DISTRIBUTED INFORMATION STORAGE

BY

JAMES R. ROCHE

STANFORD UNIVERSITY

TECHNICAL REPORT NO. 79
MARCH 1992

PREPARED UNDER THE AUSPICES
OF
NATIONAL SCIENCE FOUNDATION
GRANT NCR-89-14538

DEPARTMENT OF STATISTICS
STANFORD UNIVERSITY
STANFORD, CALIFORNIA
DISTRIBUTED INFORMATION STORAGE

BY

JAMES R. ROCHE

STANFORD UNIVERSITY

TECHNICAL REPORT NO. 79
MARCH 1992

PREPARED UNDER THE AUSPICES
OF
NATIONAL SCIENCE FOUNDATION
GRANT NCR-89-14538

DEPARTMENT OF STATISTICS
STANFORD UNIVERSITY
STANFORD, CALIFORNIA
DISTRIBUTED INFORMATION STORAGE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
James R. Roche
March 1992
Abstract

Information is distributed in many different applications. In conventional parallel processing, for example, data is distributed in order to speed up computation. We consider the less-studied problem of distributing data among different sites for fault-tolerance. The primary example that we consider involves storing information on a network of disks so that the information can be recovered reliably even when some of the disks fail. The techniques developed in this thesis apply equally well, however, to problems in which communication links can fail and information packets can be lost.

Using techniques from Galois field theory and network flow theory, we show how to store information at different sites with an absolute minimum of redundancy so as to prevent any loss of data if several sites become inaccessible. We allow the different sites to have different storage capacities and to be arranged in a variety of network topologies.

The theoretical results on reliable information storage derived for general networks can be specialized to a particular configuration of disks that is widely used in practice. Although the information storage scheme for this configuration is simple in principle, there are practical difficulties because of the need to update parity-check blocks continually as data blocks are modified.

For the setup above, the delay between reading a parity block and writing its updated value back onto the disk can seriously degrade the performance of a disk array. We analyze a technique called floating parity track that dramatically reduces the average delay between reading and writing a parity block. We find that the delay can typically be reduced by a factor of thirty in return for a slight decrease in storage efficiency.
Acknowledgments

It is with pleasure that I acknowledge my debt to my advisor, Professor Tom Cover. His clarity of thought and his enthusiastic, even playful, approach to research have helped to make my time at Stanford very enjoyable. I am grateful to my associate advisor, Professor Martin Hellman, for his insights on technical and nontechnical matters alike. Thanks to Professor John T. Gill III for serving on my orals committee and to Professor John Hennessy for chairing the committee and for reading this dissertation. Thanks also to Dr. Shuki Bruck for bringing some related work to my attention.

Fellowships from the Office of Naval Research and IBM got me through the first four years of graduate study. Several research grants \(^1\) saw me through the last \((n - 4)\) years.

I would like to thank Laura Ekroot, Will Equitz, Drew Nobel, Joy Thomas, and Mitchell Trott for making the office such an interesting place to work. Drew’s attempt to introduce an office theme song will not soon be forgotten, nor will Mitchell’s seminal work on infinitely self-similar something-or-others.

I am especially grateful to the people who transferred the contents of this dissertation from scratch paper and paper napkins into the computer. Bryan Biegel, Denise Cuevas, Luis Rodríguez, Dan Scales, and Mitchell Trott all spent many hours helping me to finish. The intercession of St. Jude is acknowledged with equal gratitude.

My luck in finding roommates has been remarkable. It has been a privilege to live with Luis Rodríguez, Scott Gallert, and Greg Scott. Special recognition is due my

\(^1\)This work was supported in part by NSF Grant NCR-89-145538, DARPA Contract JFBI-91-218, and JSEP DAAL03-91-0010.
former roommate Robert Taft, an individual of unusual generosity. There is nothing that he would not do to a friend.

In addition to the friends already listed, there are many others who have enriched my life in California. The friends that I met in the St. Ann Young Adult Group and in pro-life work are particularly cherished. Most cherished of all is Maria Placer, who is never far from my thoughts.

Finally, I thank my parents, my brother Joe, and my sisters Ellen and Weezy for reflecting the love of the One without Whom this thesis would certainly not have been written.
# Contents

Abstract

Acknowledgments

## 1 Introduction

1.1 Overview ........................................ 1
1.2 Theoretical Limits ................................ 1
1.3 Improving the Performance of Disk Arrays ....... 6
1.4 Outline of Thesis and Related Work ............... 7
1.5 Contributions of this Thesis ....................... 9

## 2 Single-User Distributed Information Storage

2.1 Disks of Equal Capacity .......................... 11
   2.1.1 A Preliminary Problem ....................... 11
   2.1.2 A More General Distributed Information Problem .... 17
2.2 Disks of Unequal Capacities ...................... 20
   2.2.1 Derivation of Capacity with Disk Failures ....... 20
   2.2.2 Computational Considerations .................. 23

## 3 Multiple Users with Common Information

3.1 Identical Information ............................ 24
3.2 Sequentially Refined Information ................. 27
   3.2.1 No Disk or Link Failures .................... 28
   3.2.2 Disk or Link Failures Allowed ............... 50
3.3 Other Problems to Consider ........................................... 52

4 Storage of Independent Information ................................. 54
  4.1 Introduction ............................................................ 54
  4.2 Cases Without Disk or Link Failures ............................ 55
  4.3 Node and Link Failures Allowed ................................. 68
    4.3.1 Maximally Connected Networks .................. 68
    4.3.2 Users with Private Disks ................................. 72
  4.4 Other Problems to Consider .................................... 75

5 Floating Parity for Disk Arrays .................................... 77
  5.1 Introduction ............................................................ 77
  5.2 Ordinary Parity Technique ..................................... 78
  5.3 Terminology .............................................................. 80
  5.4 Schemes for Improving the Write Performance of the Parity Technique .......... 81
    5.4.1 Overview ......................................................... 81
    5.4.2 Double Parity ................................................... 81
    5.4.3 Floating Parity Track .......................................... 85
  5.5 Summary ................................................................. 90

6 Analysis of the Floating-Parity Scheme ........................... 91
  6.1 Introduction ............................................................ 91
  6.2 Single Track ($T = 1$) .............................................. 92
    6.2.1 Without Skew .................................................. 92
    6.2.2 Single Track, With Skew ................................. 96
  6.3 Multiple Tracks ($T > 1$) ......................................... 98
    6.3.1 Without Skew .................................................. 98
    6.3.2 Multiple Tracks ($T > 1$), With Skew ................... 104
    6.3.3 Dependent Case ................................................ 105
  6.4 Summary of Analysis .............................................. 106
  6.5 Other Problems to Consider .................................... 107
7 Summary

7.1 Conclusions ................................................. 108
7.2 Suggestions for Further Research .......................... 109

Bibliography .................................................. 111
List of Tables

3.1 An example for which sequential refinement is impossible ........ 52
5.1 Storage efficiency and table overheads for double parity .......... 85
5.2 Storage efficiency and table overheads for floating parity .......... 89
List of Figures

