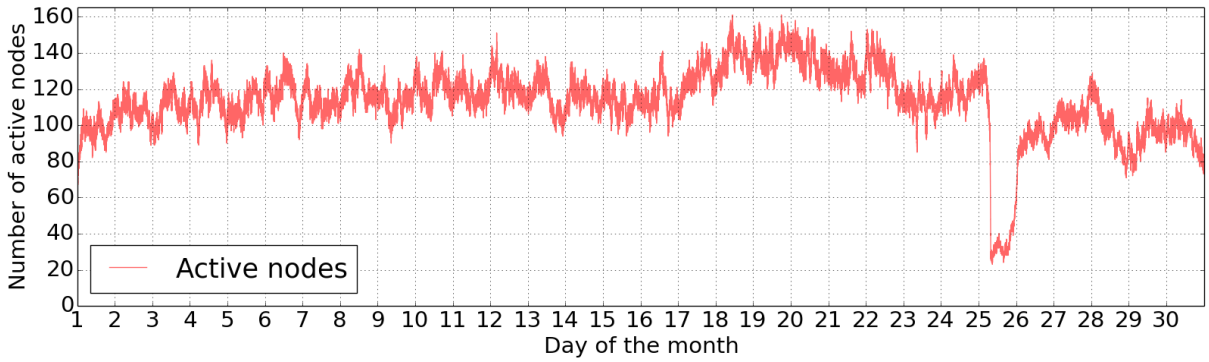
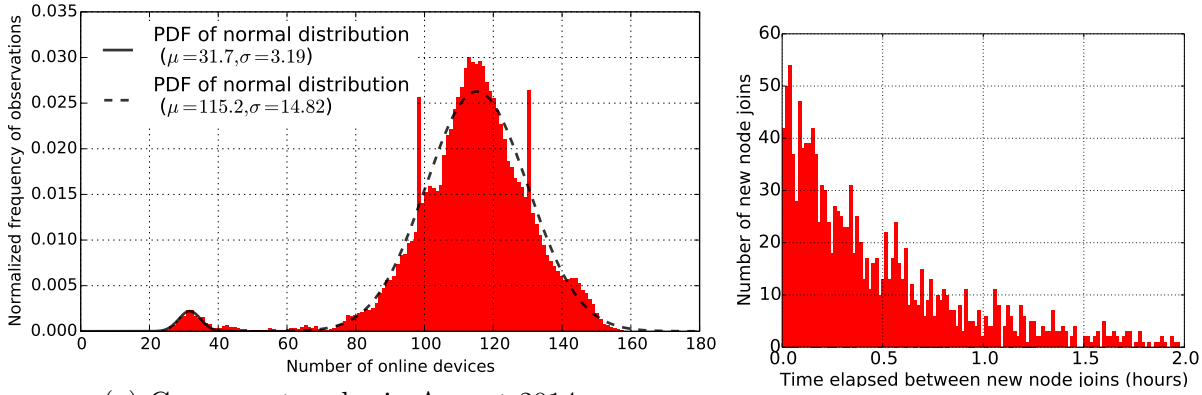


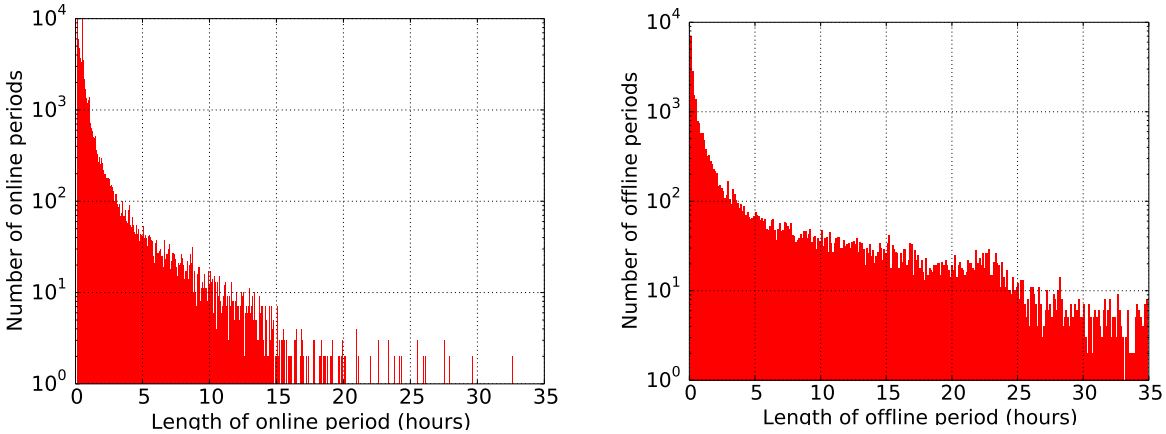
Figure 4.13: The number of concurrently online nodes in August 2014.



(a) Concurrent nodes in August 2014.

(b) Time between new node joins

Figure 4.14: Histograms showing the number of concurrent online users and elapsed time between node joins.



(a) Time users spend online

(b) Time users spend offline

Figure 4.15: Histograms showing the usage patterns. Values below 5 minutes have been removed to eliminate reconnects due to technical issues.

cycle and the globally distributed user base. However, there are periods – such as the 25th of August – when the number of nodes falls considerably bellow the mean value. We have no knowledge of the cause of this event. Figure 4.14a shows the histogram of the number of connected nodes. Values over 60 have a close to normal distribution with a mean value of $\mu = 115.2$ and a relatively low standard deviation of $\sigma = 14.82$. This suggests based on the central limit theorem, that the number of similar random processes – individual user actions – is high enough to make statistically sound conclusions. Values between 20 and 40 are caused by the large node loss on the 25th. We have fitted a normal distribution with a mean value of $\mu = 31.7$ and standard deviation of $\sigma = 3.19$. Values between 40 and 60 are caused by the transition from the high node numbers to the low numbers. We argue that traces for August 2014 are suitable to base our evaluation on, because the data set shows both normal operation and a challenging situation it may face.

4.5.3.2 Individual user behavior

This subsections presents figures and statistics about the individual behavior of DrTorrent users relevant to distributed storage. These are generally at a lower level of abstraction compared to our model and are presented here to give an insight into their influence on higher–level processes.

Figure 4.14b is an illustration of the dynamics of the system, showing a histogram of the elapsed time between subsequent new node joins. Values below 30 minutes dominate. Figure 4.15a shows the typical length of online sessions. Most values are below 1 hour, with higher values becoming increasingly rarer. The very low values can be attributed in part to issues related to BitTorrent applications in general: failed downloads caused by not having enough peers to connect to, users not being able to use the application without the appropriate .torrent files and so on. Figure 4.15b shows the offline times between online sessions. Low values are more numerous than high values, but the effect is less pronounced compared to the online periods. Most users use the application at least once a day. All three figures demonstrate that it is a very dynamic P2P system.

The connection type is important in terms of the available bandwidth it offers. Examining the trace, 73.6% of the time devices connected to the Internet using WiFi and 25.9% of the time using a cellular network, while the rest of time ($< 0.5\%$) devices used either Ethernet or WiMAX. As the measurement involved devices using mostly battery–powered devices, a critical aspect was the power status of the nodes. The traces show an unexpected result: the devices are being charged 69% of the time while the application is running. This is an indication that users are aware of the energy implications of data and computation–intensive applications and are willing to adapt their habits to a certain degree.

4.5.4 Bounds on required bandwidth

In this subsection we apply Equation (4.20) to calculate $T_{\max}$, the maximum amount of data that can be transferred across the network to each joining node. We have computed the available upload bandwidth by counting the number of connected nodes using WiFi and cellular with a resolution of 1 second and applied it to Equation (4.18), with values of $B_{\text{WiFi}} = 8.7$ Mbit/s and $B_{\text{cell}} = 3.27$ Mbit/s. These are the global average upload rates for the month of August 2014 as measured by Ookla [Ookla, 2014], one of the largest public services for measuring bandwidth. First, we have applied the formula over the whole month of August to get a first estimate. The average values for this period are $B = 828.25$ Mbit/s and $C_h = 1412$ nodes/month, making $\gamma_{\text{max,month}} = B/C_h = 1\,571\,094$ Mbit/node. Whilst only a ballpark figure, it far exceeds the storage capacity of current

devices. We continued by applying Equation (4.20) for several much shorter time intervals over the entire month: 1 day, 1 hour and 10 minutes respectively. This gave sets of more exact values and the lowest values from each set are better indications of the real bandwidth limitation: $\gamma_{\text{max,day}} = 1\,259\,343$ Mbit/node, $\gamma_{\text{max,hour}} = 318\,798$ Mbit/node, $\gamma_{\text{max,10min}} = 86\,909$ Mbit/node. There were time spans in the evaluated period when the ability of the network to fill new nodes decreased significantly, but the calculated limit is still high enough to not pose a practical limitation.

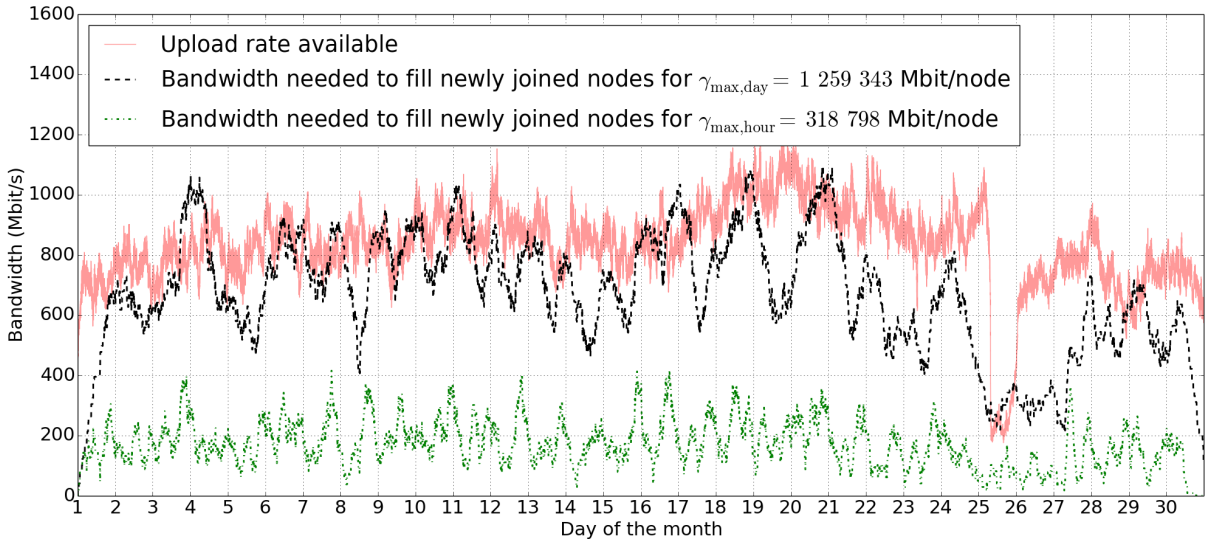


Figure 4.16: Upload bandwidth available and the amount of bandwidth needed for $\gamma_{\text{max,day}}$ and $\gamma_{\text{max,hour}}$.

To evaluate the accuracy of these theoretical values, we simulated the node join and loss processes based on the trace with a 1 second resolution. We looked at how much upload bandwidth B_{req} was required to transfer γ_{max} data to every newly joined node as part of the reconstruction process. We took into consideration the limited download bandwidth of joining nodes, spreading transfers out over time. We consider the mobile cloud self–sustaining as long as $B \geq B_{\text{req}}$ is achieved at all times. Figure 4.16 shows that the values of $B_{\text{req,day}}$ for $\gamma_{\text{max,day}}$ give good estimations of the overall capability of the mobile cloud. However, values of $B < B_{\text{req,day}}$ persist for periods of time in excess of 12 hours, impairing self–sustainability for those periods. On the other hand, storing only $\gamma_{\text{max,hour}}$ ensures self–sustainability with a significant margin. This suggests that taking hourly mean values for node dynamics over a period of 1 month provides a safe estimate on the maximum amount of data that can be stored on individual nodes. Since even the most conservative calculated values are an order of magnitude higher than the typical available

storage space on most mobile devices, we can conclude that self–sustaining mobile storage clouds are feasible in this respect.

4.5.5 Measuring the required level of redundancy

This subsection looks at how much redundancy would have been needed to provide enough slack to ensure data availability throughout the month of August. All results are shown on Figure 4.17. First, we used Equation 4.23 to evaluate a pure mobile cloud without high–availability nodes ($N_A = 0$). We select $N = 115$ as it is close to the mean value for the number of connected users in the month, all other values originate from the trace. As expected, no redundancy ensures data availability around 50% of the time. $R = 75\%$ increases this to 99.28% and 100% data availability is first ensured for $R = 200\%$. Pure mobile clouds are therefore feasible in this regard, requiring levels of redundant storage close to those used in data centers [Shvachko et al., 2010].

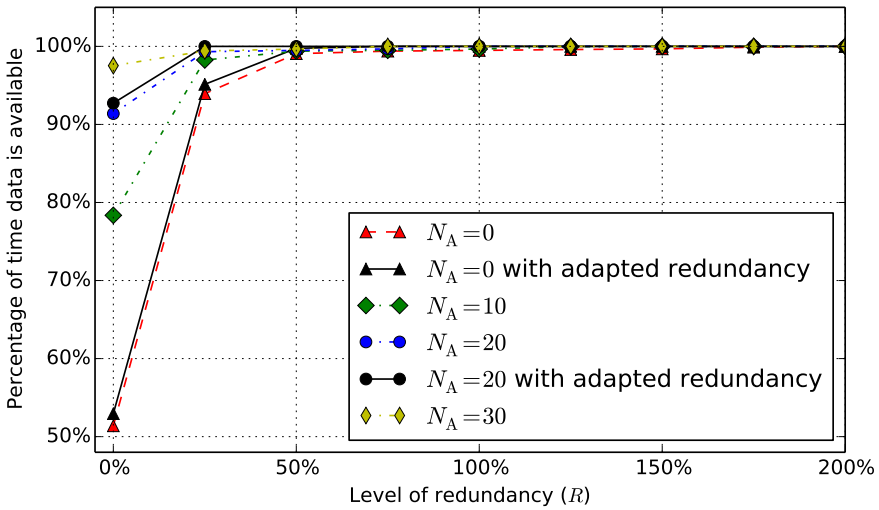


Figure 4.17: The relationship between data availability and storage redundancy

Second, we employed Equation (4.24) to evaluate how adding *always–on* nodes changes data availability. This brings the model much closer to fog computing. It increased the time data was available significantly, especially for low values of R, providing 100% data availability for $R = 100\%$ and $N_A = 20$. Furthermore, because mobile clouds used in hybrid systems with always–on nodes do not necessarily require 100% data availability, they can be deployed effectively, even when using lower levels of storage redundancy.

Third, we investigated the effect of adding redundancy on–demand, when a significant drop in the number of nodes is detected. Figure 4.13 shows such a drop on the 25th of August. We assumed that the system would be able to detect the sudden loss of nodes quickly enough to compensate. We doubled the amount of stored pieces on each nodes for this day. Once the number of nodes rebound the following day, we removed the extra

pieces. This resulted in a significant improvement in data availability, especially when employing always–on nodes. Thus, if the system is able to adapt to changing conditions, it could employ much lower levels of redundancy in the long term. For $N_{\mathrm{A}} = 20$, the mobile storage only needed 50% redundancy to achieve data availability 100% of the time.

4.6 Predicting node availability

This section presents an evaluation of a two–stage model that predicts the number of concurrent nodes in a system. It can be used to plan for a given level of redundancy, effectively managing the network and storage loads experienced by the nodes. Lacking real–world data of sufficient granularity and scope, we created a simulation to generate inputs to our proposed model. Thus, the goal of this section first and foremost is to provide a framework for calculating how much redundancy is needed. The expected accuracy of such predictions in a real–world setting is hard to discern, our experimental results serve more as examples.