2.1 A naive approach to information storage .................................. 12
2.2 Information storage for three sites ........................................... 13
2.3 Distributed information storage with small chunks ...................... 19
3.1 A symmetric information storage example .................................. 25
3.2 An asymmetric information storage example ............................... 26
3.3 An example in which sequential refinement is possible .................. 28
3.4 Realization of sequential refinement ......................................... 29
3.5 An example in which sequential refinement is not possible .............. 29
3.6 An example illustrating the definition of $d_j$ ............................... 31
3.7 An example illustrating the definition of $N(s)$ ............................. 37
3.8 Continuation of example illustrating $N(s)$ ................................ 37
3.9 An example illustrating Theorem 3 .......................................... 38
3.10 Storing information using linear algebra ................................... 39
3.11 Realization in $GF(2)$ .......................................................... 41
3.12 Sequential refinement example with link failures allowed .............. 50
4.1 Example showing storage of independent information .................... 58
4.2 Transformation from storage to flow ........................................ 59
4.3 An achievable $(x_1, x_2, x_3)$ for the flow problem ....................... 59
4.4 An achievable $(x_1, x_2, x_3)$ for the storage problem .................... 60
4.5 An example showing capacities of arcs ..................................... 61
4.6 An example showing an achievable flow .................................... 62
4.7 A storage network .................................................................. 64
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.8 A flow network equivalent to the previous storage network</td>
<td>64</td>
</tr>
<tr>
<td>4.9 Users with private disks</td>
<td>72</td>
</tr>
<tr>
<td>4.10 The communal network derived from the previous network</td>
<td>73</td>
</tr>
<tr>
<td>5.1 A parity example with three data tracks and one parity track</td>
<td>78</td>
</tr>
<tr>
<td>5.2 The double-parity method with 3 data tracks</td>
<td>81</td>
</tr>
<tr>
<td>5.3 Floating parity with three families</td>
<td>86</td>
</tr>
<tr>
<td>6.1 Expected delay for double parity</td>
<td>101</td>
</tr>
<tr>
<td>6.2 Expected delay for floating parity</td>
<td>104</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Overview

Traditional single-user information theory deals with both information transmission and information storage. Multi-user information theory, however, has dealt almost exclusively with information transmission, or communication. There has been little systematic study of multi-user information storage.

This dissertation identifies a variety of problems that will, it is hoped, help to define the boundaries of a general theory of distributed information storage. Some of the problems can be solved readily using known results from coding theory and network flow theory, while others require entirely new methods of solution.

1.2 Theoretical Limits

Information is distributed in many different applications. In conventional parallel processing, for example, data is distributed in order to speed up computation. We consider the less-studied problem of distributing data among different sites for fault-tolerance. Some of the ideas used in our analysis are similar to those used in "secret sharing," in which data is distributed for privacy.

Using simple techniques from Galois field theory and network flow theory, we show how to store information at different sites with an absolute minimum of redundancy.
so as to prevent any loss of data if several sites become inaccessible. We allow the different sites to have different storage capacities and to be arranged in a variety of network topologies.

We begin by considering the following problem, essentially due to Singleton [65]: Given a body of information (e.g., a library) that we wish to store and reliably recover from a network of storage sites (e.g., disks), how well can we protect ourselves against multiple site failures while minimizing the amount of storage space used? More specifically, represent the library by a string of $2t$ bits, store $t$ bits on each of $n$ disks (where $n > 2$), and try to reconstruct the original library of $2t$ bits from the contents of any pair of disks. (We could, of course, store the full $2t$ bits on each disk, but this would require twice as much storage space.)

It is straightforward to show that the desired reconstruction is impossible if each disk contains *subsets* of the original bits. However, by storing *functions* of the original bits on the various disks, we can hope to reconstruct the original $2t$ bits given the contents of any two disks, provided that the total number of disks, $n$, is not too large. By thinking of each $t$-bit sequence as an element of a finite field, or Galois field, with $2^t$ elements, we can represent the contents of each disk as an appropriate linear combination of two field elements, $U_1$ and $U_2$. The field elements $U_1$ and $U_2$ are the field elements having as their binary representations the first and second halves, respectively, of the original $2t$ information bits. The only restriction is that the total number of disks is bounded above by the number of different elements in the field. More precisely,

$$n \leq 2^t + 1.$$ 

This upper bound on the number of disks can be established by an argument due to Singleton involving orthogonal Latin squares. The Galois-field construction actually achieves this upper bound exactly. Thus this bound $n \leq 2^t + 1$ is necessary and sufficient for reliable recovery of the original information.

Having addressed the problem of safeguarding $2t$ bits by storing $t$ bits at each of $n$ sites, we consider the following generalization: Given $kt$ bits of information to be protected, store $t$ bits on each of $n$ disks (where $n > k$), and try to reconstruct the original $kt$ bits from the contents of any $k$ disks. (It is clear that at least $k$ disks must
be available if we are to recover the original $kt$ bits.)

The problem above can be solved in essentially the same way as the previous problem, in which we had $k = 2$. An argument based on orthogonal Latin hypercubes (again due to Singleton) shows that the total number of disks, $n$, is bounded as follows:

$$n \leq 2^t + k - 1.$$  

A Galois-field construction actually allows us to reconstruct the original information from any $k$ of $n$ sites as long as $n \leq 2^t + 1$. For practical values of $k$ and $t$, the two bounds are almost indistinguishable.

The results given so far have been based on the assumption that all disks have the same storage capacity, $t$ bits. Let us now drop this assumption. Suppose that we have $n$ disks with capacities $C_1, C_2, \ldots, C_n$ bits, where $C_1 \leq C_2 \leq \ldots \leq C_n$. If it is assumed that at least $k$ of the $n$ disks will always be accessible, how much information can we store reliably in the network?

Certainly we cannot hope to store more than $C_1 + C_2 + \cdots + C_k$ bits reliably, because the $k$ disks to which we have access might be the $k$ smallest disks. (This upper bound holds even if we are told ahead of time that these are the $k$ disks to which we will have access.)

Happily, there is a constructive scheme by which we can actually achieve the upper bound of $C_1 + C_2 + \cdots + C_k$ bits to be stored reliably. This scheme involves the use of a systematic erasure-correcting code over the Galois field $GF(2^{C_k})$ with $2^{C_k}$ elements. The only constraint is that the total number of disks, $n$, should not exceed $2^{C_k}$. In practice, $C_k$ is in the thousands or millions of bits, so $2^{C_k}$ is an immense number. Hence there is no practical restriction on $n$, the number of disks.

We have thus extended the initial results (for disks of equal capacity) to the case with disks of unequal capacity. Now let us drop the assumption that there is a single user with access to a number of different disks. Suppose instead that there are $m$ different users, $U_1, U_2, \ldots, U_m$, with each $U_i$ having access to some subset $S_i$ of a set of $n$ disks, $\{D_1, D_2, \ldots, D_n\}$. (These subsets can overlap.) For simplicity, assume that the $n$ disks all have the same capacity, $C$ bits. Suppose further that as many as $f$ of the $n$ disks can fail. Under these conditions, how much common information
can be reliably recovered by all \( m \) users?

It is clear that User \( i \), with access to \( |S_i| \) disks, cannot hope to recover more than \((|S_i| - f)\) disks' worth of information reliably. Letting \( X_i \) be the amount of information that User \( i \) can reliably recover, we thus have

\[
X_i \leq M_i \quad \text{bits},
\]

where \( M_i = \max\{0, (|S_i| - f)C\} \).

Therefore the amount of common information that can be reliably recovered by all \( m \) users must satisfy

\[
X_{\text{common}} \leq \min_i \{M_i\} \quad \text{bits.}
\]

In fact, it follows readily from the previous results that a constructive information distribution scheme actually achieves this bound on the amount of common information that can be reliably recovered.

We might generalize the preceding problem further by supposing that the body of information to be stored in the network contains some especially important information that should be available to all users, followed by some less important information that should be available to any user with access to a sufficient amount of disk space. That is, represent the original library as a bit sequence, \((Y_1, Y_2, \ldots, Y_r)\). Now we try to find the set of achievable \( m \)-tuples \( (X_1, X_2, \ldots, X_m) \) such that for \( 1 \leq i \leq m \), user \( U_i \) can reliably recover the first \( X_i \) bits of the library, \((Y_1, Y_2, \ldots, Y_X)\).

Ideally, we might hope to achieve \((X_1, \ldots, X_m) = (M_1, \ldots, M_m)\), where \( M_i = \max\{0, (|S_i| - f)C\} \) as before. In such a case, we say that complete sequential refinement of information is possible. It is not always possible for each User \( i \) to simultaneously recover the first \( M_i \) bits of the library. However, we find necessary and sufficient conditions for a general network to admit complete sequential refinement. We also demonstrate a constructive scheme for achieving sequential refinement given any network satisfying the conditions.

Sequential refinement has applications to information transmission as well. For example, it is often advantageous to transmit information progressively from a sender to a receiver, sending the most important data first and then transmitting more
CHAPTER 1. INTRODUCTION

details as time and bandwidth permit. See Equitz [24] and Equitz and Cover [25] for
an analysis of sequential refinement in communication applications.

So far we have assumed that all users are interested in the same information. Let
us now turn to the case in which the \(m\) users \(U_1, \ldots, U_m\) wish to store independent
information in a network of \(n\) disks, \(D_1, \ldots, D_n\), with respective capacities \(C_1, \ldots, C_n\)
bits. Suppose that for \(1 \leq i \leq m\), User \(i\) has access to some subset \(S_i\) of the \(n\)
disks. If User \(i\) has \(X_i\) bits of information that he wishes to recover reliably, what
different \(m\)-tuples \((X_1, \ldots, X_m)\) are achievable? Equivalently, how much information
can each user simultaneously store in the network if different users store independent
information?

For simplicity, let us first suppose that there are no disk failures at all. In this
case, the Galois-field techniques used to solve the previous problems are unnecessary.
Using straightforward information-theory arguments, we find that it is optimal just
to store different subsets of the original bits on the various disks. Thus the different
users can simply reserve blocks of space on the various disks.

Simple cut-set arguments place an outer bound on the set of \(m\)-tuples \((X_1, \ldots, X_m)\)
that are feasible. Using a modified version of the Ford-Fulkerson max-flow, min-cut
theorem from network flow theory, we show that the entire region allowed by the
cut-set bounds can actually be achieved by having each user reserve the appropriate
amount of space on each disk. This result holds whether or not the \(n\) disks have the
same capacity. Furthermore, the algorithm for allocating space is computationally
efficient; it can be carried out in time quadratic or cubic in \((m + n)\).

The problem takes on an information-theoretic character if some of the disks are
allowed to fail. More generally, we suppose that for \(1 \leq i \leq m\), User \(i\) might lose
access to as many as \(f_i\) disks, either because of a disk failure (node failure) or because
of a link failure. (If we simply allow any \(f\) of the \(n\) total disks to fail, then we have
\(f_1 = \cdots = f_m = f\).)

This last problem is perhaps the most interesting problem that we have consid-
ered. Under fairly general circumstances—\(e.g.,\) when each user has exclusive access to
a sufficient number of sufficiently large disks, in addition to the disks which he shares
with other users—it is possible to extend the results from the previous problem by
using Galois-field techniques to characterize exactly the set of achievable $m$-tuples $(X_1, \ldots, X_m)$. In these cases, we can find an outer bound on the region of simultaneously achievable values $X_1, \ldots, X_m$, and we can constructively achieve all the values in this region.

We have discussed a number of problems involving multiple users who simultaneously store information in a network of storage sites. Different storage sites—often called disks for concreteness—have been allowed to have different capacities. Different users have been allowed access to different (perhaps overlapping) subsets of the disks and have been permitted to store either common or independent information. We have considered cases in which different users can tolerate different numbers of disk or link failures.

The problems that we have identified may be thought of as the extreme points of a general theory of network information storage. The general theory would include all of these extreme points as special cases. Based on the results already obtained, we are hopeful that even the most general problems may be susceptible to solution.

### 1.3 Improving the Performance of Disk Arrays

The problems described in the previous section deal primarily with storage efficiency. When users are continually updating the information stored on a set of disks, however, the performance of the disk system can become a more important consideration than storage efficiency. Conventional techniques for creating fault-tolerant disk arrays can significantly increase the time spent modifying old data.

Computer systems use many different methods to prevent loss of data in the event of disk failures. One of the best ways to implement such protection while conserving disk space is a parity technique described in [42], [15], and [55]. This technique requires fewer disks than duplexing (the duplication of all data) but still achieves much faster recovery times than checkpoint-and-log techniques.

The parity technique is a cost-effective way to maintain a fault-tolerant disk array. Its main drawback is that it requires four disk accesses to update a data block—two to read old data and parity, and two to write new data and parity. Due to its relatively
poor update performance, the parity technique can degrade disk system performance with respect to a system that does not use this technique. Therefore, ways to improve write performance must be found if the system is to enjoy widespread use.

In [49], Menon and Kasson describe four related schemes to improve the write performance of disk arrays that use the parity technique. The schemes all improve write performance by sacrificing some storage efficiency and by relaxing the requirement that the modified data and parity blocks be written back into their original locations. In all of the strategies, the updated block can be written to a free location after a delay that is much shorter than the time required for a full revolution of the disk. The average time to update a block is thus improved, and the number of disk accesses is reduced from four to three or even to two.

The four schemes mentioned above can all be analyzed in essentially the same way. In this dissertation we focus on the floating-parity-track technique, which appears to be the best of the four schemes. For simplicity in comparing the floating-parity-track method with the straightforward parity technique, we assume low I/O rates (so that there is no device queuing) and compute only the portion of the service time due to rotational delay of the disks, or disk latency. (The average radial delay, or seek time, is about the same for both techniques.) We find that the floating-parity-track method achieves substantially better single-block update performance than the conventional parity technique, with minimal loss of storage efficiency.

1.4 Outline of Thesis and Related Work

In this thesis we are primarily concerned with the storage of information rather than its transmission. Much of the analysis that we present, however, can readily be applied to problems in communication theory. By considering the communication links between information storage sites rather than focusing on the storage sites themselves, we can adapt many of the results to solve related problems in communications.

Conversely, the results that we derive for storage networks owe much to fundamental work in error-correcting codes, information theory, and network flow theory, all of which have strong ties to the theory of communication.
CHAPTER 1. INTRODUCTION

The preliminary results of Chapter 2 follow from work done by Singleton on maximum-distance-separable erasure-correcting codes [65]. Singleton's work, in turn, is a natural application of Reed-Solomon codes [59], which are used to overcome noise in many different settings. (See Ko et al. [41] for an application to compact disc players.)

Singleton's results were applied to problems involving disk and link failures by Roche, Dembo, and Nobel [60], Rabin [56], and Chih-Lin I, Ayanoglu, Gitlin, and Mazo [35]. The first three authors were interested primarily in the case of disk failures, while the other authors were more interested in the case of communication link failures.

Singleton's results also contributed to the very interesting study of secret sharing, developed by Blakley [2], Shamir [63], and Karnin, Greene, and Hellman [40]. In secret sharing, information is distributed for privacy rather than for fault-tolerance. For example, two people may each be given a portion of a secret in such a way that individually they have no information whatsoever about the secret, yet they can pool their information to recover the entire secret. For more recent work in the area of secret sharing, see Yamamoto [68] and Dwork [21].

The fundamental ideas of fault-tolerance and secret sharing have been applied in recent years to the theory of distributed computation to yield a rich theory of secure distributed computing. See Yao [70] and Beaver [4]. Also see Brassard [10] for an overview of secure distributed computing and related areas.

The third chapter of this dissertation addresses the problem of sequential refinement of information in data storage networks. For an important problem in distributed file storage with a similar flavor, see Naor and Roth [53]. For work on a different type of sequential refinement, see Equitz [24] and Equitz and Cover [25]. Also see El Gamal and Cover [22] for work on an analogous problem in multi-user information theory.