4.6.1 Training data

We have written an application in C++ that simulates the behavior of nodes in a P2P network. For the sake of simplicity the network addresses of the nodes are set explicitly and a specialized (central) node polls the other nodes once every 5 seconds. The other (simple) nodes respond with a YES/NO flag, signaling whether they are available or not (this is the target variable that we want to predict) and also send several key values on their condition. These values are the ones that the predictive model described in Subsection 4.6.2 is trained with. The simple nodes are initialized with the following variables: device id, battery capacity when new, available storage, age, and mobility, a synthetic variable.

The central node collects the following parameters from the simple nodes:

- the id of the device,
- the average number of nodes it could reach, measured from the last poll,
- the percentage of time the node was unable to reach any other nodes, measured from the last poll,
- the charge level of the battery,
- the available storage space on the device,
- the age of the device, and
- the reason, in case of failure (network unavailability, discharged battery, insufficient storage space, miscellaneous malfunction).

The last parameter was included in order to check the validity of the simulation, but was not used in building the model since it directly correlates with the target variable. These parameters (except for the first and the last one) are recalculated every 5 seconds. The charge level of the battery decreases each time by a random value. Recharging is simulated based on the current battery charge level, the lower the level, the higher the probability that the user recharges it. The available storage space increases or decreases randomly as well, with a decrease having a higher probability. When it drops below a certain level, the user might make a larger amount available again. The network parameters are simulated based on the mobility variable with which the node was initialized. A higher mobility simulates a user which is more active and as a result the node is more likely to lose contact with neighboring nodes. The passage of time is simulated at an increased rate, each cycle corresponds to 1 hour of operation. This enables us to analyze a longer period of time, taking into account the storage habits of users and battery degradation amongst other things.

4.6.2 Local predictions at the nodes

It is important to first define what we understand by the term availability. According to the Merriam–Webster Dictionary [Merriam-Webster, 2013] a resource is available, if it is *present or ready for immediate use*. Let us consider a node available only if it satisfies both of these conditions. Note that reachability is not the only important parameter for distributed storage, for example a node with a good network connection, but insufficient storage space or available computational capacity cannot be used to store files and as such, should be considered unavailable.

The role of this step is to predict whether a node will be available during the following cycle based on the value of a series of variables. For this task we used a binary classification model with two categories: *Available* and *Unavailable*. There are several methods for creating such a model and one of the simplest is based on logistic regression, a type of regression analysis used [Hosmer and Lemeshow, 2000] to predict the value of a categorical dependent variable. This simplicity enables a low energy implementation on mobile devices that is a common requirement in today's battery–limited era. We have also considered using other techniques and evaluated them with the same data. We selected a naive Bayes classifier [Rish, 2005] and one based on an alternating decision tree [Freund and Mason, 1999], both work based on fundamentally different principles compared to logistic regression.

Table 4.1 represents the results of the comparison between the three classifiers. It can be seen that the naive Bayes classifier is significantly faster, but has lower accuracy, especially in the case of available nodes. The alternating decision tree offers better ac-

	Logistic regr.	Naive Bayes	AD Tree
Accuracy (Overall)	78.3%	75.4%	81%
Accuracy (Available)	94.3%	82%	95.9%
Accuracy (Unavailable)	38.5%	57.7%	44.1%
Model build time	7.85s	2s	82.93s

Table 4.1: A comparison of classification methods

curacy, but at the cost of significantly longer model build times and therefore higher computational complexity and energy demands. In a mobile scenario where the model is refreshed regularly to account for variances in network conditions, the method with a lower run time is advantageous, whereas in a scenario with infrequent model rebuilding the one with the highest accuracy is preferred. In this case, logistic regression is a good compromise between accuracy and computational complexity.

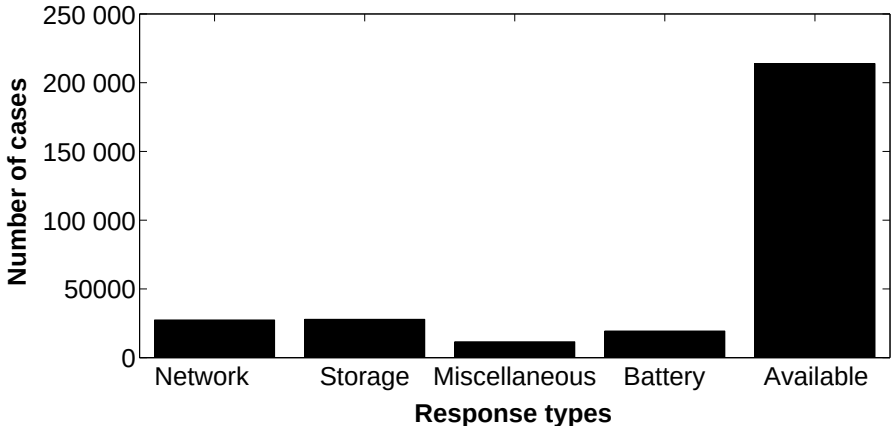


Figure 4.18: The frequencies of the responses

Figure 4.18 highlights the number of the cases where devices were unavailable along with the corresponding reasons. There are 4 such reasons: no network connection, insufficient storage space, the battery is discharged, or a general malfunction has occurred. For our purpose of predicting the target variable, it is not necessary to predict each of these individually, but rather if any of them will occur.

	Run1	Run2	Run3	Run4	Run5
Accuracy (Overall)	78.3%	78.1%	78.2%	75.5%	79.1%
Accuracy (Avail.)	94.3%	88.6%	92.1%	95.1%	94.4%
Accuracy (Unavail.)	38.5%	57.9%	48.3%	24%	40.7%
Model build time	7.85s	7.45s	6.73s	6.46s	8.57s

Table 4.2: Prediction accuracy for five different models

We performed five simulations and the resulting data was used to build five models with significantly differing parameters. The overall accuracy of the predictions based on

these models was similar, therefore subsequent calculations have been performed using only the results from the first run. The prediction accuracy for the two classes as well as the build times of the models are shown in Table 4.2.

The training set was generated using the simulation described in Section 4.6.1. 10000 cycles were run with 30 nodes resulting in a training set of 300000. To test the results of the prediction, we employed 10–fold cross–validation when building the model. In order to ensure that the computed coefficients for the variables are up–to–date, this process must be repeated from time to time. It is beyond the scope of this thesis to determine its optimal frequency.

The training set was used in its entirety to build a single model. However, the simulated devices all had different characteristics, therefore it is safe to say that they behave quite differently. A separate per–device model would be preferred as this should in theory outperform the general model that we have described. We have abstained from this to be able to give mathematical guarantees on the accuracy of the aggregation presented in the following subsection. However, we did build per–device models with 1/30 part of the data to see whether this was a reasonable compromise. These showed quite large variance in their accuracy, but the mean was close to that of the shared model.

4.6.3 Centralized predictions

The second step of our proposed technique is to aggregate results from the first step. Nodes report their individual predictions to a central location. There are several ways to aggregate results, the simplest is to add the results together. Provided the expected value of the error for the estimation at the node level is known and equal for all participating nodes, we can calculate the correctness of our estimation with this simple technique. A further requirement is for the local results to not influence each other. This comes naturally from the distributed nature of the estimation and can also be verified using a Chi–square test with random sampling.

Let us suppose we have N participating nodes. Out of these, let us define M as the number of those responding positively, stating that they will be available during the next cycle. Let $X_1..X_M$ be the Bernoulli indicator variables of the predictions of these nodes. These have to independent and identically distributed as stated less formally in the previous paragraph.

$$\text{Let } \begin{cases} X_i = 1 & \text{if the prediction is correct} \\ X_i = 0 & \text{if the prediction is incorrect} \end{cases}$$

Based on the training set, we have a statistical estimate for $P(X_i = 1) = 0.943$ and $P(X_i = 0) = 0.057$. Similarly, let K denote the number of nodes responding that they

will be unavailable and $Y_1..Y_K$ the independent identically distributed indicator variables of their predictions:

$$\text{Let } \begin{cases} Y_i = 1 & \text{if the prediction is incorrect} \\ Y_i = 0 & \text{if the prediction is correct} \end{cases}$$

Note, that here we assign the two values in the opposite manner to simplify calculations later. We are allowed to do this as $P(Y_i = 1) = 1 - P(Y_i = 0)$. Based on the training set, we have a statistical estimate for $P(Y_i = 1) = 0.615$ and $P(Y_i = 0) = 0.385$.

We are interested in both $\sum X_i$ and $\sum Y_i$ to estimate the number of nodes that will be available. $\sum X_i$ is the expected number of nodes that predict correctly that they will be available, $\sum Y_i$ is the number of nodes that incorrectly predict that they will be unavailable. The sum of independent Bernoulli type indicator variables is a random variable with a binomial distribution, which makes the calculations simple.

$$\text{Let } B_x = \sum X_i \text{ and } B_y = \sum Y_i,$$

where both B_x and B_y are random variables of binomial distribution.

We can easily calculate the expected value of both using the formula for the binomial distribution $\eta = np$, where η is the expected value, n is the number of experiments and p is the probability of one experiment:

$$\begin{aligned} \eta_{Bx} &= MP(X_i = 1) = 0.943\,M \\ \eta_{By} &= KP(Y_i = 1) = 0.615\,K \end{aligned} \tag{4.25}$$

and $N_A = \eta_{Bx} + \eta_{By}$ will be the estimation on the number of available nodes.

This is useful, but so far we do not have a measure of certainty for our estimation. To do this, we can use the formula for calculating the cumulative probability of a binomial variable. The cumulative distributed function is the following:

$$P(X > x) = 1 - \sum_{i=1}^{\lfloor n \rfloor} \binom{n}{i} p^i (1-p)^{(n-i)} \tag{4.26}$$

By calculating both $P(B_x > j)$ and $P(B_y > l)$ for several j and l, where $j \in [0, M]$ and $l \in [0, K]$, we will get several predictions with various levels of confidence. We can select the one appropriate for our scenario and based on this we can calculate the number of available nodes for that level of confidence by simply adding j and l together. Let N_A be the predicted number of available nodes and $N_\text{A} = j + l$ for the chosen level of confidence.

For example, suppose we have $N = 100$ nodes and out of these $M = 75$ nodes respond with YES, $K = 25$ with NO. The probabilities for X_i and Y_i are those previously mentioned.

In this case the estimate for the number of available nodes is:

$$\begin{aligned} \eta_{Bx} + \eta_{By} &= P(X_i = 1) \cdot M + P(Y_i = 1) \cdot K \\ \eta_{Bx} + \eta_{By} &= 0.943 \cdot 75 + 0.615 \cdot 25 = 86.1 \end{aligned} \tag{4.27}$$

By calculating the cumulative probabilities for B_x and B_y for several values j and l we get the following table:

j	25	65	66	67	68	75
Confidence	100%	99.6%	98.9%	97.3%	93.6%	1.2%

l	5	9	10	11	12	25
Confidence	100%	99.7%	99.1%	97.6%	94.2%	0.0%

Table 4.3: Confidence levels for various values of j and l

For example, for $j = 67$ the level of confidence is 97.3%, for $l = 11$ the level of confidence is 97.6%. In this case we estimate the number of available nodes to be $N_{\mathrm{A}} = k + l = 78$ with a lower bound of 97.3% on the level of confidence. We can calculate the required redundancy based on this. For 100 percent redundancy we need to distribute approximately $2N_{\mathrm{A}} = 156$ fragments of information assuming any subset of size N_{A} is enough to recover it. It is also possible to calculate N_{A} for a given level of confidence more accurately using Chernoff's inequality.

So far, we have considered the simplest case, with local predictions done solely based on the model built in Section 4.6.2. In an extended model, where a node has further knowledge that could influence its prediction, only η_{Bx} should be taken into consideration. This is because in this case there are nodes which signal that they will be unavailable based on a certain event which modifies the probabilities $P(Y_i = 1)$ and $P(Y_i = 0)$. Devices might opt–out without penalties when it is very likely that they will be unavailable. An example of this is when a user sets the phone to turn off its network interfaces when the battery discharges below a certain level. Another one is checking the calendar of the mobile device and opting–out if the user is set to go abroad and does not have data roaming activated. It is also possible to further enhance prediction precision using, amongst other things, location–based services [Feher and Forstner, 2011]. Only considering η_{Bx} seemingly lowers the accuracy of the aggregation, but assuming that such further knowledge makes the local predictions more accurate, this is no longer true. Therefore, a useful

addendum to the presented technique is to enable the nodes to have three kinds of predictions: YES/NO/OPT–OUT. If we assume that the probability of a node incorrectly opting–out to be 0, we can use the above presented formulas with just one modification: N should be equal to the number of nodes that didn't opt–out.

The aggregation can be performed using more elaborate methods as well. For example, if $X_1..X_N$ and similarly $Y_1..Y_N$ are not identically distributed, the above formulas can no longer be applied. Indeed, this is a more correct assumption, since it should be more accurate for the nodes to build different models with different error rates. This makes calculating the precision of the aggregation more difficult, but also increases overall accuracy because we can safely assume that the local predictions will be at least as accurate given a training set of equal size.

In order to ensure that nodes that perform poorly or are trying to cheat with their personal predictions do not cause availability issues or degrade the performance of the system, a penalty scheme should be put in place. The simplest way to penalize a node is to disregard its prediction and exclude it from active service for a predefined duration of time in case of an incorrect prediction. To ensure that nodes that do this more often are penalized more, this duration of time can be increased with each incorrect prediction. Other solutions present themselves as well: a simple multiplier that expresses how many times the node has incorrectly reported its availability or a ranking system based on more complicated assessments. These might make calculating the confidence of the aggregated prediction more complicated, but if applied correctly, may further increase overall accuracy.

4.7 Conclusion

More and more storage, network capacity and computations can be found at the edge of the network and more and more services require a large amount of computations, low latency and large throughput. This trend is likely to continue with the introduction of 5G. Many of these services work with shared data, thus it makes sense to store it, at least partially, on user devices. The core idea behind fog computing is to take advantage of this and move communication and storage away, at least partly, from the backbone infrastructure.

This chapter looked at employing network coding for fog computing as the enabler of a mobile storage cloud. Moving to this highly dynamic setting presented more opportunities to use RLNC's flexibility compared to the scenarios described in the previous two chapters. We showed that conventional erasure codes do not work well in decentralized systems, where nodes perform data reconstruction without a central node to direct the process.

The random coefficient selection used during recoding made it particularly suited to this scenario. RLNC's rateless nature was also a natural fit when handling the continuously changing number of nodes.

We introduced theoretic bounds with realistic considerations to characterize recoding–based repair and created a simulation environment to assess how it compares to more traditional approaches. Interestingly, RLNC outperformed Reed–Solomon and replication codes even when a central entity to control the reconstruction was present.

Finally, we looked at whether these systems can sustain themselves. We have shown that even using current technology, mobile storage clouds have enough aggregate bandwidth to deal with typical changes in node numbers. We have also proposed a two–phase mechanism to predict user behavior in mobile environments.

Chapter 5

Updating erasure–coded data

5.1 Introduction

The goal of this chapter is to address the efficient update of encoded fragments from a file that has been modified, i.e. where bytes have been changed, added, or removed. This is a key issue that has not been thoroughly studied for network codes (or other erasure codes) and where the state–of–the–art follows a costly approach [Esmaili et al., 2013], namely, to update encoded fragments entirely regardless of the nature or size of the change. Beyond a strain on the network, this process also requires a large use of space, especially if several versions of the file need to be maintained. The reason for this issue comes from the fact that each encoded fragment is created by the combination of all fragments from the original file. Thus, even small changes in various fragments can be compounded to large changes in the encoded fragments.

5.1.1 Structure and overview of the contributions of this chapter

Our main contribution is a mechanism that exploits the linear operations of erasure codes to provide a means to update encoded fragments. Our solution does not send the original data and supports version control in encoded storage systems while reducing network use and storage use of the overall system. The mechanism supports parts of files being added, deleted, or individual bytes being modified. It is useful in distributed storage using linear erasure codes and is fully compatible with random linear network codes and other network code variants. Crucially, it does not introduce changes to the encoding and decoding processes, thus it can be viewed as a "bolt–on" solution for existing storage systems.