The fourth chapter of this dissertation deals with problems involving multiple users who store independent information. The solutions to these problems use results from network flow theory. We have been primarily interested in storage efficiency.
but for large networks, computational complexity can also be an important consideration. We find upper bounds on the computational complexity of some information storage schemes by using well-known results on the complexity of solving the max-flow problem in networks. (See Chen [14] and Papadimitriou and Steiglitz [54].) See Gusfield, Martel, and Fernandez-Baca [31] for algorithms that are particularly efficient for certain network topologies.

The fifth and sixth chapters of this dissertation address some of the practical difficulties of implementing fault-tolerant disk arrays. See Menon and Kasson [49] for a discussion of implementation details not covered in this thesis. Also see Menon and Hartung [50], Chen and Patterson [13], and Muntz and Lui [52] for some other techniques that have been proposed to improve disk performance.

1.5 Contributions of this Thesis

The primary contribution of this dissertation is the call for a general theory of distributed information storage. Although single-user information theory deals with both information transmission and information storage, multi-user information theory has dealt almost exclusively with information transmission, or communication. Problems of multi-user information storage that have been identified and solved have tended to be rather special cases arising in other disciplines.

This dissertation identifies a variety of problems that will, it is hoped, help to define the boundaries of a general theory of distributed information storage. The results obtained so far indicate that we can use techniques from coding theory and network flow theory to find practical, constructive storage schemes that can be proved optimal using techniques from information theory.

The preliminary problems given in Section 2.1 are essentially due to Singleton [65], but the extension to the case of arbitrary unequal disk capacities (Section 2.2) appears to be new. The work in Chapter 3, dealing with the complete sequential refinement of information, is original.

Chapter 4 addresses problems involving multiple users with independent information. The results of Section 4.2, when no disk or link failures are allowed, follow
readily from known results in network flow theory; nonetheless, the application to information storage and the proof of optimality appear for this application to be new. The results of Section 4.3, in which disk and link failures are allowed, are all original.

Chapter 5 first describes a parity technique for maintaining fault-tolerant disk arrays (due to Lawlor [42], Patterson et al. [55], and Clark et al. [15]), then describes refinements (due to Menon and Kasson [49]) designed to improve the performance of the system when data is updated. Chapter 6 is new and provides a theoretical analysis of the techniques described in Chapter 5.
Chapter 2

Single-User Distributed Information Storage

2.1 Disks of Equal Capacity

2.1.1 A Preliminary Problem

Suppose that there is some body of information distributed throughout a number of libraries; say that there are \( n \) libraries in all. We would like to store half the information at each library so that if any two libraries are open, we can recover all of the information. How large can \( n \) be?

More precisely, suppose that we have \( 2t \) bits of information and that we store \( t \) bits per site. Find \( n^* \), the maximum possible number of sites such that the bits stored at any pair of sites allow us to recover all \( 2t \) bits of information.

As a first attempt, we might store the first \( t \) bits at some of the sites and last \( t \) bits at the other sites. Letting \( u_1 = (y_1, y_2, \ldots, y_t) \) be the vector of the first \( t \) bits and \( u_2 = (y_{t+1}, \ldots, y_{2t}) \) be the vector of the remaining \( t \) bits, we obtain Figure 2.1.1.

This approach is clearly too simple, however, because it does not even work with three sites; the two sites that we access might be exact duplicates, in which case we can recover only half of the \( 2t \) bits of information.

A slightly less naive approach is to store arbitrary \( t \)-bit subsets of the original
2t bits at the various sites. This approach is also doomed to failure, though, for n greater than 2. Suppose that \( y_k \), the \( k \)th of the 2t bits, is stored at sites \( i \) and \( j \). Then it follows that there is at least one bit of redundancy in the information stored at the two sites, hence we can recover at most \( 2t - 1 \) bits of information from the two sites, not \( 2t \) bits. The only way to avoid this difficulty is to make sure that the subsets of bits stored at any two sites are disjoint, but then we are essentially left with the first approach, which does not suffice for any \( n \) greater than 2.

There is no satisfactory solution in which we simply store a subset of the original bits at each site. However, there is another possibility: we can store functions of the original bits at each site. When \( n \), the number of sites, equals three, it is not hard to find an appropriate scheme. We can simply store \( u_1 \) (the vector of the first \( t \) bits) at Site 1, store \( u_2 \) (the vector of the remaining \( t \) bits) at Site 2, and store their bitwise exclusive-OR, \( u_1 \oplus u_2 \), at Site 3. (See Figure 2.2.)

Site 3 is simply a parity check on Sites 1 and 2, and its contents can be exclusive-ORed with either of the other two sites to obtain the contents of the remaining site. Thus, for each \( t \geq 1 \), we see that it is possible to let \( n = 3 \), so \( n^* \geq 3 \) for each \( t \geq 1 \).

Before continuing the search for parity-check schemes with more than three sites, let us first find an upper bound on the maximum number of sites, \( n^* \), as a function of \( t \), the number of bits stored at each site. Note that there are \( 2^{2t} \) possible information
sequences \((y_1, y_2, \ldots, y_{2t})\). Impose a uniform distribution on the \(2^{2t}\) possibilities; i.e., assume that each of the \(2^{2t}\) different possible information sequences has probability \(2^{-2t}\). Then the entropy of the sequence \((y_1, y_2, \ldots, y_{2t})\) is exactly \(2t\) bits. (See Cover and Thomas [17], Blahut [9], or Gallager [29] for a review of the definitions and properties of the information-theoretic quantities entropy and mutual information.)

For each \(j \leq n\), the \(j^{th}\) storage site contains \(t\) bits and may thus be identified with a discrete random variable \(V_j\) having \(2^t\) possible outcomes. Each such random variable has entropy \(H(V_j) \leq t\) bits, with equality if and only if the \(2^t\) possible \(t\)-bit sequences stored at the site are equiprobable. Given any pair of sites, site \(i\) and site \(j\), the entropy of their combined contents is just the joint entropy \(H(V_i, V_j)\). However, it is well known that the joint entropy is at most the sum of the individual entropies; that is,

\[ H(V_i, V_j) \leq H(V_i) + H(V_j). \]

But \(H(V_i) \leq t\) bits, with equality iff \(V_i\) is uniformly distributed; similarly, \(H(V_j) \leq t\) bits, with equality iff \(V_j\) is uniformly distributed. Therefore \(H(V_i, V_j) \leq 2t\) bits, with equality only if \(V_i\) and \(V_j\) are uniformly distributed (and independent).

Since we wish to recover the entire sequence \((y_1, \ldots, y_{2t})\), which has an entropy
of $2t$ bits, it follows that $V_j$ must be uniformly distributed for each $j, 1 \leq j \leq n$. Thus, for every site $j$, each of the $2^t$ possible outcomes of $V_j$ must be associated with exactly $2^t$ of the $2^{2t}$ possible information sequences $y = (y_1, \ldots, y_{2t})$.

It is easy to see that for each pair of sites $i$ and $j$, there must be a one-to-one correspondence between the $2^{2t}$ possible values of the random variable $(V_i, V_j)$ and the $2^{2t}$ possible values of the information sequence $y = (y_1, \ldots, y_{2t})$. Thus the contents of any individual storage site are consistent with exactly $2^t$ possible values of the information sequence $y$, and the joint contents of any two (or more) sites are consistent with exactly one value of $y$. Given the contents of the $n$ storage sites, the $2^{2t}$ possible values of $y$ can be partitioned as follows. There is exactly one value of $y$ that is consistent with the data stored at all $n$ sites. Each of the other $2^{2t} - 1$ possible values of $y$ can be consistent with at most one site’s data. It follows that

$$n(2^t - 1) \leq 2^{2t} - 1,$$

with equality only if every possible value of $y$ is consistent with the contents of some storage site. Therefore

$$n^* \leq \frac{2^{2t} - 1}{2^t - 1} = 2^t + 1.$$

Somewhat surprisingly, the upper bound just derived on $n^*$ can actually be achieved. We observe that our earlier construction with three sites used the XOR operation, which is just addition in the Galois field of characteristic 2. (For a review of the basic properties of finite fields, or Galois fields, see Blahut [8] or Berlekamp [6].) By considering all $t$ bits at each site together, we may think of the contents of each storage site, $V_j$, as an element of $GF(2^t)$, the Galois field with $2^t$ elements. Let $U_1 \in GF(2^t)$ be the field element having as its binary representation the first half of the information vector, namely $(y_1, \ldots, y_t)$. Let $U_2 \in GF(2^t)$ be the field element having as its binary representation the second half of the information vector, namely $(y_{t+1}, \ldots, y_{2t})$. Then we wish to store at each site a different linear combination of $U_1$ and $U_2$ (over $GF(2^t)$) such that $U_1$ and $U_2$ can be uniquely determined given any pair $(V_i, V_j)$ of field elements associated with the contents of storage sites $i$ and $j$. Denoting the $2^t$ elements of $GF(2^t)$ by $\alpha_1, \alpha_2, \ldots, \alpha_{2^t}$, we can choose $V_1, V_2, \ldots, V_{2^t+1}$ as follows.
For $1 \leq j \leq 2^t$, let

$$V_j = \alpha_j U_1 + U_2.$$ 

For $j = 2^t + 1$, let

$$V_{2^t+1} = U_1$$

It is easy to verify that for $i \neq j$, $V_i$ and $V_j$ are linearly independent functions of $U_1$ and $U_2$ over $GF(2^t)$ and can thus be used to reconstruct the field elements $U_1$ and $U_2$, or equivalently, the information vector $Y = (y_1, \ldots, y_{2^t})$.

Since the construction above has $n = 2^t + 1$ different storage sites, we see that the upper bound derived earlier is tight. Thus we have the answer to the question posed at the beginning of this chapter: the maximum possible number of sites such that the data stored at any pair of sites allows us to recover all $2t$ bits of information is

$$n^* = 2^t + 1. \tag{2.1}$$

For any given value of $t$, the construction above can be expressed in terms of the ordinary XOR operation. We show for the case $t = 2$ how this can be done.

**Example:** Suppose that $t = 2$. Then $n^* = 2^t + 1 = 5$. Thus we wish to store $2t$, or four, bits of information $(y_1, y_2, y_3, y_4)$ in a distributed fashion among $n^*$, or five, different storage sites, with two bits of data per site, such that the contents of any pair of sites can be used to reconstruct $y = (y_1, y_2, y_3, y_4)$.

**Solution:** We will work in the field $GF(2^2) = GF(4)$. Represent each field element of $GF(4)$ by a two-bit vector $(a_0, a_1)$. Add field elements $(a_0, a_1)$ and $(b_0, b_1)$ componentwise modulo 2 to obtain the sum $(a_0 \oplus b_0, a_1 \oplus b_1)$. Thus, for example, $(1, 0) + (1, 1) = (0, 1)$. Multiply field elements by representing them as polynomials and multiplying the polynomials modulo the primitive polynomial $x^2 + x + 1$, again using mod-2 arithmetic. For example, $(0, 1) \cdot (1, 1) \leftrightarrow (0 \cdot 1 + 1 \cdot x)(1 \cdot 1 + 1 \cdot x) = x(1 + x) = x + x^2 \equiv 1 \pmod{x^2 + x + 1}$, and $1 = 1 + 0 \cdot x \leftrightarrow (1, 0)$. Thus, $(0, 1) \cdot (1, 1) = (1, 0)$.

Let $\alpha_1 = (0, 0)$, $\alpha_2 = (1, 0)$, $\alpha_3 = (0, 1)$, $\alpha_4 = (1, 1)$. Then, letting $U_1 = (y_1, y_2)$
and $U_2 = (y_3, y_4)$, we obtain

$$V_j = \alpha_j(y_1, y_2) + (y_3, y_4) \text{ for } 1 \leq j \leq 4,$$

$$V_5 = U_1 = (y_1, y_2).$$

Letting $\alpha_j$ take on the values $(0, 0), (1, 0), (0, 1), (1, 1)$ in turn, we can express the contents of the five storage sites as follows:

$$V_1 = (0, 0) + (y_3, y_4) \equiv (y_3, y_4);$$

$$V_2 = (y_1, y_2) + (y_3, y_4) \equiv (y_1 \oplus y_3, y_2 \oplus y_4);$$

$$V_3 = (y_2, y_1 + y_2) + (y_3, y_4) \equiv (y_2 \oplus y_3, y_1 \oplus y_2 \oplus y_4);$$

$$V_4 = (y_1 + y_2, y_1) + (y_3, y_4) \equiv (y_1 \oplus y_2 \oplus y_3, y_1 \oplus y_4);$$

$$V_5 = (y_1, y_2) + (0, 0) \equiv (y_1, y_2).$$

Given any pair $(V_i, V_j)$, we can recover $y_1$, $y_2$, $y_3$, and $y_4$ by XOR operations. For example, given $V_3$ and $V_4$, we have $(y_2 \oplus y_3) \oplus (y_1 \oplus y_2 \oplus y_3) = y_1; (y_1 \oplus y_2 \oplus y_4) \oplus (y_1 \oplus y_4) = y_2; (y_2 \oplus y_3) \oplus y_2 = y_3; \text{ and } (y_1 \oplus y_4) \oplus y_1 = y_4.$

For larger values of $t$, the values of $V_1, V_2, \ldots, V_{2^t+1}$ can also be expressed in terms of the individual bits $y_1, \ldots, y_{2t}$ using the XOR function, but the expressions become more complicated, and it is simpler to use the Galois-field representation.

**Note:** It is usual to suppose that information is represented in binary, but the upper bound and the constructive lower bound on $n^*$ can both be computed just as before using an alphabet of $q$ symbols, where $q$ is any power of a prime. Thus, it is easily shown that when we have $2t$-ary symbols of information (a prime power) to be recovered from the contents of any two of $n$ sites, each containing $t$ symbols of data, the maximum possible value of $n$ is $n^* = q^t + 1$, achievable by the scheme already outlined for the case $q = 2.$
2.1.2 A More General Distributed Information Problem

Now let us consider a more general problem of distributing data. In the last section we had $2t$ symbols of information to be reconstructed from any pair of data storage sites containing $t$ symbols each. Now suppose that we have $kt$ symbols of information and $n$ storage sites with $t$ symbols of data each. Find $n^*$, the maximum possible number of sites such that the contents of any $k$ sites suffice to reconstruct the original information.

When $k = 2$, we simply have the problem posed at the beginning of the last section, so $n^* = q^t + 1$, when $q$, the size of the symbol alphabet, is a prime power. For general $k$, we shall find that the construction of $V_1, V_2, \ldots, V_n$ can be readily adapted to allow $n = q^t + 1$ as before. However, the upper bound requires more care. The upper-bounding technique of the last section does not generalize readily to larger values of $k$, but a similar bound can be obtained by considering appropriately defined orthogonal Latin squares when $t = 2$ and orthogonal Latin hypercubes when $t \geq 3$. (See Singleton [65], Mann [45].) This technique was used by Singleton to find properties of maximum-distance-separable error-correcting codes. The arguments of Mann and Singleton yield the bound

$$n^* \leq q^t + k - 1. \quad (2.2)$$

We shall see shortly that a straightforward construction yields $n = q^t + 1$. Thus

$$q^t + 1 \leq n^* \leq q^t + k - 1 \quad (2.3)$$

for general $k$, with $q$ any prime power. When $k > 2$, there is a slight gap between the upper and lower bounds on $n^*$.