To get a better understanding of how our proposed technique may work in practice, we focused the analysis of its effectiveness on an application area that is particularly update–

heavy: version control. We use a Git repository with the source code for an actively maintained software library. We consider systems such as Git [Chacon, 2009] as state of the art and use it as a baseline to evaluate the storage and transmission overhead of our proposed solution. We have no knowledge of any version control systems in use today that supports erasure codes.

This chapter is organized as follows. Section 5.2 describes how an erasure–coded distributed storage system can store multiple versions of a file using elements from Chapter 3 as an example. Section 5.3 presents the problem of updating encoded data following three types of changes and proposes a solution that can be applied to all three with slight changes. Formulas describe the storage and network overhead of our solution. With the goal of increasing the practical value of our work, Section 5.4 and 5.5 present the details regarding the implementation of our solution, including an algorithm that remaps data to set aside an element from a finite field for signaling purposes. Section 5.6 describes our experiments using a Git repository to establish the effectiveness of our solution. Finally, Section 5.7 presents the conclusions and future challenges.

5.2 Description of cloud–based version control

Figure 5.1 shows an overview of our proposed solution, seen as an extension to the cloud storage system described in Chapter 3. We use this scenario to better describe the high–level workings of our solution and to tie theory to practice. We have strived to make our algorithms as general as possible to have very few technical requirements. Thus, they can be applied in a multitude of other scenarios. The figure shows the state of the system where a file has previously been encoded and distributed to five clouds and the client application has just changed a part of the file. First, the client calculates the difference between the latest and the newly modified version of the stored file. The difference can then be divided into a number of *unencoded difference packets* by the client, which are then linearly combined using the same coefficients as used for the original distribution of the file. This generates a somewhat larger number of *encoded difference packets*. Each of these are linked to a specific *encoded packet* already stored on one of the clouds based on which coefficients were used to create it. If the *encoded difference packets* contain only a few non–zero elements, they can be effectively compressed before being uploaded to the clouds. This is possible if the changes occur in bursts, only affecting a localized part of the original file, as shown in the example. The encoded differences can then be stored to give a simple and storage–friendly form of version control. If the cloud can perform simple computations on the data, such as XORs, the difference can also be applied to update the stored files to the newest version. If this is not possible, the client must retrieve both

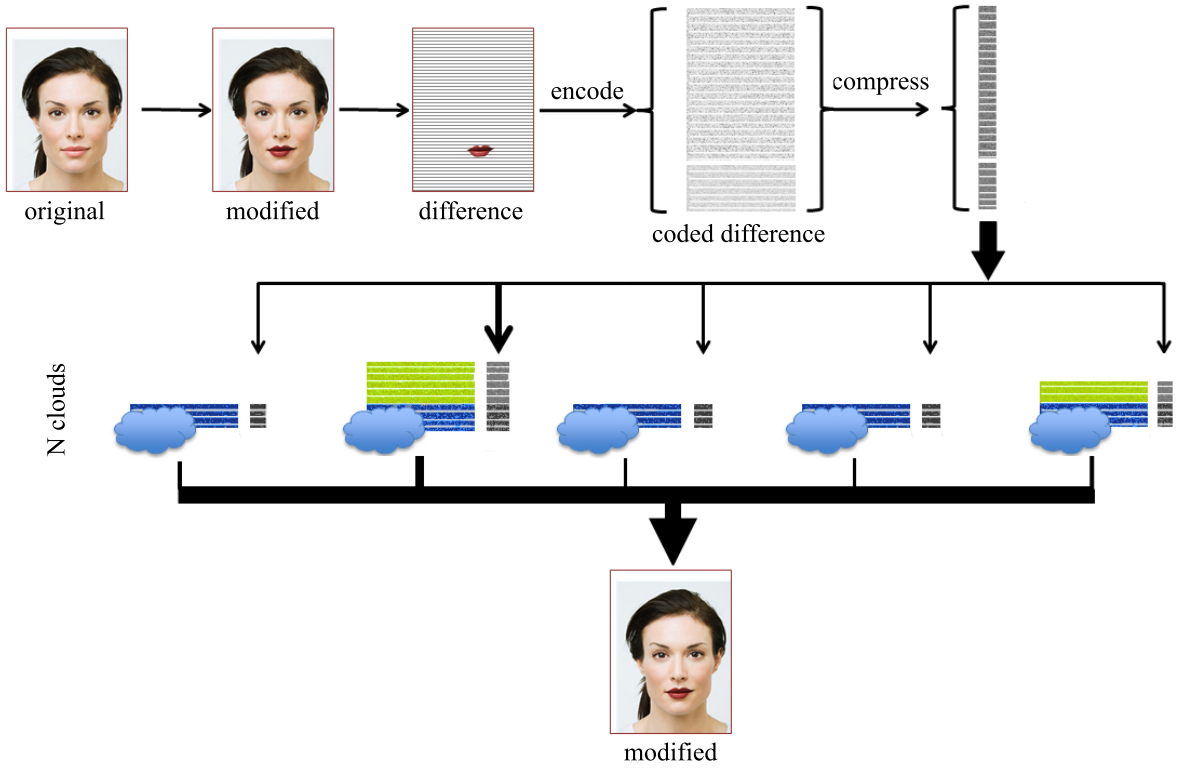


Figure 5.1: Example of updating an erasure–coded bitmap. The difference between the original and modified bitmap files is calculated, encoded and then uploaded to the storage clouds in a compressed form. This can either be stored for later use on the nodes as part of version control system or applied directly on the encoded data. During retrieval, the modified file is recreated after decoding.

the *encoded packets* and the *encoded difference packets* and apply them before decoding. The result is the modified version of the file in both cases. If several encoded differences are stored, the client can choose which version of the file to retrieve, making for a simple but effective version control system.

Decompressing the encoded difference only happens right before its application. This ensures that it is always in a compressed encoded form when stored or sent over the network, minimizing the use of network and storage resources.

Crucially, the technique works with all linear block codes without requiring changes to the encoding and decoding mechanisms. It can be applied transparently to both systematic and non–systematic codes and does not require distinguishing between systematic and non–systematic packets.

5.3 A theoretical model of updating encoded data

To update a file to a new version, we have identified three types of changes on its elements that we wish to support. These cover all types of changes that can be made to the content of a file apart from deleting the entire file.

Definition 5.1 (Modification)**.** A modification is a change in the value of one or more elements of a file*.

Definition 5.2 (Deletion)**.** A deletion is the removal of one or more elements of a file from a specific location.

Definition 5.3 (Insertion)**.** An insertion is the addition of one or more elements into a specific locations in the file.

This section provides solutions supported by examples to all three types of changes and introduces most of the notation, summarized in Table 5.1. In the examples, calculations are performed over GF(3) with elements $\{0, -, +\} \in$ GF(3) and the summation and product operations shown in (5.1). Most practical solutions use a non–prime field in the form of GF(2^m), where m is typically 1, 2, 4, 8 or 16. We decided on using GF(3) for our examples because its small size makes it easier for readers to follow. While GF(2) seemed like a better choice in this regard, illustrating deletions would have been difficult. Since our algorithms only use summation and product operations, our proposed solution works over any finite field.

$\oplus$	$-$	0	$+$
$-$	$+$	$-$	0
0	$-$	0	$+$
$+$	0	$+$	$-$

$\otimes$	$-$	0	$+$
$-$	$+$	0	$-$
0	0	0	0
$+$	$-$	0	$+$

(5.1)

We can use GF(3) with the aforementioned operations in the examples because:

- it is closed under both addition and multiplication:
  $\forall a, b \in GF(3) : a \oplus b \in GF(3), a \otimes b \in GF(3)$
- associativity of addition and multiplication:
  $\forall a, b, c \in GF(3) : (a \oplus b) \oplus c = a \oplus (b \oplus c),\ (a \otimes b) \otimes c = a \otimes (b \oplus c)$
- commutativity of addition and multiplication:
  $\forall a, b \in GF(3) : a \oplus b = b \oplus a,\ a \otimes b = b \otimes a$

*Modifying elements could also be achieved through a deletion and an insertion. However, it would require more operations and generally be less storage and network efficient.

- 0 is the additive identity element:
  $\forall a \in GF(3) : a \oplus 0 = a$
- all elements have an additive inverse:
  $\forall a \in GF(3) : \exists \ominus a, a \ominus (\ominus a) = 0$
- $+$ is the multiplicative identity element:
  $\forall a \in GF(3) : a \otimes + = a$
- all elements except 0 have a multiplicative inverse:
  $\forall a \in GF(3), a \neq 0 : \exists a^{-1}, a \otimes a^{-1} = +$
- distributivity of multiplication over addition:
  $\forall a, b, c \in GF(3) : a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$

Let us first look at the operation of modifying some elements in a file through a slightly simplified example, starting from the initial creation of the file and finishing with the retrieval of a modified version from a storage node.

We organize the original data into matrix $\mathbf{M}$ by dividing it into 4 fragments ($k = 4$) with 3 symbols each ($pac\_len = 3$). The encoding matrix $\mathbf{V}$ is defined by the erasure code and is not modified by our solution in order to ensure compatibility with all linear block codes. Each of its n rows is associated with data in an erasure coded packet. Normally, additional encoded packets would be generated depending on the desired redundancy using an encoding matrix with additional rows, i.e. $n > k$. To recover the data, at least one of the $k \times k$ submatrices of $\mathbf{V}$ associated with available encoded data must be invertible. We use $n = k$ to keep the example simple without losing generality. Let us use matrices generated using random elements from $GF(3) \setminus \{0\}$[†] and $GF(3)$ for $\mathbf{M}$ and $\mathbf{V}$ respectively:

$$\mathbf{M} := \begin{pmatrix} - & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \qquad \mathbf{V} := \begin{pmatrix} + & - & + & - \\ - & + & + & 0 \\ 0 & + & - & 0 \\ + & - & + & 0 \end{pmatrix} \tag{5.2}$$

Initially, the client encodes four packets that are distributed to the storage nodes.

$$\mathbf{X} = \mathbf{V} \cdot \mathbf{M} = \begin{pmatrix} + & - & - \\ 0 & + & + \\ 0 & + & - \\ - & + & 0 \end{pmatrix} \tag{5.3}$$

[†]We set aside the 0 value as it will have a special significance in signaling deleted elements later on. A more practical solution that does note set aside values is described in Section 5.5.2.

Table 5.1: Table of Notations

		Matrices
$\mathbf{M}$	–	the original data
$\mathbf{M}'$	–	the modified data
$\hat{\mathbf{M}}$	–	difference between the original and the modified data: $\hat{\mathbf{M}} = \mathbf{M}' - \mathbf{M}$
$\mathbf{V}$	–	the encoding matrix containing the coefficient vectors
$\mathbf{X}$	–	the original encoded data
$\mathbf{X}'$	–	the modified encoded data
$\hat{\mathbf{X}}$	–	the encoded difference between the original and the modified data: $\hat{\mathbf{X}} = \mathbf{X}' - \mathbf{X}$
		Row vectors and single values
del	–	vector of deleted indices
ins	–	vector of elements to be inserted
ind	–	the index of the first element to be inserted into the original data
n	–	the number of encoded packets – the number of rows of $\mathbf{M}$
pac_len	–	the length of encoded packets in symbols – the number of columns of $\mathbf{M}$
		Modified matrices
$\mathbf{A}^t$	–	the transformed version of matrix $\mathbf{A}$
$\mathbf{A}_r$	–	the remapped version of matrix $\mathbf{A}$
$\tilde{\mathbf{A}}$	–	a version of matrix $\mathbf{A}$ that contains some invalid elements
$\mathbf{A}^+$	–	a version of matrix $\mathbf{A}$ that contains some extra columns
$\mathbf{A}^-$	–	a version of matrix $\mathbf{A}$ that has had a sequence of its elements removed

The data is later updated on the client side $\mathbf{M} \rightarrow \mathbf{M}'$, the change is marked with () in (5.4).

$$\mathbf{M}' := \begin{pmatrix} (+) & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \tag{5.4}$$

The client then calculates the difference $\hat{\mathbf{M}}$ between the original and the modified file.

Definition 5.4 (Unencoded difference)**.** $\hat{\mathbf{M}}$ is a matrix representing the difference between two versions of a file, defined as $\hat{\mathbf{M}} := \mathbf{M}' - \mathbf{M}$.

$$\hat{\mathbf{M}} = \mathbf{M}' - \mathbf{M}$$
$$\begin{pmatrix} - & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} = \begin{pmatrix} (+) & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} - \begin{pmatrix} - & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \tag{5.5}$$

This can then be used to generate the encoded updates for the four original encoded packets by taking the product of $\hat{\mathbf{M}}$ and the original coefficients, which remain unchanged.

Definition 5.5 (Encoded difference)**.** $\hat{\mathbf{X}}$ is a matrix representing the difference between two versions of a file in an erasure–coded form, defined as $\hat{\mathbf{X}} := \mathbf{V} \cdot \hat{\mathbf{M}}$.

$$\hat{\mathbf{X}} = \mathbf{V} \cdot \hat{\mathbf{M}} = \begin{pmatrix} - & 0 & 0 \\ + & 0 & 0 \\ 0 & 0 & 0 \\ - & 0 & 0 \end{pmatrix} \tag{5.6}$$

Each row of $\hat{\mathbf{X}}$ corresponds to an encoded difference packet and is sent to the storage node that has the data associated with the coefficients in that row of $\mathbf{V}$. If the node can perform simple computations, it can apply the difference once it receives it using matrix addition. Otherwise, this operation is performed on the client during data retrieval[‡]:

$$\mathbf{X}' = \mathbf{X} + \hat{\mathbf{X}}$$
$$\begin{pmatrix} 0 & - & - \\ + & + & + \\ 0 & + & - \\ + & + & 0 \end{pmatrix} = \begin{pmatrix} + & - & - \\ 0 & + & + \\ 0 & + & - \\ - & + & 0 \end{pmatrix} + \begin{pmatrix} - & 0 & 0 \\ + & 0 & 0 \\ 0 & 0 & 0 \\ - & 0 & 0 \end{pmatrix} \tag{5.7}$$

The resulting data can be decoded seamlessly to recover the updated file:

$$\mathbf{M}' = \mathbf{V}^{-1} \cdot \mathbf{X}' = \begin{pmatrix} (+) & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \tag{5.8}$$

[‡]Another option that is not covered in this work is to transmit a compressed version of $\hat{\mathbf{M}}$ to each storage node and calculate parts of $\hat{\mathbf{X}}$ there.

Proposition 5.6. *The changed version ($\mathbf{M'}$) of a file can be recovered by decoding ($\mathbf{M'} = \mathbf{V^{-1}} \cdot \mathbf{X'}$) the encoded representation of the change ($\mathbf{X'}$), where $\mathbf{V}$ is the invertible encoding matrix associated with the linear block code used to encode the original file and the unencoded difference.*

Proof.

$$\begin{aligned} \mathbf{M'} &= \mathbf{V^{-1}} \cdot \mathbf{X'} \\ \mathbf{M'} &= \mathbf{V^{-1}} \cdot (\mathbf{X} + \hat{\mathbf{X}}) \\ \mathbf{M'} &= \mathbf{V^{-1}} \cdot \mathbf{V} \cdot \mathbf{M} + \mathbf{V^{-1}} \cdot \mathbf{V} \cdot \hat{\mathbf{M}} \\ \mathbf{M'} &= \mathbf{M} + \hat{\mathbf{M}} \end{aligned} \tag{5.9}$$

□

5.3.1 Representing bursty modifications in an encoded form

Most changes in textual data occur in bursts. Words, sentences, paragraphs, lines of code are more likely to be modified, added and deleted together. Using a natural approach of representing data, the cost of changing each element is multiplied by the number of packets once encoded. In fact, a single change that is at least *pac_len* long is propagated to all elements of all packets, making the encoded change as large as the file itself.