We now demonstrate that $n^* \geq q^t + 1$. As in the previous section, we break the information vector $(y_1, \ldots, y_{kt})$ into blocks of $t$ symbols each, and identify each $t$-symbol block with an element of $GF(q^t)$, the Galois field with $q^t$ elements. Call the $k$ field elements $U_1, U_2, \ldots, U_k$. Refer to the $q^t$ elements of $GF(q^t)$ as $\alpha_1, \alpha_2, \ldots, \alpha_{q^t}$. Referring to the contents of the $j^{th}$ storage site as $V_j$, where $V_j \in GF(q^t)$, we let

$$V_j = U_1 + \alpha_j U_2 + \alpha_j^2 U_3 + \ldots + \alpha_j^{k-1} U_k \quad (2.4)$$
CHAPTER 2. SINGLE-USER DISTRIBUTED INFORMATION STORAGE

\[ V_{q^i+1} = U_k. \] (2.5)

Given the contents of any \( k \) storage sites \( V_{j_1}, V_{j_2}, \ldots, V_{j_k} \), we have a system of \( k \) linear equations over \( GF(q^i) \) in the \( k \) unknowns \( U_1, U_2, \ldots, U_k \). Because the matrix of coefficients is a Vandermonde matrix (see Blahut [8]), it follows immediately that \( U_1, \ldots, U_k \) can be determined uniquely, as desired.

A Computational Note

In the last subsection we showed that it is possible to distribute information among a large number of sites and to reconstruct the information perfectly with the minimum conceivable number of available data storage sites. Although the results are of theoretical interest, they are much stronger than is required in practice. With just one kilobit (1024 bits) of information to store, the construction demonstrated earlier would allow us to have \( 2^{512}(> 10^{150}) \) data storage sites with 512 bits each, with only two of the sites needed to allow perfect reconstruction of the original information. It is unreasonably pessimistic, however, to assume that such a small fraction of the storage sites would be accessible at any one time. Nonetheless, the Galois-field technique for data distribution described earlier can be used to advantage in real systems.

Rather than taking the capacity of each disk as given and trying to determine how many disks we can have, let us take the number of disks, \( n \), as given and then determine how large a finite field we need. The minimum allowable size of the finite field will correspond to the minimum capacity \( C \) needed on each disk so that any \( k \) of \( n \) disks will suffice to recover \( kC \) bits of information without error.

We see from our previous results that for reliable reconstruction of the original information, it suffices to have \( 2^C + 1 \geq n \). Therefore it suffices to have \( C \geq \log_2 n \). For practical problems, \( C \gg \log_2 n \), so the constraint is easily met. However, we can greatly reduce the computational complexity of recovering information if we break each disk into \( r \)-bit "chunks," where \( r = [\log_2 n] \) or \( [\log_2 n] + 1 \). We associate a finite field element from \( GF(2^r) \) with each chunk and store appropriate linear combinations of these elements on the \( n \) disks.

**Example:** See Figure 2.3 for an example with \( k = 2, n = 6, \) and \( C = 1000 \) bits.
There are 6 (= n) disks in all, each with capacity 1,000 bits. We wish to store 2,000 (= kC) bits of information in the network of disks such that the contents of any 2 (= k) disks suffice to recover the information. Then

\[ \lceil \log_2 n \rceil = \lceil \log_2 6 \rceil = 3. \]

1,000 = (333)3 + 1 = (332)3 + (1)4

Therefore, using the information-distribution technique described earlier, we can break each disk into 3- and 4-bit chunks, thereby decomposing the original problem (which required computations in \( GF(2^{1,000}) \)) into 333 subproblems involving computations in \( GF(2^3) \) and \( GF(2^4) \).

\[ \text{Disk 1: } \begin{array}{ccccccc}
xxx & xxx & xxx & \cdots & xxx & xxx & xxx
\end{array} \\
\text{Disk 2: } \begin{array}{ccccccc}
xxx & xxx & xxx & \cdots & xxx & xxx & xxx
\end{array} \\
\vdots
\text{Disk 6: } \begin{array}{ccccccc}
xxx & xxx & xxx & \cdots & xxx & xxx & xxx
\end{array} \\
\hline
332 \text{ 3-bit chunks}
\]

Figure 2.3: Distributed information storage with small chunks

In general, we see that we can break each disk into small segments and solve a number of linear algebra problems over small fields rather than solving a single problem over an enormous field. Doing field arithmetic by table lookup, we can reconstruct the first r-bit chunk of the original information in \( O(k^3 r) \) operations by inverting a matrix over \( GF(2^r) \).

Once we have computed the inverse matrix, we can reconstruct each of the other chunks in \( O(k^2 r) \) operations by a single matrix-vector multiplication. Therefore, when
\( C \gg r \approx \log_2 n \), the average number of operations required to reconstruct an \( r \)-bit chunk is \( O(k^2 r) \).

Equivalently, we can reconstruct information in \( O(k^2) \) operations per bit, on average, where \( k \) is the minimum number of sites guaranteed to be accessible at any given time. In contrast, the original storage technique (using arithmetic in the field \( GF(2^C) \)) requires a number of operations proportional to \( C \), where \( C \) can easily be in the millions of bits per site.

## 2.2 Disks of Unequal Capacities

### 2.2.1 Derivation of Capacity with Disk Failures

Until now we have assumed that all storage sites in a network have the same storage capacity, in which case the failure of a site is equivalent to the erasure of a single symbol in an error-control code over a very large alphabet. In practice, however, different sites might have different storage capacities. In this case, the problem of recovering from one or more site failures can no longer be solved directly with an erasure-correcting code. However, as we shall see in the following example, a simple modification allows us to handle this more general case as well.

**Example:**

Suppose that we have access to a set of four different disks with capacities 100, 150, 180, and 200 bits (or kilobits, or megabits, or gigabits). Assuming that at least 3 of the 4 disks are accessible at any given time, how much information can be recovered reliably?

**Solution:**

Clearly, if the disk with capacity 200 is inaccessible, we cannot hope to recover more than \( 100 + 150 + 180 = 430 \) bits of information. On the other hand, it is possible to **achieve** this upper bound as follows: Letting the information bits be represented by the vector \( \mathbf{y} = (y_1, y_2, \ldots, y_{430}) \), we store \( (y_1, y_2, \ldots, y_{100}) \).
on Disk 1, \((y_{101}, \ldots, y_{250})\) on Disk 2, and \((y_{251}, \ldots, y_{430})\) on Disk 3. Now we pad the subvectors stored at each site with trailing zeros so that each site is associated with a 180-bit vector, which we identify with an element of the finite field \(GF(2^{180})\). Let \(U_1, U_2,\) and \(U_3\) be the field elements associated with Disks 1, 2, and 3, respectively. Now on Disk 4, we store the field element \(V_4 = U_1 + U_2 + U_3\), that is, the element associated with the bitwise XOR of the (padded) contents of the other three sites. Thus on Disk 4, we store the binary vector \((y_1 \oplus y_{101} \oplus y_{251}, \ldots, y_{99} \oplus y_{199} \oplus y_{349}, y_{100} \oplus y_{200} \oplus y_{350}, y_{201} \oplus y_{351}, y_{202} \oplus y_{352}, \ldots, y_{250} \oplus y_{400}, y_{401}, y_{402}, \ldots, y_{430})\).

The final 20 bits on Disk 4 can be chosen arbitrarily. Now, given the contents of any 3 of the 4 sites, we can reconstruct \((y_1, y_2, \ldots, y_{430})\) using simple XOR operations.