Definition 5.7 (Bursty change)**.** A bursty change is a modification, deletion or addition that affects several subsequent elements of the stored version of the data.

Throughout this chapter, we focus on reducing the storage required to represent bursty changes of length *len* in erasure-coded data. Let us first look at why the location of changes in a file is important once erasure coding is applied, again through an example.

Let $\hat{\mathbf{M}}$ be

$$\hat{\mathbf{M}} := \begin{pmatrix} 0 & (-) & (+) \\ (-) & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}. \tag{5.10}$$

Once the change is encoded with the same $\mathbf{V}$ as defined previously, the resulting matrix becomes dense, even though the length of the change was only 3:

$$\hat{\mathbf{X}} = \mathbf{V} \cdot \hat{\mathbf{M}} = \begin{pmatrix} + & - & + \\ - & + & - \\ - & 0 & 0 \\ + & 0 & + \end{pmatrix}. \tag{5.11}$$

Let us measure the density of a matrix by counting the number of elements that are potentially[§] non–zero. We use this simple metric to express how well the data in the rows of the matrix can be compressed in a lossless manner. A metric that also expresses the distribution of non–zero elements would have carried more information on this aspect, but is unnecessary for our purposes thanks to the well–determined position of potentially non–zero elements.

Let $nz(\mathbf{A})$ denote the number of non–zero elements of matrix $\mathbf{A}$ and let $\kappa(len) := nz(\hat{\mathbf{X}})$ after encoding a change of len consequent elements in $\mathbf{M}'$ using a completely dense[¶] encoding matrix to get $\hat{\mathbf{X}}$.

Definition 5.8 (Overhead of encoded updates)**.** $O(len)$ is the storage and equivalent network overhead of representing a bursty change of length len in an encoded form compared to a non–encoded form, defined as $O(len) := \frac{\kappa(len)}{len} - 1$.

Using the natural representation of data as in the example, we can determine $O_{\text{nat}}(len)$ based on how elements are multiplied and added together during encoding. The overhead shown on Equation (5.12) is irrespective of the location of the first changed element. $\kappa_{\text{nat}}(len)$ only depends on the number of columns of $\hat{M}$ with at least one non–zero element and that is irrespective of where the bursty change is located.

$$O_{\text{nat}}(len) = \frac{\min(pac\_len, len) \cdot n}{len} - 1 \tag{5.12}$$

This is maximized by $len = pac\_len$, where $O_{\text{nat}}(pac\_len) = n-1$. With $len \geq pac\_len$, the number of non–zero elements in the encoded change packets reaches the size of the original encoded packets, making storing each version of the file as costly as storing multiple copies of the file. Clearly, such a high storage cost is unacceptable and would make storing changes in an encoded form unfeasible.

We propose transforming the data before any operations are executed on it by laying it out column by column rather then row by row as shown on Figure 5.2. Thus, bursts of changes would now be in columns of $\mathbf{M}'$, impacting far fewer columns of $\hat{\mathbf{X}}$ after encoding and reducing $\kappa_{\text{trans}}(len)$.

[§]0 is an element of the field and thus may occur naturally in the encoded data. However, when generalizing over a family of matrices, we are interested in the number of elements which can take values other than zero.

[¶]This can be considered a worst–case scenario, since a sparse encoding matrix would potentially create more 0 elements in $\hat{\mathbf{X}}$. Thus, for example, we expect systematic codes to be able to achieve a slightly lower overhead using appropriate compression algorithms.

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix} \qquad \begin{pmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{pmatrix}$$

(a) Natural representation (b) Transformed representation

Figure 5.2: Example on how data may be represented to increase storage efficiency.

For example, let us look at the same changes as before, but in a transformed representation:

$$\hat{\mathbf{M}} = \begin{pmatrix} 0 & 0 & 0 \\ (-) & 0 & 0 \\ (+) & 0 & 0 \\ (-) & 0 & 0 \end{pmatrix}. \tag{5.13}$$

After encoding $\hat{\mathbf{M}}$, only the first column in $\hat{\mathbf{X}}$ is affected, all other elements are zero:

$$\hat{\mathbf{X}} = \mathbf{V} \cdot \hat{\mathbf{M}} = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ + & 0 & 0 \\ - & 0 & 0 \end{pmatrix}. \tag{5.14}$$

The number of affected columns varies based on the location of the change. For example, if the changes had occurred one or two positions later, both the first and second columns would have been affected. Equation (5.15) shows the expected value of the cost of updating *len* consequent elements, derived from the expected number of columns of $\hat{\mathbf{X}}$ that will potentially contain non–zero elements.

$$E[\kappa_{\text{transf}}(len)] = \left(\left\lfloor \frac{len}{n} \right\rfloor + 1 + \frac{(len \bmod n) - 1}{n} \right) n \tag{5.15}$$

Besides the location of the change, the cost is determined by how many columns the changes span in $\mathbf{M}'$. At least $\lfloor \frac{len}{n} \rfloor$ columns are needed to hold the modified elements. Depending on the location of the change, it may overflow into the following column. The expected value of the overflow is given by the second and third term of the addition. Finally, n is the number of elements in a column.

$$E[O_{\text{transf}}(len)] = \frac{E[\kappa_{\text{transf}}(len)]}{len} - 1 \tag{5.16}$$

Theorem 5.9. *The expected overhead of representing a coded bursty change is at most* $\frac{2n-2}{len}$.

Proof. Let us simplify (5.16) by using upper estimates for $\lfloor\ \rfloor$ and mod operations.

$$\begin{aligned} E[O_{\text{transf}}(len)] &= \frac{(\lfloor \frac{len}{n} \rfloor + 1 + \frac{(len \bmod n)-1}{n})n}{len} - 1 \\ E[O_{\text{transf}}(len)] &\leq \frac{(\frac{len}{n} + 1 + \frac{n-2}{n})n}{len} - 1 \\ E[O_{\text{transf}}(len)] &\leq \frac{(len + 2n - 2)}{len} - 1 \\ E[O_{\text{transf}}(len)] &\leq \frac{2n-2}{len} \end{aligned} \tag{5.17}$$

□

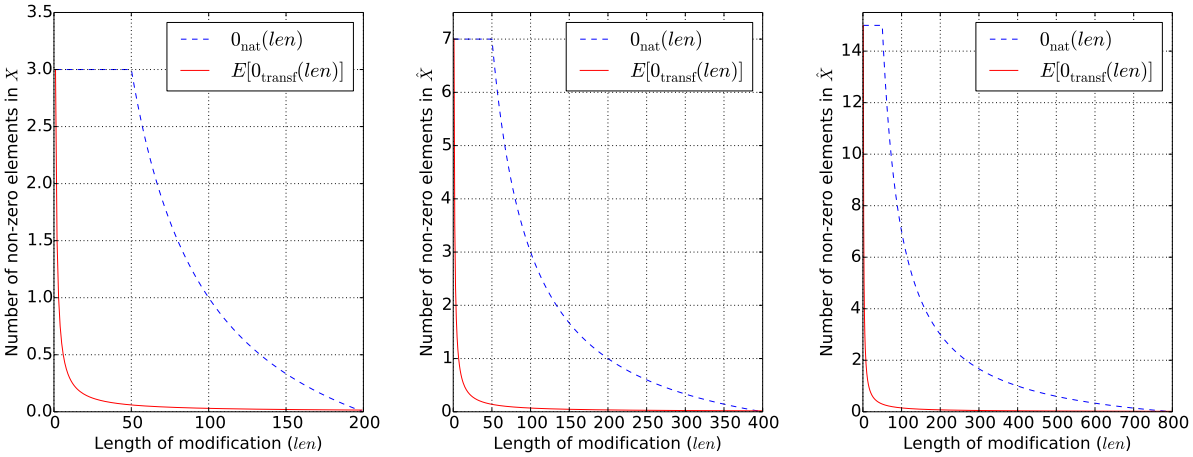
Corollary 5.10. *The expected overhead of representing a coded bursty change tends to* 0 *as the length of the change tends towards the length of the file and the length of a packet tends towards infinity.*

Proof. The length of the file is $pac\_len \cdot n$, thus:

$$\lim_{\substack{len \to pac\_len \cdot n \\ pac\_len \to \infty}} \frac{2n-2}{len} = \frac{2}{pac\_len} - \frac{2}{pac\_len \cdot n} = 0 \tag{5.18}$$

□

Figure 5.3 compares the overheads of the transformed and the natural representation of bursty changes of varying length. The transformed representation is significantly more storage efficient, and its overhead becomes negligible as *len* increases. We do not expect the transformed representation to bring benefits for changes that are random in nature in terms of their location in the original file, i.e. non–bursty changes.



(a) $n = 4$ (b) $n = 8$ (c) $n = 16$

Figure 5.3: The overhead of different representations of data ($pac\_len = 50$).

5.3.2 Deleting elements

The second important change with regard to updating files is the deletion of individual elements. We have chosen to trace this back to the previously presented operation of modifying elements. Thus, the overhead of encoded updates is identical to that presented in Equations (5.12) and (5.16). We propose signaling deletions in $\mathbf{M}'$ by using a special symbol. However, in order to be able to rely on the same basic operations of matrix addition, substraction, multiplication and inversion, the special symbol must be a part of the finite field we use to represent data. This requires either setting aside an element of the field and ensuring that it does not occur in $\mathbf{M}$ or representing data using a higher field. We argue for the first option as it does not require changing decoding and encoding operations, and makes it compatible with erasure codes that are tailored to a particular field. We describe a solution for this problem in Section 5.5.5 and analyze the overhead it introduces.

Here we show how deletions work through an example, using 0 as the signaling symbol and deleting the first element of the file. Let $\mathbf{M}$ and $\mathbf{M}'$ have the following values:

$$\mathbf{M} := \begin{pmatrix} - & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \qquad \mathbf{M}' := \begin{pmatrix} (0) & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \tag{5.19}$$

The encoded difference can be calculated the same way as previously:

$$\hat{\mathbf{X}} = \mathbf{V} \cdot (\mathbf{M}' - \mathbf{M})$$
$$\begin{pmatrix} + & 0 & 0 \\ - & 0 & 0 \\ 0 & 0 & 0 \\ + & 0 & 0 \end{pmatrix} = \begin{pmatrix} + & - & + & - \\ - & + & + & 0 \\ 0 & + & - & 0 \\ + & - & + & 0 \end{pmatrix} \cdot \begin{pmatrix} + & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{5.20}$$

Based on Proposition 5.6, we can recover the updated file by decoding the stored coded difference after adding it to $\mathbf{X}$. At this point, the client can simply remove elements from the recovered $\mathbf{M}'$ that have been marked with the deleted symbol to get the modified version of the file. The size of the data remains unchanged until the last moment, making deletions fully transparent to encoding and decoding operations.

5.3.3 Adding elements

Adding elements can be done in the same general way. However, unlike the previous two changes, it changes the size and shape of the data. We avoided this for deletions by only

removing the elements at the last moment, after data has successfully been recovered through decoding. This is not possible for additions if we want to keep using the same simple matrix operations. To make sure that data lines up with the original version, we pad some of the matrices with the special deletion symbol. Thus, this operation also has the requirement that the deletion symbol is not present in $\mathbf{M}$ or in the inserted elements. Having previously shown the benefits of using a transformed data representation, we focus solely on this.

Let us again showcase this operation through an example. Let $\mathbf{M}$ and $\mathbf{M}'$ have the following values:

$$\mathbf{M} := \begin{pmatrix} - & - & - \\ + & - & + \\ + & + & - \\ + & - & + \end{pmatrix} \qquad \mathbf{M}' := \begin{pmatrix} - & - & (0) & - \\ + & (+) & - & + \\ + & (0) & + & - \\ + & (0) & - & + \end{pmatrix} \tag{5.21}$$

Equation (5.21) shows how an element was added into the second position in the second column of $\mathbf{M}'$ followed by deletion symbols as padding up to a multiple of n. The padding ensures that elements that are not in the affected columns are not shifted from their original rows, reducing $\kappa_{\text{trans}}(len)$. To be able to calculate the unencoded difference between the two versions of the file, $\mathbf{M}$ must also have $\lceil \frac{len}{n} \rceil n$ deletion symbols inserted as columns following the insertion. We denote the padded matrix with $\mathbf{M}^+$ and calculate $\hat{\mathbf{M}}$ as previously.

$$\hat{\mathbf{M}} = \mathbf{M}' - \mathbf{M}^+$$
$$\begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & - & - & 0 \\ 0 & - & + & 0 \\ 0 & + & - & 0 \end{pmatrix} = \begin{pmatrix} - & - & (0) & - \\ + & (+) & - & + \\ + & (0) & + & - \\ + & (0) & - & + \end{pmatrix} - \begin{pmatrix} - & - & 0 & - \\ + & - & 0 & + \\ + & + & 0 & - \\ + & - & 0 & + \end{pmatrix} \tag{5.22}$$

Encoding the difference is performed in the same way and is not affected by the change in the size of *pac_len*, apart from the increase in the number of operations. The same $\mathbf{V}$ can be used as n has not changed.

$$\hat{\mathbf{X}} = \mathbf{V} \cdot \hat{\mathbf{M}}$$
$$\begin{pmatrix} 0 & - & 0 & 0 \\ 0 & + & 0 & 0 \\ 0 & 0 & - & 0 \\ 0 & 0 & + & 0 \end{pmatrix} = \begin{pmatrix} + & - & + & - \\ - & + & + & 0 \\ 0 & + & - & 0 \\ + & - & + & 0 \end{pmatrix} \cdot \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & - & - & 0 \\ 0 & - & + & 0 \\ 0 & + & - & 0 \end{pmatrix} \tag{5.23}$$

Before $\hat{\mathbf{X}}$ can be added to $\mathbf{X}$ to apply the update, it must also be padded in the same way $\mathbf{M}$ was, as shown on Equation (5.24).

$$\begin{gathered}\mathbf{X}' = \mathbf{X}^+ + \hat{\mathbf{X}} \\ \begin{pmatrix} + & + & 0 & - \\ 0 & - & 0 & + \\ 0 & + & - & - \\ - & + & + & 0 \end{pmatrix} = \begin{pmatrix} + & - & 0 & - \\ 0 & + & 0 & + \\ 0 & + & 0 & - \\ - & + & 0 & 0 \end{pmatrix} + \begin{pmatrix} 0 & - & 0 & 0 \\ 0 & + & 0 & 0 \\ 0 & 0 & - & 0 \\ 0 & 0 & + & 0 \end{pmatrix}\end{gathered} \tag{5.24}$$

Based on Proposition 5.6, we can recover the updated file by decoding $\mathbf{X}'$ as in the case of the previous two operations. Finally, the deletion symbols introduced as padding can be removed from the recovered $\mathbf{M}'$.

A transformed representation of data benefits bursty additions as well. Compared to the previous operations, the extra column adds a fixed n to the formula in Equation (5.15).

$$E[\kappa_{\text{transf}}(len)] = \left(\left\lfloor \frac{len}{n} \right\rfloor + 1 + \frac{(len \bmod n) - 1}{n} \right) n + n \tag{5.25}$$

Corollary 5.11. *The expected overhead of representing a coded bursty insertion is at most* $\frac{3n-2}{len}$.

Proof.

$$\begin{aligned} E[O_{\text{nat}}(len)] &= \frac{(\lfloor \frac{len}{n} \rfloor + 2 + \frac{(len \bmod n)-1}{n})n}{len} - 1 \\ E[O_{\text{nat}}(len)] &\leq \frac{(\frac{len}{n} + 2 + \frac{n-2}{n})n}{len} - 1 \\ E[O_{\text{nat}}(len)] &\leq \frac{(len + 3n - 2)}{len} - 1 \\ E[O_{\text{nat}}(len)] &\leq \frac{3n-2}{len} \end{aligned} \tag{5.26}$$

□

Corollary 5.12. *The expected overhead of representing a coded bursty insertion tends to* 0 *as the length of the insertion tends towards the length of the file and the length of a packet tends towards infinity.*

Proof. The length of the file is $pac\_len \cdot n$.

$$\lim_{\substack{len \to pac\_len \cdot n \\ pac\_len \to \infty}} \frac{3n-2}{len} = \frac{3}{pac\_len} - \frac{2}{pac\_len \cdot n} = 0 \tag{5.27}$$

□

5.4 Proposed solution – implementation

So far we have included a formal description of methods for handling three types of changes that may occur in files, complemented by examples. These translate relatively well into practice with a few exceptions. This section presents an overview of how the methods can be implemented, broken down into individual steps. This is followed in the next section by a more detailed algorithmic presentation of the individual steps.

The first difference between the proposed mathematical model and the implementation arises from the fact that files cannot always be divided into rows of equal length to form a matrix, therefore most files will have an incomplete last row. In a purely mathematical model, this can easily be overcome by padding the file with 0s, however this would be wasteful in practice. Because of this, all algorithms that we present deal with non–padded files. We keep the notation consistent and use non–bold letters (M, M', V, X, etc.) to distinguish data structures that cannot be considered matrices but correspond to the previously defined $\mathbf{M}, \mathbf{M}', \mathbf{V}, \mathbf{X}$, etc..

The second difference also stems from the desire to make the operations more efficient. We store and transmit the encoded differences using a compressed form. To make the representation of changes as sparse as possible and thus enable effective compression, we transform the way data is represented before operating on it. This requires some metadata regarding the changes be transmitted and stored alongside the data.

Thirdly, signaling deletions requires a special data symbol that is not present in the original data. Computers typically use fixed sized variables to represent any one of a fix number of symbols. For example, one byte can be used to represent at most 256 different symbols. Using an alternate representation of data that changes the number of representable symbols would greatly reduce storage efficiency and may break the correctness of mathematical operations. Thus, a workaround is needed that does not require specialized, and thus unoptimized operations and has relatively low overhead.

Finally, maintaining compatibility between operations is important to enhance the practical usability of our algorithms for version control as it is necessary to enable different types of modifications to be aggregated. This intention, along with the interactions between the different steps adds significant complexity.

5.4.1 Modifying elements

This operation involves modifying any number of individual elements in a location in the original data and is shown on Figure 5.4. It does not involve deleting or adding new elements, therefore the size and the structure of the data remains unchanged.

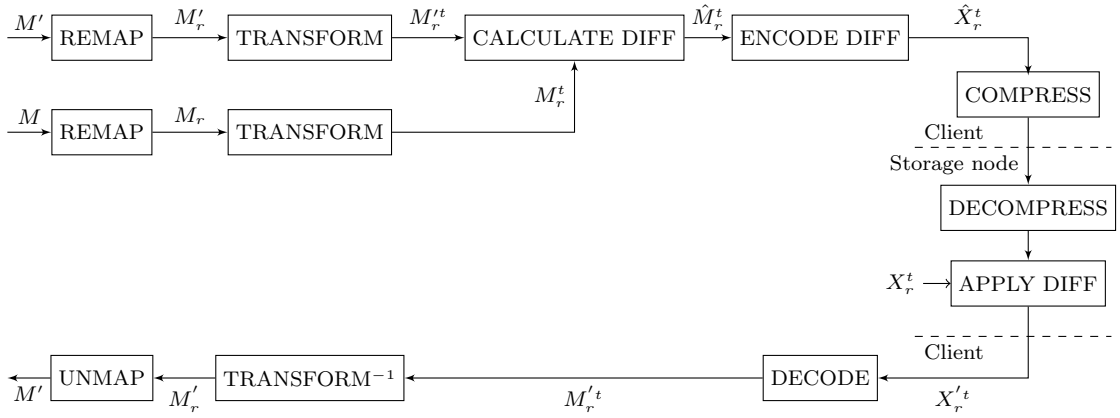


Figure 5.4: Overview of modifying individual elements

First, both the original (M) and the changed (M') data is remapped to enable signaling deletions[‖]. Both are then transformed to minimize the impact of bursty modifications on the encoded difference ($\hat{X}$). Following this, the difference between the two matrices is calculated and encoded. This is then compressed and sent over the network to the storage node. The storage node can store it as a separate version of the file (not shown on the figure) or decompress it and apply it. If the storage node does not support computations on the data, only the former is possible. In this case, the encoded difference and the original encoded data are transferred to the client when the file is requested and the application of the difference takes place here. This is followed in both cases by the decoding of the modified data. Finally, the reverse of the transformation and mapping processes are performed to recover the modified file (M').

5.4.2 Deleting elements

Deleting is the process of removing an arbitrary number of elements from an arbitrary location in the original data. The basic idea is to signal deletions with a special value (we use 0 without loss of generality) in M' and then perform the modification operation. However, implementation-wise there are aspects that make it differ slightly from this. A general overview is shown on Figure 5.5.

[‖]While this process is not strictly necessary for this operation, we have included it to maintain compatibility between the three types of changes.

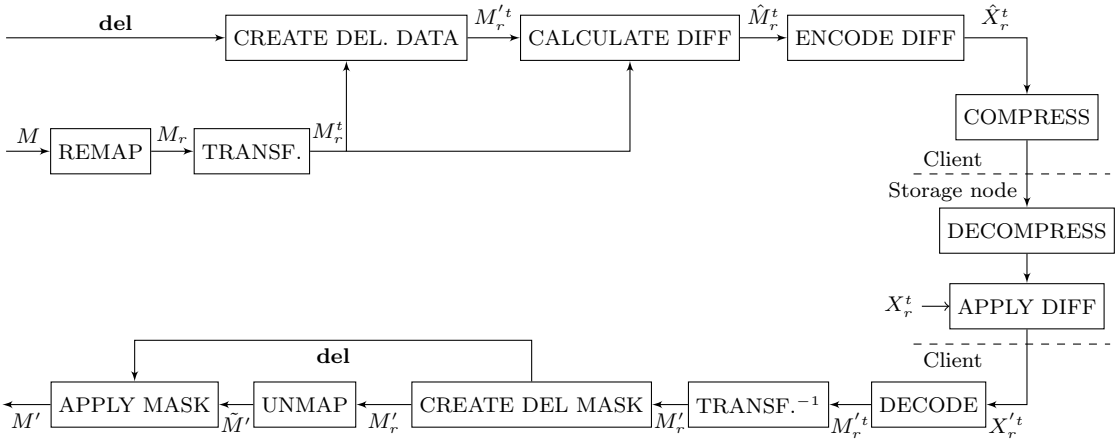


Figure 5.5: Overview of deleting elements

First, the original data is remapped and transformed. Based on this and a vector of deleted indices **del**, the changed data $M_r^{t'}$ is calculated. This is then used to calculate the difference. It is possible to calculate the difference without creating the deleted data by only modifying the removed elements individually in $\hat{M}_r^t$, as all other values are 0. Encoding and applying the difference and decoding are identical to the previous operation. Following this, the data is transformed back. A mask **del** (a vector of the indices of the deleted elements) of the deleted elements is created. This is applied after unmapping to actually remove the deleted elements. The elements could also be removed before unmapping is performed. However, this would require a more complex unmapping process that takes into account the amount of removed elements in each part of the remapped data.

5.4.3 Adding elements

Adding data is the process of inserting an arbitrary number of elements into the original data. It differs significantly from both previously presented operations as it increases the size of the data. A general overview is shown on Figure 5.6. The basic idea is to insert 0 values where the new data is supposed to go into the original data in the form of new columns before calculating the difference. The length of the inserted elements and the index of the first element are stored as separate metadata. Most of the complexity of the algorithms is given by the need to align data in M and M' in such a way, that the inserted data only affects localized parts of $\hat{M}$ and subsequently $\hat{X}$.

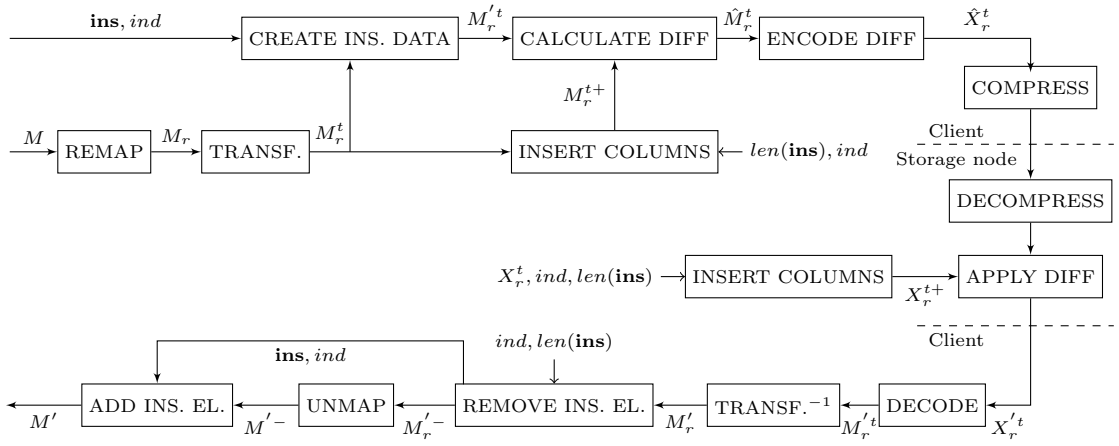


Figure 5.6: Overview of adding elements

The first step is to remap and transform the original data. Based on this, along with the vector of inserted elements **ins**, and the index of the first element in M, the changed data $M_r^{t'}$ is calculated. To calculate the difference, columns filled with 0 values must first be inserted into the required place in the original data. Encoding, applying the differences and decoding is identical to the other two operations. Before the difference can be applied, columns filled with 0 values must first be inserted into the stored encoded data as well. Finally, after the reverse of the transformation is applied to the changed data, the inserted elements must temporarily be removed for unmapping to take place and then reinserted. Similarly to the previous operation, the final three steps can be merged. This should reduce computational complexity but makes them more difficult to implement.

5.5 Detailed description of algorithms

5.5.1 Creating, applying, encoding and decoding the difference between two versions of a file

Since most erasure codes of practical interest work over finite fields of type GF(2^n), we focus our discussion on these. The difference between two vectors can be calculated by performing XOR operations on their elements individually.

Likewise, applying the difference on the stored version of the file in order to update it can also be done using Algorithm 5.1 with the stored encoded file and encoded change as input.

The encoding of linear block codes is typically performed by multiplying the data matrix and the encoding matrix : $\hat{X} = \hat{M} \cdot V$. Decoding is performed after the difference is applied and in general can be done by multiplying the encoded data with the inverse of the encoding matrix: $M' = X' \cdot V^{-1}$.

Algorithm 5.1 Creating the difference between two vectors

```
1: procedure CALCULATE DIFF(original, changed)
2:     for i < len(original) do
3:         difference[i] ← original[i] XOR changed[i]
4:     end for
5:     return difference
6: end procedure
```

5.5.2 Remapping data to enable signaling deletions

To be able to mark elements as deleted in M', we have arbitrarily chosen the element 0 to use as the signaling value. However, before it becomes suitable for this purpose, we must ensure it is not present in either M or M'. The remapping operation (shown on Figure 5.7) solves this by removing all 0 values from a vector and substituting them with other values that are not present in a certain part of the data. Let q denote the size of the finite field in use by the encode and decode operations. The original vector is divided into blocks, the maximum amount of data in each block is the amount of values that can be represented by an element minus 1, i.e. $q - 1$. For example, if we work with bytes, blocks are at most $2^8 - 1 = 255$ long. This guarantees that there will be at least one value between 0 and 255 in each block that is not present. This unused value becomes the remapping element in the remapped data for that block and will store the value that was used to substitute 0 values with.

Algorithm 5.2 Remapping a vector over GF(q)

```
1: procedure REMAP(original)
2:     for each q − 1 long block of original do
3:         unused_sym ← find a symbol between 0..q − 1, which ∉ block
4:         result[block_begin + block_ctr · q] ← unused_sym
5:         for i ← 0 to len(block) do
6:             if block[i] = 0 then
7:                 result[block_begin + block_ctr · q + i + 1] ← unused_sym
8:             else
9:                 result[block_begin + block_ctr · q + i + 1] ←
10:                original[block_begin + i]
11:            end if
12:        end for
13:        block_ctr ← block_ctr + 1
14:    end for
15:    return result
16: end procedure
```

Unmapping is the reverse of the mapping operation. It takes a vector of elements and substitutes the 0 values back into it based on the signaling elements.

Algorithm 5.3 Unmapping a vector over GF(q)

```
1: procedure UNMAP(remapped)
2:     for each q long block of remapped do
3:         remapped_symbol ← remapped[block_begin]   ▷ retrieve the remapped symbol
4:         for i ← 1 to len(block) do                ▷ unmap the remapped values to 0
5:             if block[i] = remapped_symbol then
6:                 result[block_begin + i − block_ctr − 1] ← 0
7:             else
8:                 result[block_begin + i − block_ctr − 1] ←
9:                 remapped[block_begin + i]
10:            end if
11:        end for
12:        block_ctr ← block_ctr + 1
13:    end for
14:    return result
15: end procedure
```

$$
\begin{aligned}
M &= (\quad m_1 \quad m_2 \quad \cdots \quad 0 \quad \cdots \quad m_{q-1} \quad |\cdots) \\
&\downarrow \\
M_r &= (\quad \boldsymbol{\alpha} \quad m_1 \quad m_2 \quad \cdots \quad \boldsymbol{\alpha} \quad \cdots \quad m_{q-1} \quad |\cdots)
\end{aligned}
$$

Figure 5.7: Remapping the data to replace 0 values with unused values over GF(q).

$$
\begin{aligned}
M'_r &= \Big(\quad \boldsymbol{\alpha} \quad m'_1 \quad m'_2 \quad \cdots \quad \boldsymbol{\alpha} \quad \cdots \quad \boldsymbol{\alpha} \quad \cdots \quad m'_{q-1} \quad | \quad \cdots\Big) \\
&\downarrow \\
M' &= \Big(\quad m'_1 \quad m'_2 \quad \cdots \quad 0 \quad \cdots \quad 0 \quad \cdots \quad m'_{q-1} \quad | \quad \cdots\Big)
\end{aligned}
$$

Figure 5.8: Unmapping the data to reinsert 0 values over GF(q).

The remapping element introduces a small overhead that depends on the size of the finite field in use.

Definition 5.13 (Remapping overhead)**.** O_{remap} is the ratio of the additional elements inserted to signal deletions to the length of the original file, defined as $O_{\text{remap}} = \frac{\left\lfloor \frac{file\_len}{q} \right\rfloor + \left(\frac{file\_len}{q} - \left\lfloor \frac{file\_len}{q} \right\rfloor \right)}{file\_len}$.

All blocks apart from the last one have 1 remapping element for q-1 data elements (the last block will have fewer data elements if $(q-1) \nmid \dim(M)$). For larger files, the overhead is approximately $\frac{1}{q}$, as the impact of the last block diminishes:

$$\lim_{file\_len \to \infty} OP_{\text{remap}} = \frac{1}{q} \tag{5.28}$$

Thus, for moderate–sized fields such as GF(2^8), the overhead is almost negligible. On the other hand, it is overwhelming for small fields such as GF(2), making our proposed solution unpractical for these cases.

5.5.3 Transforming the data

All of the three proposed operations can be performed without this step, however we have found that most modifications in text files generally occur grouped together in chunks. To use this property for creating a more efficient solution, we propose transforming the order of elements in a vector to make them align in a particular way. The goal is to be able to represent changes that are grouped together more efficiently after applying an erasure code. Each file is represented as a potentially incomplete 2D array, where the number of rows is the number of coded packets and the number of columns is the size of each packet. It is potentially incomplete as the last row might be shorter than others. The transformation is identical in concept to transposing a square matrix, but slightly more complicated due to the data matrix generally not being a square matrix and the potential of an incomplete last row. It places elements that were close to each other in the original array in the same column or in adjacent columns in the transformed array. This ensures that changes made to chunks of the original data (M') only affect the encoded difference ($\hat{X}$) in a localized manner. The operation is illustrated on Figure 5.9 and defined in detail as Algorithm 5.4. The reverse operation can be performed using Algorithm 5.5.