More generally, we can use the technique illustrated in the last example to recover the information stored in any network of \(n\) storage sites, at least \(k\) of which are assumed to be accessible at any given time. For simplicity, let us continue to call the storage sites “disks.” Without loss of generality, label the disks Disk 1, \ldots, Disk \(n\) in increasing order of storage capacities; i.e., if Disk \(j\) has capacity \(C_j\) bits, then

\[C_1 \leq C_2 \leq \ldots \leq C_n.\]

Clearly, the maximum amount of information that we can hope to recover reliably from the network is

\[C_{\text{max}} = C_1 + \ldots + C_k,\]

because it might be that only the \(k\) disks with smallest capacities are accessible. On the other hand, we can actually achieve \(C_{\text{max}}\) using the technique demonstrated in the last example. In general, we allocate \(C_k\) bits per disk, where \(k\) is the minimum number of disks assumed to be accessible. If \(C_j > C_k\) for some Disk \(j\), then we leave the last \((C_j - C_k)\) bits of Disk \(j\) unused. If \(C_j < C_k\) for some Disk \(j\), then we think of extending the \(j\)th disk by padding it with \((C_k - C_j)\) virtual trailing zeros.

Recall that the disks are labeled such that \(C_1 \leq C_2 \leq \ldots \leq C_k \leq \ldots \leq C_n\). Then for \(1 \leq j \leq k\), the contents of Disk \(j\) are padded with \((C_k - C_j)\) virtual zeros.
so that Disk 1 through Disk $k$ are each associated with a binary vector of length $C_k$. Each of these $k$ binary vectors is in turn identified with an element of the field $GF(2^{C_k})$; call this field element $V_j$, where $1 \leq j \leq k$. Now, provided that $2^{C_k} \geq n$, we can use a slightly modified version of the Galois-field-based information storage technique presented earlier to store the appropriate field elements $V_{k+1}, V_{k+2}, \ldots, V_n$ on the remaining disks.

Recall that in Section 2.1.2, for $1 \leq j \leq n$, we stored the field element $V_j$ at Site $j$, where

$$V_j = U_1 + \alpha_j U_2 + \alpha_j^2 U_3 + \ldots + \alpha_j^{k-1} U_k,$$

and where the $\alpha_j$'s were distinct elements of the field $GF(q^t)$. Here we take $q = 2$ (binary alphabet) and $t = C_k$ (the number of bits per disk).

In Section 2.1.2, we were able to identify the field elements $U_1, \ldots, U_k$ with the $k$ subvectors of the information vector $y$. In the present case, we must modify our approach because the field elements $V_1, \ldots, V_{k-1}$ cannot take on arbitrary values; their associated binary vectors must have the appropriate number of trailing zeros. However, beginning with $V_1, \ldots, V_k$ as information symbols, we can work backwards to determine $U_1, \ldots, U_k$ as follows:

$$A = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ \alpha_1 & \alpha_2 & \cdots & \alpha_k \\ \alpha_1^2 & \alpha_2^2 & \cdots & \alpha_k^2 \\ \vdots & \vdots & & \vdots \\ \alpha_1^{k-1} & \alpha_2^{k-1} & \cdots & \alpha_k^{k-1} \end{pmatrix}$$

$$(V_1, \ldots, V_k) = (U_1, \ldots, U_k)A^{-1},$$

Thus

$$(U_1, \ldots, U_k) = (V_1, \ldots, V_k)A^{-1},$$

where the indicated matrix $A$ is Vandermonde and thus has an inverse. Then for $1 \leq j \leq n$,

$$V_j = (U_1, \ldots, U_k)(1, \alpha_j, \alpha_j^2, \ldots, \alpha_j^{k-1})^t$$

$$= (V_1, \ldots, V_k)A^{-1}(1, \alpha_j, \alpha_j^2, \ldots, \alpha_j^{k-1})^t.$$
CHAPTER 2. SINGLE-USER DISTRIBUTED INFORMATION STORAGE

As in Section 2.1.2, it follows that if \( n \leq 2^{C_k} \), then any \( k \) of the field elements \( V_1, \ldots, V_n \) can be used to reconstruct \( U_1, \ldots, U_k \) and thus to reconstruct \( V_1, \ldots, V_k \), which contain the \( (C_1 + C_2 + \ldots + C_k) \) bits of information. Since we have already established \( (C_1 + \ldots + C_k) \) as an upper bound on the number of bits that can be reliably recovered, we have the following definition and theorem.

Definition: Given \( n \) disks with capacities \( C_1, \ldots, C_n \) bits, with at least \( k \) of the disks guaranteed to be accessible at any given time, let the capacity of the network be the maximum amount of information that can be stored and reliably recovered from the network of disks.

\[ \blacksquare \]

**Theorem 1** Given \( n \) disks \( D_1, \ldots, D_n \) with respective capacities \( C_1, \ldots, C_n \) bits, where \( C_1 \leq C_2 \leq \ldots \leq C_n \), the capacity of the network under the assumption that at least \( k \) of the \( n \) disks are always accessible is \( (C_1 + C_2 + \ldots + C_k) \) bits, provided that \( n \leq 2^{C_k} \).

\[ \blacksquare \]

### 2.2.2 Computational Considerations

The construction just given requires us to do arithmetic in a field of \( 2^{C_k} \) elements. In realistic examples, \( C_k \) can easily be in the millions of bits, but it is completely impractical to work with elements of the field \( GF(2^{1,000,000}) \). Instead, we can work in the field \( GF(2^r) \), where \( r = \lceil \log_2 n \rceil \) or \( \lfloor \log_2 n \rfloor + 1 \); that is, \( r \) is (essentially) the smallest exponent such that \( 2^r \geq n \).

In fact, we can decompose our problem into \( r \)-bit chunks exactly as in before. It then follows as before that we can reconstruct information in \( O(k^2) \) operations per bit, on average, where \( k \) is the minimum number of disks guaranteed to be accessible at any given time. Even with disks of enormous capacity—say, \( 10^{12} \) bits—we can reconstruct information rapidly by focusing on a relatively small segment from each disk. For example, if there are no more than 256 disks in all (i.e., if \( n \leq 256 \)), then we can work with byte-sized pieces, since \( \log_2 256 = 8 \).
Chapter 3

Multiple Users with Common Information

3.1 Identical Information

In the last section we generalized the results of Section 2.1 by allowing different disks to have different capacities. There is another interesting direction in which we can extend the results of Section 2.1. Rather than assuming that there is a single user with access to all $n$ disks (some of which may be inaccessible at any given time), we now assume the existence of many users, each with access to a different subset of the $n$ disks. As in the last chapter, Galois fields will play an important role in the storage and reconstruction of information.

For simplicity, we assume throughout this chapter that each disk has the same storage capacity, $C$ bits. For completeness, we list below some notation used throughout the chapter.

Notation: We suppose that there are $m$ users $(U_1, \ldots, U_m)$ and $n$ disks $(D_1, \ldots, D_n)$ in all. We further suppose that each disk has capacity $C$ bits and that for $1 \leq i \leq m$, User $i$ is connected to $n_i$ disks, at least $k_i$ of which are accessible at any given time.

One of the simplest examples occurs when each user is connected to the same number of disks and when each user is able to tolerate the same number of link or
disk failures. Then we have the following class of problems:

**Problem 1:** In addition to the assumptions already listed, suppose that \( k_1 = k_2 = \ldots = k_m \) and that \( n_1 = n_2 = \ldots = n_m \). Find the maximum amount of common information that can be recovered by all \( m \) users. (See Figure 3.1 for an example with \( m = 6, C = 100, n = 6, k_i = 2 \) for all \( i \), and \( n_i = 3 \) for all \( i \).)