Algorithm 5.4 Transforming a 2D array

```
1:  procedure TRANSFORM(original, pac_len)
2:      current_element ← 0                          ▷ the index of the data to be copied
3:      lrl ← len(original) mod pac_len              ▷ length of the last row
4:      for i ← 0 to len(original) do
5:          result[i] ← original[current_element]
6:          if (i + 1) mod pac_len > lrl then
7:              offset ← 1
8:          else
9:              offset ← 0
10:         end if
11:         current_element ← current_element + offset
12:         if (i + 1) mod pac_len = 0 then
13:             current_element ← (i + 1)/pac_len
14:         end if
15:     end for
16:     return transformed
17: end procedure
```

$$M_r = \begin{pmatrix} m_1 & m_2 & \cdots & \cdots & m_s \\ m_{s+1} & m_{s+2} & \cdots & \cdots & m_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{(n-2)\cdot s+1} & \cdots & \cdots & \cdots & m_{(n-1)\cdot s} \\ \underbrace{m_{l-lrl} \quad \cdots \quad m_l}_{\text{last row length } (lrl)} & & & & \end{pmatrix} \longrightarrow M_r^t = \begin{pmatrix} m_1 & m_{n+1} & \cdots & \cdots & m_{l-n+1} \\ m_2 & m_{n+2} & \cdots & \cdots & \vdots \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ m_{n-1} & \cdots & \cdots & \cdots & m_l \\ \underbrace{m_n \quad \cdots \quad m_{n\cdot lrl}}_{lrl} & & & & \end{pmatrix}$$

l – size of the file, s – *pac_len*

Figure 5.9: Transforming the way data is represented to increase the effectiveness of compression.

Algorithm 5.5 Transforming back a 2D array

```
1: procedure TRANSFORM^{-1}(transformed, pac_len)
2:     current_element ← 0
3:     n ← ceil(len(transformed)/pac_len)
4:     last_row_length ← len(transformed) mod pac_len
5:     for i ← 0 to len(original) do
6:         result[current_element] ← transformed[i]
7:         if (i + 1) mod pac_len > lastRowLength then
8:             offset ← 1
9:         else
10:            offset ← 0
11:        end if
12:        current_element ← current_element + n − offset
13:        if current_element ≥ len(transformed) then
14:            current_element ← (i + 1)/pac_len
15:        end if
16:    end for
17:    return transformed
18: end procedure
```

5.5.4 Compressing the difference

Important to the effectiveness of our proposed solution is the creation of highly–compressible encoded differences. Compression should be applied after the difference is calculated, just before it is sent over the network to the storage node. Decompression should always proceed applying the difference. This can be performed either on the storage node or on the client that retrieves the file. In either case, these rules ensure that the minimal amount of data is transferred over the network and stored.

Compression methods make up an entire field of information theory and certain techniques work better for certain types of data. However, generally, sparse data can be stored using significantly fewer symbols than dense data. Thus, our techniques aim to increase the number of 0s in the encoded difference. To evaluate how well they performed in this regard, we didn't actually use compression in our experiments. Instead, we sent

and stored only the elements in the changed columns, along with the index of the first affected column. The overhead of this metadata can be avoided for modifications and deletions, but not for adding elements due to the positioning of the padding.

5.5.5 Algorithms used for deleting elements

Algorithm 5.6 creates the array that stores the changed information to support the delete operation based on the list of deleted indices as shown on Figure 5.10. It requires that the input (M) array be previously remapped to ensure that it does not contain the 0 value.

$$M = \begin{pmatrix} m_1 & m_2 & \cdots & \boxed{m_{d1}} & \cdots & \boxed{m_{d2}} & \cdots \end{pmatrix}$$

$$\downarrow$$

$$M' = \begin{pmatrix} m'_1 & m'_2 & \cdots & 0 & \cdots & 0 & \cdots \end{pmatrix}$$

$\boxed{m'_i}$ - elements marked for deletion

Figure 5.10: Creating the modified version of that has had some elements removed.

Algorithm 5.6 Create deleted data

```
1: procedure CREATE DEL. DATA(original, del)
2:     for i ← 0 to len(original) do
3:         if adjust_index(i) ∈ del then
4:             result[i] ← 0
5:         else
6:             result[i] ← original[i]
7:         end if
8:     end for
9:     return result
10: end procedure
```

Algorithm 5.7 returns the indices of all 0 values in an array.

Algorithm 5.8 removes values from an array based on a list of deleted indices, as shown on Figure 5.11.

$$\tilde{M}' = \begin{pmatrix} m'_1 & \cdots & m'_a & 0 & m'_{a+1} & \cdots & 0 & \cdots & m'_n \end{pmatrix}$$

$$\downarrow$$

$$M' = \begin{pmatrix} m'_1 & \cdots & m'_a & m'_{a+1} & \cdots & 0 & \cdots & m'_n \end{pmatrix}$$

Figure 5.11: Removing elements based on a list of deleted indices.

Algorithm 5.7 Create deleted mask

```
1: procedure CREATE DEL. MASK(original)
2:     delete_count ← 0
3:     for i ← 0 to len(original) do
4:         if original[i] = 0 then
5:             result[delete_count] ← i                ▷ save the deleted index
6:             delete_count ← delete_count + 1
7:         end if
8:     end for
9:     return result
10: end procedure
```

Algorithm 5.8 Apply deleted mask

```
1: procedure APPLY MASK(original, del)
2:     delete_count ← 0
3:     for i ← 0 to len(original) do
4:         if adjust_index(i) ∉ del then
5:             result[i − delete_count] ← original[i]   ▷ copy non-deleted items
6:             delete_count ← delete_count + 1
7:         end if
8:     end for
9:     return result
10: end procedure
```

5.5.6 Algorithms used for adding elements

Algorithm 5.9 adds padding in the form of 0 values to the array that will be inserted to make its length a multiple of n. This ensures that the data stays aligned after the insert operation. The padded array is then copied into the array that holds the original data, as presented on Figure 5.12.

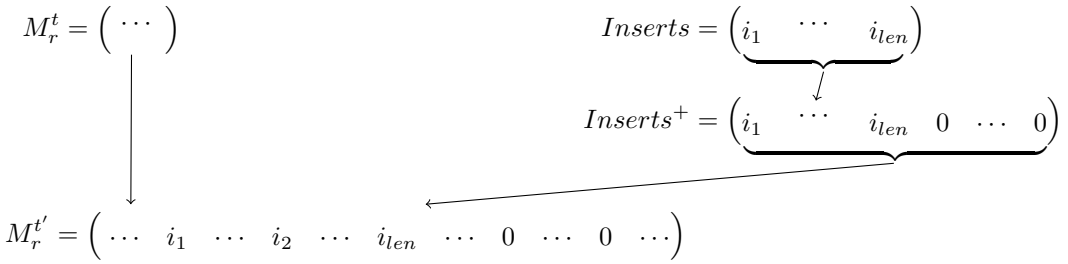


Figure 5.12: Creating the modified version of the data that includes newly inserted elements.

Algorithm 5.10 appends columns into the original array as shown on Figure 5.13. It also copies elements that are to be overwritten into the last newly inserted column.

Algorithm 5.9 Create inserted data

```
1: procedure CREATE INS. DATA(original, ins, ind, n)
2:     for i ← len(ins) to len(ins) + n − (len(ins) mod n) do
3:         ins[i] ← 0                                  ▷ pad the ins array with 0s to make
4:     end for                                         ▷ its length a multiple of n
5:     for i ← 0 to ind do
6:         result[adjust_idx(i)] ← original[adjust_idx(i)]
7:     end for
8:     for i ← ind to len(ins) do                      ▷ Insert content
9:         result[adjust_idx(i)] ← ins[i]
10:    end for
11:    for i ← ind + len(ins) to len(original) + len(ins) do
12:        result[adjust_idx(i)] ← original[adjust_idx(i + len(ins))]
13:    end for
14:    return result
15: end procedure
```

Algorithm 5.10 Create inserted columns

```
1: procedure INSERT COLUMNS(original, len(ins), ind, n)
2:     Calculate number of columns_to_insert
3:     for i ← 0 to ind do
4:         result[adjust_index(i)] ← original[adjust_index(i)]
5:     end for
6:     for i ← 0 to columns_to_insert · n do
7:         result[adjust_index(ind + i)] ← 0
8:     end for
9:     overwritten_length ← n − (ind mod n)
10:    for i ← 0 to overwritten_length do
11:        result[adjust_index(ind + columns_to_insert · n + i)]
12:        ← original[adjust_index(ind + i + overwritten_length)]
13:    end for
14:    return result
15: end procedure
```

Algorithm 5.11 temporarily removes elements from an array and saves the removed elements in a separate array. As shown on Figure 5.14, it does not copy 0 values which were added as padding.

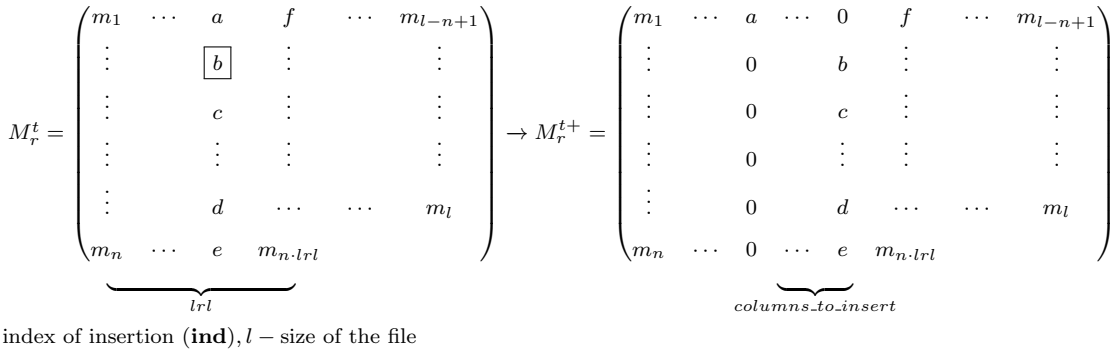


Figure 5.13: Appending the original version of the data to include new columns. Shows the case in which the number of the column where the insertion starts is smaller than the last row's length.

Algorithm 5.11 Remove inserted elements

```
1: procedure REMOVE INS. ELEMENTS(original, ind, length, n)
2:     j ← 0
3:     padding ← length mod n
4:     for i ← 0 to len(original) do
5:         if i ≥ ind and i < ind + length + padding then
6:             if < ind + length then
7:                 inserted[j] ← original[i]                ▷ inserted data
8:             end if
9:             j ← j + 1
10:        else
11:            result[i − j] ← original[i]                  ▷ original data
12:        end if
13:    end for
14:    return result, inserted
15: end procedure
```

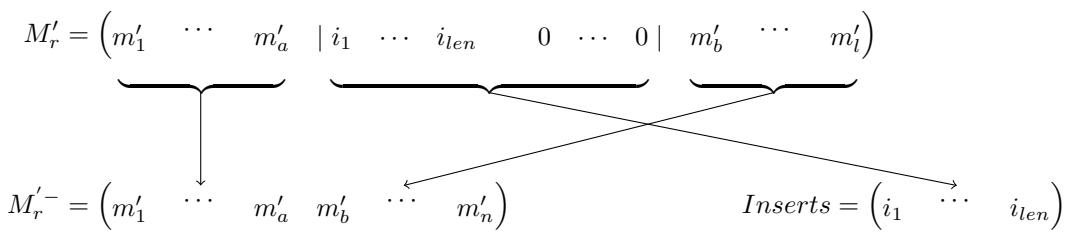


Figure 5.14: Temporarily removing inserted elements.

Finally, Algorithm 5.12 inserts elements into an array. It is different from Algorithm 5.9 as it works with unmapped, untransformed data.

Algorithm 5.12 Add inserted elements

```
1: procedure ADD INS. ELEMENTS(original, ind, ins)
2:     for i ← 0 to ind do
3:         result[i] ← original[i]
4:     end for
5:     for i ← ind to ind + len(ins) do
6:         result[i] ← ins[i − ind]
7:     end for
8:     for i ← ind + len(ins) to len(original) + len(ins) do
9:         result[i] ← original[i − len(ins)]
10:    end for
11:    return result, ins
12: end procedure
```

5.6 Experiments on a version control system

5.6.1 Measuring storage/transmission overhead

We evaluated the effectiveness of our proposed solution using data from the publicly available Git repository [Steinwurf ApS, 2014] of the *Storage Benchmarks* project. The project has the goal of comparing the coding throughput of some of the most popular publicly available erasure correcting libraries. We chose to use a Git repository to work with data from a real–world scenario. As Git stores all individual modifications of files line by line, it is possible to recreate any version from the original to the most recent. The project contains 261 files out of which 29 undergo significant modifications. We restricted our evaluation to these, with a total of 1749 modified lines. The rest mostly had a single version.

We examined commits on the master branch that occurred between February 12th and December 1st 2014 using the following command:

```
$ git log -p --reverse --full-diff
```

We parsed the output and saved each modified line along with its offset in the file. We considered each modification as a separate version of the file. Git stores differences on a line by line basis: when data is modified, the containing line is deleted and re–added in its modified form. This is potentially wasteful if only a few elements are modified and something that our proposed solution can avoid by only storing individual modified elements instead of lines. We did not include this optimization to give a fairer comparison

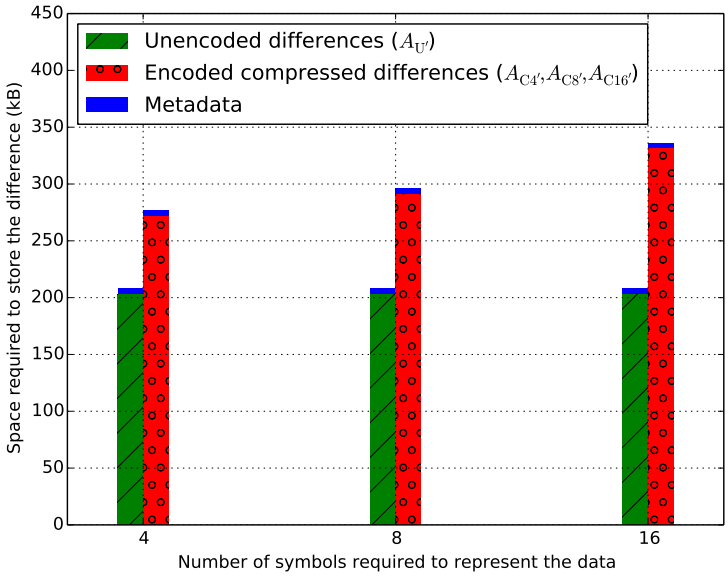


Figure 5.15: Storage and network traffic overhead of our proposed solution compared to a non–encoded representation.

to Git's unencoded way of storing differences between versions. Thus, we only used two of our supported operations: adding and deleting elements. Besides the actual modifications, we also created a metadata file roughly 5 kB in size that stored the type of operation, the offset of the modification in the previous version and the length of the modification. We measured the size of modifications, as output by the *git log* command instead of the files created by Git, since it stores a significant amount of metadata for each change including date, modifying user and file permissions in the same files.

We compared the amount of data necessary to represent the original files and the modifications using three different values for the number of encoded n: 4, 8 and 16. We restricted ourselves to $n = k$ to have a more direct comparison with non–encoded solutions. For systems that include redundancy, results scale according to the rate of the code as expected. All results are aggregated across the entire project. To store all versions of the files, $A_{\mathrm{U}} = 79.887$ MB needed to be transmitted across the network in one direction and stored. This is how much a current typical state of the art storage system needs to transfer to update encoded data. Next, we calculated the differences and encoded them using RLNC over GF(2^8). This required $A_{\mathrm{C4}} = 80.290$ MB, $A_{\mathrm{C8}} = 80.670$ MB and $A_{\mathrm{C16}} = 82.087$ MB for values of n of 4, 8 and 16 respectively (including the encoding matrix V). However, thanks to the transformation step, most of the encoded data was very sparse as the modifications aligned and only affected a few columns of X' and consequently $\hat{X}$. By removing the coefficients from the data (which are already available on the storage nodes along with the original version of the file) as well as the non–modified portions

containing only 0 values, the storage cost is reduced by 5 orders of magnitude to $A_{C4'} = 271.6$ kB, $A_{C8'} = 291.2$ kB and $A_{C16'} = 330.9$ kB. The effectiveness of our solution results less from choosing the appropriate compression algorithm and more from creating a highly compressible representation of differences by aligning data. Therefore, conventional compression algorithms that do not have knowledge of the metadata should also perform well.

We compared our solution to systems which store data in a non–encoded fashion and found the efficiency of our proposed solution is only slightly worse. Storing only the difference in a non–encoded form using the line by line difference approach employed by Git requires $A_{U'} = 202.7$ kB. Figure 5.15 compares this value to our solution using several values for the number of encoded packets n. The extra cost in storage and network traffic is relatively small compared to the benefits of being able to erasure–code modifications in a file. Smaller values of n achieve a slightly smaller compressed representation. This can be explained by looking at Equations (5.15) and (5.25): the penalty in the overhead associated with changes overflowing into subsequent columns of $\hat{X}$ decreases with decreasing values for the multiplication factor n.

5.6.2 Some practical considerations

Our solution also has a computational overhead compared to state of the art systems that do not encode differences. It is a natural consequence of using erasure codes. Fortunately, calculating and applying encoded and unencoded differences, remapping, unmapping, transformations and reverse transformations are all linear operations in the size of the data. The same is true for algorithms specific to deleting and adding elements. We chose not to go into greater detail as we do not claim that our solution is optimal in terms of minimizing the number of operation. We have broken down the three main operations into as many small steps as we could to increase clarity. When implementing a system with our proposed solution, we suggest looking at merging some of these steps to reduce computational complexity.

A large part of the computational overhead can be attributed to encoding and decoding operations as they are both $\mathcal{O}(n^3)$. However, our solution creates very sparse differences, therefore the practical number of operations for encoding the difference should be much lower than for the original encoding.

There is a trade–off therefore between the storage and network overhead on one side, and the computational overhead on the other. Larger values of n decrease storage and network overhead and increase the flexibility of the data distribution in some cases. On the other hand, they increase the number of operations that are performed during encoding and decoding. Real–world performance is dependent on a great deal of things besides the

number of operations and significant advances have been made by erasure coding libraries in recent years [Steven Max Paterson, 2014].

Finally, when our technique is applied to a version control system, a large number of versions and associated metadata will be created over time. If this causes issues, older versions can be archived by merging them together. This reduces the amount of metadata stored and may reduce overall storage use as well if the merged versions modify the same parts of a file.

5.7 Conclusion

We have presented a detailed mechanism to support updating files when using erasure correcting codes for storage applications, including any random linear network coded system. These ideas can be applied to standard data center systems as well as novel cloud storage and peer–to–peer distributed storage systems that are keen on supporting various versions of the same file without high storage costs or for efficient updates without the inherent costs of uploading full files to all storage nodes. It is, to our knowledge, the first solution that tackles this problem and the first to be designed around the idea of being transparent to the erasure code used. We have shown through theoretical results as well as real–world measurements that the transmission and storage overhead it introduces is relatively low for bursty changes.

Significant challenges still lie ahead in terms of making the encoded representation of non–bursty changes compressible as well as finding compression algorithms that work well for these cases. Furthermore, while our solution supports signaling deletions over small finite fields as well, it is inefficient for these cases.

Nevertheless, we hope the work presented in this chapter is a significant step towards making erasure coding systems better suited to storing mutable data and the creation of commercially available erasure–coded version control systems.

Chapter 6

Conclusions and future perspectives

We have shown that many of the drawbacks associated with erasure coding in general can be alleviated by using network coding judiciously. While we tried to identify the key challenges and suggest solutions, a lot remains to be done. It is difficult to know what new challenges the future holds for erasure–coded storage. Nevertheless, we would like to close this dissertation with a few ideas on how our contributions might be applied in future systems. Some of these use cases are already observable today, but most might only become relevant in 5 to 10 years.

In Chapter 2 we proposed a very generic solution to making erasure codes network–aware. We believe the technique addresses a very current problem and therefore carries potential benefits for today's systems. Unfortunately, it requires a relatively deep knowledge of network bottlenecks and traffic schemes. To make it more accessible, the creation of a mechanism that maintains costs automatically based on a predefined goal would be beneficial. It is hard to foresee how data center networks will evolve in coming years, but they will most likely remain dynamic, ever–changing entities and erasure codes that possess these same properties will no doubt remain well suited for this scenario.

Chapter 3 dealt with aggregated cloud storage systems. Several commercial applications follow this basic idea, yet few if any use the techniques we proposed. Thus, our research could form the basis of a network–coded commercial product that offers increased reliability, retrieval performance and security. Several open questions remain related to our adaptive approach, mainly the rate and measure of interventions. A solution based on constructs from control theory would be ideal for this purpose. Furthermore, an information theoretical analysis of how much random linear network coding adds to security in conjunction with conventional methods would help convince users to move more of their data to the cloud.

The balance of storage space, network transfers and computing resources greatly influences both the need for and the feasibility of using erasure coding. It is hard to predict how

it will evolve with time and will likely differ across scenarios. At least for mobile devices, the limited spectrum and the amount of effort placed in improving battery technology makes it likely that network costs will continue to outweigh computing costs. Thus, our results from Chapter 4 can be used in a variety of future applications. As cars become smarter and start communicating with each other and the surrounding infrastructure, it makes sense to keep data that has local relevance, such as current road conditions on-board or in small microservers close to its physical origin. Thus, it can be retrieved and updated with much lower latency and makes the system more robust by not having to rely on a failure–prone central service or continuous network connectivity to a cell tower. This scenario also carries with it one of the biggest obstacles faced by fog computing, the continuous variation in the nodes of the system. Many expect swarms of drones or other robots to be tasked with several tasks such as mapping terrain, observing natural phenomena, solving sanitation issues and so forth. Like self–driving cars, much of the data has local relevance and is shared by the nodes.

Our contributions from Chapter 5 are perhaps our most generic and widely applicable. However, practical systems that implement the algorithms should be fairly unbalanced in the sense that computing resources must be readily available while network traffic and storage space should be at a premium. We showcased our ideas through a version control system and hope that they can be used for other purposes as well. The development of compression techniques that work for less bursty modifications and engineering techniques that cache and merge updates would help in this regard.

Scientific publications

Journal papers

1. **Sipos, M.**, Gahm, J., Venkat, N., and Oran, D. (2018) Network-aware repair for erasure coded distributed storage. IEEE/ACM Transactions on Networking (Early access)

2. **Sipos, M.**, Braun, P. J., Lucani, D. E., Fitzek, F. H. P., and Charaf, H. (2017). On the Effectiveness of Recoding-based Repair in Network Coded Distributed Storage. Periodica Polytechnica Electrical Engineering and Computer Science.

3. **Sipos, M.**, Heide, J., Lucani, D., Pedersen, M., Fitzek, F., and Charaf, H. (2015). Adaptive Network Coded Clouds: High Speed Downloads and Cost-Effective Version Control. IEEE Transactions on Cloud Computing, PP(99):1-1.

Conference papers

4. **Sipos, M.**, Gahm, J., Venkat, N., and Oran, D. (2016). Erasure Coded Storage on a Changing Network: The Untold Story. In 2016 IEEE Global Communications Conference (GLOBECOM), pages 1-6. Washington DC, USA

5. **Sipos, M.**, Fitzek, F. H. P., and Lucani, D. E. (2015). On the feasibility of a network coded mobile storage cloud. In 2015 IEEE International Conference on Communications (ICC), pages 466-471. London, United Kingdom

6. **Sipos, M.**, Fitzek, F. H. P., and Lucani, D. E. (2015). Random Linear Network Coding Is Key to Data Survival in Highly Dynamic Distributed Storage. In 2015 IEEE 81st Vehicular Technology Conference (VTC Spring), pages 1-6. Glasgow, Scotland

7. **Sipos, M.** (2015). A sparse representation of changes in an erasure coded versioning system In Proceedings of the Automation and Applied Computer Science Workshop (AACS), Pages 1-12. Budapest, Hungary

8. **Sipos, M.**, Fitzek, F. H. P., Lucani, D. E., and Pedersen, M. V. (2014). Distributed cloud storage using network coding. In 2014 IEEE 11th Consumer Communications and Networking Conference (CCNC), pages 127-132. Las Vegas, USA

9. **Sipos, M.** (2014). Dynamic Data Distribution in the Clouds. In Proceedings of the Automation and Applied Computer Science Workshop (AACS), Pages 1-12. Budapest, Hungary

10. **Sipos, M.**, Fitzek, F., Lucani, D., and Pedersen, M. (2014). Dynamic allocation and efficient distribution of data among multiple clouds using network coding. In Cloud Networking (CloudNet), 2014 IEEE 3rd Inter national Conference on, pages 90-95. Luxemburg, Luxembourg

11. Fitzek, F. H. P., Toth, T., Szabados, A., Pedersen, M. V., Lucani, D. E., **Sipos, M.**, Charaf, H., and Medard, M. (2014). Implementation and performance evaluation of distributed cloud storage solutions using random linear network coding. In 2014 IEEE International Conference on Communications Workshops (ICC), pages 249-254. Sidney, Australia

12. **Sipos, M.** (2013). Predicting the Availability of Mobile Nodes in Distributed Storage Systems. In Proceedings of the Automation and Applied Computer Science Workshop (AACS). pages 1-13. Budapest, Hungary

13. **Sipos, M.** and Ekler, P. (2013). Predicting Availability of Mobile Peers in Large Peer-to-Peer Networks. In 2013 3rd Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC), pages 71-77. Budapest, Hungary

Other conference papers

14. Braun, P. J., **Sipos, M.**, Ekler, P., and Fitzek, F. H. (2017). On the Performance Boost for Peer to Peer WebRTC-based Video Streaming with Network Coding. In IEEE International Conference on Communications, ICC 2017. Paris, France

15. Braun, P. J. and **Sipos, M.** (2015). Reducing linear dependencies in network coding assisted BitTorrent networks. In Proceedings of the Automation and Applied Computer Science Workshop (AACS). Budapest, Hungary

16. Braun, P. J., **Sipos, M.**, Ekler, P., and Charaf, H. (2015). Increasing data distribution in BitTorrent networks by using network coding techniques. In Proceedings of

European Wireless 2015; 21th European Wireless Conference, pages 1-6. Budapest, Hungary

17. **Sipos, M.** (2012). Guidelines to the Development of Content Management Systems Using the B Method. In Proceedings of the Automation and Applied Computer Science Workshop 2012 (AACS), pages. 81-92 Budapest, Hungary

Patent applications

18. **Sipos, M.**, Venkat, N., Gahm, J., and Oran, D.(2016). Network-aware storage repairs, US filing, pending international filing, Cambridge, USA

19. **Sipos, M.**, Venkat, N., Gahm, J., and Apostolopoulos, J. (2016). Efficient repair of erasure coded data based on coefficient matrix decomposition, US filing, pending international filing, Cambridge, USA

20. **Sipos, M.**, Lucani, D., Heide, J., Pedersen, M., and Fitzek, F. (2015). Method for file updating and version control for linear erasure coded and network coded storage, US and international filing, Aalborg, Denmark

List of Figures

Bibliography

[Abdrashitov and Medard, 2015] Abdrashitov, V. and Medard, M. (2015). Durable network coded distributed storage. In *2015 53rd Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 851–856.

[Ahlswede et al., 2000] Ahlswede, R., Cai, N., Li, S. Y. R., and Yeung, R. W. (2000). Network information flow. *IEEE Transactions on Information Theory*, 46(4):1204–1216.

[Akhlaghi et al., 2010] Akhlaghi, S., Kiani, A., and Ghanavati, M. R. (2010). A fundamental trade-off between the download cost and repair bandwidth in distributed storage systems. In *2010 IEEE International Symposium on Network Coding (NetCod)*, pages 1–6.

[Alder, 1969] Alder, H. (1969). Partition identities – from Euler to the present. *The American Mathematical Monthly*, 76(7):733–746.

[Ananthanarayanan et al., 2013] Ananthanarayanan, G., Ghodsi, A., Shenker, S., and Stoica, I. (2013). Effective straggler mitigation: Attack of the clones. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 185–198, Lombard, IL. USENIX.

[Andras Bori, 2014] Andras Bori, P. E. (2014). Google play store. https://play.google.com/store/apps/details?id=hu.bute.daai.amorg.drtorrent, visited 2014-07-01.

[Bal and Kaashoek, 1993] Bal, H. E. and Kaashoek, M. F. (1993). Object distribution in orca using compile-time and run-time techniques. *SIGPLAN Not.*, 28(10):162–177.

[Bell, 2005] Bell, J. (2005). Euler and the pentagonal number theorem. *arXiv:math/0510054*. arXiv: math/0510054.

[Benet, 2014] Benet, J. (2014). IPFS-content addressed, versioned, P2p file system. *arXiv preprint arXiv:1407.3561*.

[Bonomi et al., 2014] Bonomi, F., Milito, R., Natarajan, P., and Zhu, J. (2014). Fog Computing: A Platform for Internet of Things and Analytics. In Bessis, N. and Dobre, C., editors, *Big Data and Internet of Things: A Roadmap for Smart Environments*,

pages 169–186. Springer International Publishing, Cham. DOI: 10.1007/978-3-319-05029-4_7.

[Bonomi et al., 2012] Bonomi, F., Milito, R., Zhu, J., and Addepalli, S. (2012). Fog computing and its role in the internet of things. In *Proceedings of the First Edition of the MCC Workshop on Mobile Cloud Computing*, MCC 2012, pages 13–16, New York, NY, USA. ACM.

[Chacon, 2009] Chacon, S. (2009). *Pro Git*. Apress, Berkeley, CA, USA, 1st edition.

[Corporation, 2017] Corporation, V. (2017). Number of concurrent connected users on the Steam gaming platform. http://store.steampowered.com/stats/, visited 2017-02-24.

[Csorba et al., 2013] Csorba, K., Ekler, P., Bori, A., and Charaf, H. (2013). Analysis of mobile bittorrent client behavior. In *Cognitive Infocommunications (CogInfoCom), 2013 IEEE 4th International Conference on*, pages 613–618.

[Deb et al., 2005] Deb, S., Choute, C., Médard, M., and Koetter, R. (2005). Data harvesting: A random coding approach to rapid dissemination and efficient storage of data. *preprint*.

[Dimakis et al., 2007] Dimakis, A. G., Godfrey, P. B., Wainwright, M. J., and Ramchandran, K. (2007). Network coding for distributed storage systems. In *IEEE International Conference on Computer Communications*.

[Dimakis et al., 2010] Dimakis, A. G., Godfrey, P. B., Wu, Y., Wainwright, M. J., and Ramchandran, K. (2010). Network coding for distributed storage systems. *IEEE Transactions on Information Theory*, 56(9):4539–4551.

[Dimakis et al., 2011] Dimakis, A. G., Ramchandran, K., Wu, Y., and Suh, C. (2011). A survey on network codes for distributed storage. *Proceedings of the IEEE*, 99(3):476–489.

[Dinur and Steurer, 2014] Dinur, I. and Steurer, D. (2014). Analytical approach to parallel repetition. In *Proceedings of the 46th Annual ACM Symposium on Theory of Computing*, STOC '14, pages 624–633, New York, NY, USA. ACM.

[Esmaili et al., 2013] Esmaili, K., Chiniah, A., and Datta, A. (2013). Efficient updates in cross-object erasure-coded storage systems. In *Big Data, 2013 IEEE International Conference on*, pages 28–32.

[Feher and Forstner, 2011] Feher, M. and Forstner, B. (2011). Identifying and utilizing routines of human movement. In *2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems*, pages 135–138.

[Fitzek and Katz, 2014] Fitzek, F. and Katz, M. (2014). *Mobile Clouds: Exploiting Distributed Resources in Wireless, Mobile and Social Networks*. John Wiley & Sons Ltd.

[Ford et al., 2010] Ford, D., Labelle, F., Popovici, F., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. (2010). Availability in globally distributed storage systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*.

[Forlines et al., 2012] Forlines, C., Miller, S., Prakash, S., and Irvine, J. (2012). Heuristics for improving forecast aggregation. In *AAAI Fall Symposium*.

[Freund and Mason, 1999] Freund, Y. and Mason, L. (1999). The alternating decision tree learning algorithm. In *In Machine Learning: Proceedings of the Sixteenth International Conference*, pages 124–133. Morgan Kaufmann.

[Gartner, 2012] Gartner (2012). Gartner says that consumers will store more than a third of their digital content in the cloud by 2016. http://www.gartner.com/newsroom/id/2060215, visited 2016-09-02.

[Gastón et al., 2013] Gastón, B., Pujol, J., and Villanueva, M. (2013). A realistic distributed storage system that minimizes data storage and repair bandwidth. *arXiv:1301.1549 [cs, math]*.

[Guruswami and Wootters, 2015] Guruswami, V. and Wootters, M. (2015). Repairing reed-solomon codes. *CoRR*, abs/1509.04764.

[Heide et al., 2011a] Heide, J., Pedersen, M., Fitzek, F., and Medard, M. (2011a). On code parameters and coding vector representation for practical rlnc. In *Communications (ICC), 2011 IEEE International Conference on*, pages 1–5.

[Heide et al., 2011b] Heide, J., Pedersen, M. V., Fitzek, F. H. P., and Medard, M. (2011b). On code parameters and coding vector representation for practical rlnc. In *2011 IEEE International Conference on Communications (ICC)*, pages 1–5.

[Ho et al., 2006] Ho, T., Medard, M., Koetter, R., Karger, D. R., Effros, M., Shi, J., and Leong, B. (2006). A random linear network coding approach to multicast. *IEEE Transactions on Information Theory*, 52(10):4413–4430.

[Hosmer and Lemeshow, 2000] Hosmer, D. W. and Lemeshow, S. (2000). *Applied logistic regression (Wiley Series in probability and statistics)*. Wiley-Interscience Publication, 2 edition.

[Hu et al., 2012a] Hu, Y., Chen, H. C., Lee, P. P., and Tang, Y. (2012a). NCCloud: applying network coding for the storage repair in a cloud-of-clouds. In *Proceedings of the 10th USENIX Conference on File and Storage Technologies (FAST '12) (Short paper track), San Jose, CA, February 2012.*

[Hu et al., 2012b] Hu, Y., Lee, P. P. C., and Shum, K. W. (2012b). Analysis and construction of functional regenerating codes with uncoded repair for distributed storage systems. *CoRR*, abs/1208.2787.

[Hu et al., 2016] Hu, Y., Lee, P. P. C., and Zhang, X. (2016). Double regenerating codes for hierarchical data centers. In *2016 IEEE International Symposium on Information Theory (ISIT)*, pages 245–249.

[Hu et al., 2015] Hu, Y. C., Patel, M., Sabella, D., Sprecher, N., and Young, V. (2015). Mobile edge computing – A key technology towards 5g. *ETSI White Paper*, 11.

[Huang et al., 2012a] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. (2012a). Erasure Coding in Windows Azure Storage. pages 15–26.

[Huang et al., 2012b] Huang, C., Simitci, H., Xu, Y., Ogus, A., Calder, B., Gopalan, P., Li, J., and Yekhanin, S. (2012b). Erasure coding in Windows Azure Storage. In *Proceedings of the 2012 USENIX Conference on Annual Technical Conference*, USENIX ATC'12, pages 2–2, Berkeley, CA, USA. USENIX Association.

[Huang and others, 2011] Huang, D. and others (2011). Mobile cloud computing. *IEEE COMSOC Multimedia Communications Technical Committee (MMTC) E-Letter*, 6(10):27–31.

[Hunt et al., 1996] Hunt, J. J., Vo, K.-P., and Tichy, W. F. (1996). An Empirical Study of Delta Algorithms. In *Proceedings of the SCM-6 Workshop on System Configuration Management*, ICSE '96, pages 49–66, London, UK, UK. Springer-Verlag.

[Intel, 2017] Intel (2017). Intelligent storage acceleration library. https://software.intel.com/en-us/storage/ISA-L, visited 2017-03-09.

[Jaggi et al., 2006] Jaggi, S., Cassuto, Y., and Effros, M. (2006). Low Complexity Encoding for Network Codes. In *2006 IEEE International Symposium on Information Theory*, pages 40–44.

[Joshi, 2016] Joshi, G. (2016). *Efficient Redundancy Techniques to Reduce Delay in Cloud Systems*. PhD thesis, Massachusetts Institute of Technology.

[Khan and Chatzigeorgiou, 2016] Khan, A. and Chatzigeorgiou, I. (2016). Improved bounds on the decoding failure probability of network coding over multi-source multi-relay networks. *IEEE Communications Letters*.

[Korn and Vo, 2002] Korn, D. G. and Vo, K.-P. (2002). Engineering a Differencing and Compression Data Format. In *USENIX annual technical conference, general track*, pages 219–228.

[Ladoczki et al., 2015] Ladoczki, B., Fernandez, C., Moya, O., Babarczi, P., Tapolcai, J., and Guija, D. (2015). Robust network coding in transport networks. In *2015 IEEE Conference on Computer Communications Workshops (INFOCOM WORKSHOPS)*, pages 27–28.

[Le and Markopoulou, 2012] Le, A. and Markopoulou, A. (2012). Nc-audit: Auditing for network coding storage. In *International Symposium on Network Coding (NetCod)*, pages 155–160.

[Legout et al., 2006] Legout, A., Urvoy-Keller, G., and Michiardi, P. (2006). Rarest first and choke algorithms are enough. In *Proceedings of the 6th ACM SIGCOMM conference on Internet measurement*, pages 203–216. ACM.

[Lei et al., 2013] Lei, W., Yuwang, Y., Wei, Z., and Wei, L. (2013). Ncstorage: A prototype of network coding based distributed storage system. *TELKOMNIKA Indonesian Journal of Electrical Engineering*, 11(12):7689–7698.

[Li et al., 2010] Li, J., Yang, S., Wang, X., and Li, B. (2010). Tree-structured Data Regeneration in Distributed Storage Systems with Regenerating Codes. In *2010 IEEE INFOCOM*, pages 1–9.

[Mathieu Cunche, 2017] Mathieu Cunche, Jonathan Detchard, J. L. V. R. (2017). Openfec. http://openfec.org, visited 2017-03-09.

[Merriam-Webster, 2013] Merriam-Webster (2013). Meriam webster online dictionary. http://www.merriam-webster.com/dictionary/availability, visited 2018-02-15.

[Mickens and Noble, 2006] Mickens, J. W. and Noble, B. D. (2006). Exploiting availability prediction in distributed systems. In *Proceedings of the 3rd conference on Networked Systems Design & Implementation - Volume 3*, NSDI'06, pages 6–6, Berkeley, CA, USA. USENIX Association.

[Mogul et al., 1997] Mogul, J. C., Douglis, F., Feldmann, A., and Krishnamurthy, B. (1997). Potential benefits of delta encoding and data compression for http. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '97, pages 181–194, New York, NY, USA. ACM.

[MSPmentor, 2016] MSPmentor (2016). Cloud outages all providers arent created equal. http://mspmentor.net/msp-mentor/cloud-outages-all-providers-aren-t-created-equal, visited 2017-02-22.

[Naaman et al., 2004] Naaman, M., Garcia-Molina, H., and Paepcke, A. (2004). *Evaluation of ESI and Class-Based Delta Encoding*, pages 323–343. Springer Netherlands, Dordrecht.

[odrive, 2017] odrive (2017). Make cloud storage the way it should be. https://www.odrive.com/ visited 2017-02-22.

[Ookla, 2014] Ookla (2014). Global Broadband and Mobile Performance Data Compiled by Ookla — Net Index. http://explorer.netindex.com/maps, visited 2014-09-01.

[Otixo, 2017] Otixo (2017). Simple cloud manager. https://www.otixo.com/en/home-otixo/ visited 2017-02-22.

[Paul Crowley, 2006] Paul Crowley (2006). GF(2^{32}-5). http://www.lshift.net/blog/2006/11/29/gf232-5/, visited 2018-02-15.

[Pearson, 1905] Pearson, K. (1905). The Problem of the Random Walk. *Nature*, 72(1865):294.

[Pedersen et al., 2011] Pedersen, M., Heide, J., and Fitzek, F. (2011). Kodo: An open and research oriented network coding library. 6827:145–152.

[Pedersen et al., 2013] Pedersen, M., Heide, J., Vingelmann, P., and Fitzek, F. (2013). Network coding over the $2^{32} - 5$ prime field. In *Communications (ICC), 2013 IEEE International Conference on*, pages 2922–2927.

[Plank and Greenan, 2014] Plank, J. S. and Greenan, K. M. (2014). Jerasure: A library in c facilitating erasure coding for storage applications. http://jerasure.org/jerasure-2.0, visited 2017-03-09.

[Rashmi et al., 2015] Rashmi, K. V., Nakkiran, P., Wang, J., Shah, N. B., and Ramchandran, K. (2015). Having Your Cake and Eating It Too: Jointly Optimal Erasure Codes for I/O, Storage, and Network-bandwidth. pages 81–94.

[Rashmi et al., 2013a] Rashmi, K. V., Shah, N. B., Gu, D., Kuang, H., Borthakur, D., and Ramchandran, K. (2013a). A solution to the network challenges of data recovery in erasure-coded distributed storage systems: A study on the facebook warehouse cluster. In *Proceedings of the 5th USENIX Conference on Hot Topics in Storage and File Systems*, Berkeley, CA, USA. USENIX Association.

[Rashmi et al., 2013b] Rashmi, K. V., Shah, N. B., and Ramchandran, K. (2013b). A piggybacking design framework for read-and download-efficient distributed storage codes. In *Information Theory Proceedings (ISIT), 2013 IEEE International Symposium on*, pages 331–335.

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

[Rish, 2005] Rish, I. (2005). An empirical study of the naive Bayes classifier. In *IJCAI-01 workshop on "Empirical Methods in AI"*.

[Rouayheb et al., 2015] Rouayheb, S. E., Goparaju, S., Kiah, H. M., and Milenkovic, O. (2015). Synchronizing edits in distributed storage networks. In *2015 IEEE International Symposium on Information Theory (ISIT)*, pages 1472–1476.

[Sathiamoorthy et al., 2013a] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D., Dimakis, A. G., Vadali, R., Chen, S., and Borthakur, D. (2013a). XORing elephants: Novel erasure codes for big data. In *Proceedings of the VLDB Endowment*, volume 6, pages 325–336. VLDB Endowment.

[Sathiamoorthy et al., 2013b] Sathiamoorthy, M., Asteris, M., Papailiopoulos, D. S., Dimakis, A. G., Vadali, R., Chen, S., and Borthakur, D. (2013b). Xoring elephants: Novel erasure codes for big data. *CoRR*, abs/1301.3791.

[Schnjakin and Meinel, 2013] Schnjakin, M. and Meinel, C. (2013). Evaluation of Cloud-RAID: A Secure and Reliable Storage above the Clouds. In *2013 22nd International Conference on Computer Communication and Networks (ICCCN)*, pages 1–9.

[Shah et al., 2010] Shah, N., Rashmi, K., and Kumar, P. (2010). A flexible class of regenerating codes for distributed storage. In *2010 IEEE International Symposium on Information Theory Proceedings (ISIT)*, pages 1943–1947.

[Shah et al., 2012] Shah, N. B., Rashmi, K. V., Kumar, P. V., and Ramchandran, K. (2012). Distributed Storage Codes with Repair-by-Transfer and Non-achievability of Interior Points on the Storage-Bandwidth Tradeoff. *IEEE Transactions on Information Theory*, 58(3):1837–1852. arXiv: 1011.2361.

[Shvachko et al., 2010] Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The Hadoop distributed file system. In *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*, pages 1–10.

[Silberstein et al., 2014] Silberstein, M., Ganesh, L., Wang, Y., Alvisi, L., and Dahlin, M. (2014). Lazy Means Smart: Reducing Repair Bandwidth Costs in Erasure-coded Distributed Storage. In *Proceedings of International Conference on Systems and Storage*, SYSTOR 2014, pages 15:1–15:7, New York, NY, USA. ACM.

[Sloane, 1991] Sloane, N. J. A. (1991). The on-line encyclopedia of integer sequences. http://oeis.org/A000041, visited 2016-09-02.

[Soljanin, 2010] Soljanin, E. (2010). Reducing delay with coding in (mobile) multi-agent information transfer. In *2010 48th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1428–1433.

[Song et al., 2015] Song, G., Kim, S., and Seo, D. (2015). Saveme: client-side aggregation of cloud storage. *IEEE Transactions on Consumer Electronics*, 61(3):302–310.

[Stanley, 2011] Stanley, R. P. (2011). *Enumerative Combinatorics: Volume 1*. Cambridge University Press, New York, NY, USA, 2nd edition.

[Steinwurf ApS, 2014] Steinwurf ApS (2014). Storage benchmarks on GitHub. https://github.com/steinwurf/storage-benchmarks, visited 2018-02-18.

[Steven Max Paterson, 2014] Steven Max Paterson (2014). How MIT and Caltech's coding breakthrough could accelerate mobile network speeds. http://www.networkworld.com/article/2342846/data-breach/how-mit-and-caltech-s-coding-breakthrough-could-accelerate-/mobile-network-speeds.html, visited 2018-02-15.

[Suel and Memon, 2002] Suel, T. and Memon, N. (2002). *Algorithms for delta compression and remote file synchronization*. Lossless Compression Handbook. Academic Press.

[Szymonacedaski et al., 2005] Szymonacedaski, Deb, S., Medard, M., and Koetter, R. (2005). How Good is Random Linear Coding Based Distributed Networked Storage? In *1st Workshop on Network Coding, Theory and Applications (NetCod)*.

[Taleb and Ksentini, 2013a] Taleb, T. and Ksentini, A. (2013a). An analytical model for follow me cloud. In *2013 IEEE Global Communications Conference (GLOBECOM)*, pages 1291–1296.

[Taleb and Ksentini, 2013b] Taleb, T. and Ksentini, A. (2013b). Follow me cloud: interworking federated clouds and distributed mobile networks. *IEEE Network*, 27(5):12–19.

[Van et al., 2012] Van, V. T., Yuen, C., and Li, J. (2012). Non-homogeneous distributed storage systems. In *2012 50th Annual Allerton Conference on Communication, Control, and Computing (Allerton)*, pages 1133–1140.

[Van Horssen et al., 2002] Van Horssen, P., Pebesma, E., and Schot, P. (2002). Uncertainties in spatially aggregated predictions from a logistic regression model. *Ecological Modelling*, 154(1):93–101.

[Wang et al., 2016] Wang, Q., Médard, M., and Skoglund, M. (2016). Efficient compression algorithm for file updates under random insertions and deletions. In *2016 IEEE Information Theory Workshop (ITW)*, pages 434–438.

[Weiyan Wang, 2014] Weiyan Wang, H. K. (2014). Saving capacity with HDFS RAID. https://code.facebook.com/posts/536638663113101/saving-capacity-with-hdfs-raid, visited 2014-07-01.

[Wu and Dimakis, 2009] Wu, Y. and Dimakis, A. (2009). Reducing repair traffic for erasure coding-based storage via interference alignment. In *IEEE International Symposium on Information Theory, 2009. ISIT 2009*, pages 2276–2280.

[ZeroPC, 2017] ZeroPC (2017). Your continent navigator for the cloud. https://www.zeropc.com/, visited 2017-02-22.

ISSN (online): 2446-1628
ISBN (online): 978-87-7210-177-4

AALBORG UNIVERSITY PRESS