![Diagram](image)

**Figure 3.1:** A symmetric information storage example

**Solution:** In the particular example shown in Figure 3.1, it is clear that each user can hope to reliably recover at most 200 bits of information. More generally, each User \( i \) can at best hope to recover \( k_i C \) bits of information reliably, where we have assumed that \( k_i = k_1 \) for \( 1 \leq i \leq m \).

Furthermore, provided that \( 2^C > n \), there is a very simple way to achieve this upper bound of \( k_1 C \) bits. Let \( y = (y_1, y_2, \ldots, y_{k_1}C) \) be the binary information vector to be reconstructed. It clearly suffices to find an information storage scheme such that the contents of any \( k_1 \) of the \( n \) disks (in the figure, any 2 of the 6 disks) determine \( y \). However, we have already seen such a scheme in Chapter 2. We break each disk into \( r \)-bit chunks (where \( r \geq \log_2 n \)) and store Galois field elements \( V_1, V_2, \ldots, V_n \) at the \( n \) storage sites, where each \( V_j \in GF(2^r) \) and where the \( V_j \)'s are computed as in Chapter 2.
Now for $1 \leq i \leq m$, User $i$ can use linear algebra to reconstruct the information vector $y$ from the contents of any $k_i$ of the $n$ disks, hence from any $k_i$ of the $n_1$ disks to which he is connected. \hfill \Box

**Problem 2:** Until now, we have assumed that each user has access to the same number of disks. Now let us drop this restriction; i.e., if User $i$ is connected to $n_i$ disks, any $k_i$ of which are accessible at any given time, let us allow $k_i$ and $n_i$ to vary for $1 \leq i \leq m$. Find the maximum amount of (common) information that can be reliably recovered by all $m$ users. (See the Figure 3.2 for an example with $m = 3, n = 4, C = 100, k_1 = 2, n_1 = 3, k_2 = 1, n_2 = 2, k_3 = 2, n_3 = 2$.)

![Figure 3.2: An asymmetric information storage example](image)

**Solution:** In Figure 3.2, it is clearly impossible for User 2 to recover more than 100 bits of information reliably. Therefore, if there is some binary information vector $y$ to be reconstructible by all three users, it can have at most 100 bits.

Conversely, it is clear that we can simply store the same 100-bit information vector $y = (y_1, \ldots, y_{100})$ on each of the four disks, in which case each user can trivially recover $y$.

More generally, it is clear that the length of $y$ is bounded above by $k_{\text{min}} C$, where $k_{\text{min}} \triangleq \min \{k_i\}$. Conversely, assuming that $C \geq \log_2 n$, we can actually achieve this upper bound in a straightforward manner. For $1 \leq i \leq m$, at any given time, User $i$ has access to $k_i \geq k_{\text{min}}$ of the disks in some $n_i$-member subset.
of the \( n \) disks. We just use the Galois-field-based storage scheme of Chapter 2, which allows each user to reconstruct \( k_{\min}C \) bits of information given the contents of any \( k_{\min} \) of the \( n \) disks. When \( C \gg \log_2 n \), the \( C \) bits on each disk can be divided into blocks of \( \lfloor \log_2 n \rfloor \) or \( (\lfloor \log_2 n \rfloor + 1) \) bits each so that the computations involved take place over a (relatively) small Galois field.

### 3.2 Sequentially Refined Information

The problems considered in the preceding section can be generalized. In Problem 2 of the last section, we saw that each user could reliably recover the same \( k_{\min}C \) bits of information, where \( k_{\min} \) was the minimum number of disks to which (any) user could ever have access at any given time. However, if \( k_i \gg k_{\min} \) for some \( i \), then User \( i \) might hope to recover more information – up to \( k_iC \) bits. Thus we consider the following problem of sequential refinement, in which users with guaranteed access to many disks (i.e., users with large values of \( k_i \)) can reconstruct all of the information available to users who are not so well connected (i.e., users with small values of \( k_i \)), together with some additional information.

**Sequential Refinement Problem**: Assume as before that we have a network with \( m \) users and \( n \) disks, each disk with capacity \( C \) bits. Also assume that for \( 1 \leq i \leq m \), User \( i \) is connected to \( n_i \) disks, at least \( k_i \) of which are accessible at any given time. Suppose that the users wish to recover (subsets of) an arbitrary information vector \( y = (y_1, y_2, \ldots, y_{k_{\max}C}) \).

We say that *complete sequential refinement* of information is possible if data can be stored on the \( n \) disks such that for \( 1 \leq i \leq m \), User \( i \) can reliably recover \((y_1, y_2, \ldots, y_{k_iC})\), the first \( k_iC \) bits of information.

Equivalently, let \((i_1, i_2, \ldots, i_m)\) be a permutation of the user indices \((1, 2, \ldots, m)\) such that \( k_{\min} \leq k_{i_1} \leq k_{i_2} \leq \ldots \leq k_{i_m} \). Then, if *complete sequential refinement* of information is achievable, User \( i_1 \) may be thought of as a degraded version of User \( i_2 \), who is in turn a degraded version of User \( i_3 \), etc.
3.2.1 No Disk or Link Failures

In this subsection and the next, we shall derive necessary and sufficient conditions for a network to admit complete sequential refinement of information. In this subsection we consider the special case in which no disk or link failures occur; i.e., we assume that \( k_i = n_i \) for \( 1 \leq i \leq m \).

Positive and Negative Examples

Let us begin by giving a simple example in which complete sequential refinement of information is possible.

**Positive Example:** See Figure 3.3 for a setup in which complete sequential refinement is possible.

Since \( k_{\text{max}} = 2 \), we consider an information vector \( y = (y_1, \ldots, y_{2C}) = (w_1, w_2) \),

![Positive Example Diagram]

Figure 3.3: An example in which sequential refinement is possible

where \( w_1 \triangleq (y_1, \ldots, y_C) \) and \( w_2 \triangleq (y_{C+1}, \ldots, y_{2C}) \). In order to achieve complete sequential refinement of information (or just ‘sequential refinement’ for short), User 1 should be able to recover \( w_1 \) (the first \( C \) bits of \( y \)) while each of the other three users should be able to recover \( w_1 \) and \( w_2 \) (the first, and only, \( 2C \) bits of \( y \)).
The information storage scheme given in Figure 3.4 shows that sequential refinement is indeed possible in the present example.

![Figure 3.4: Realization of sequential refinement](image)

Although sequential refinement is sometimes possible, it is not achievable for all networks, as the following example shows.

**Negative Example:** See Figure 3.5 for a setup in which complete sequential refinement is *not* possible.

In the current example, $k_{\text{max}} = 2$ once again, so we consider an information

![Negative Example:](image)

**Figure 3.5:** An example in which sequential refinement is not possible

vector $y$ of length $2C$ bits. As in the previous example, we let $w_1$ be the vector
of the first \( C \) information bits and \( \mathbf{w}_2 \) the vector of the remaining \( C \) information bits. Then User 1 must be able to recover \( \mathbf{w}_1 \) from the contents of Disk 1. It follows that the \( 2^C \) possible states of Disk 1 must be in one-to-one correspondence with the \( 2^C \) possible values of \( \mathbf{w}_1 \). Thus, representing the contents of Disk \( j \) by the binary vector \( \mathbf{v}_j \), we must have \( \mathbf{v}_1 = f_1(\mathbf{w}_1) \) for some invertible function \( f_1 : 2^C \to 2^C \).

In the same way, User 3 must be able to recover \( \mathbf{w}_1 \) from \( \mathbf{v}_2 \), the contents of Disk 2. Therefore, \( \mathbf{v}_2 = f_3(\mathbf{w}_1) \) for some invertible function \( f_3 : 2^C \to 2^C \).

Finally, User 2 must be able to recover the pair \( (\mathbf{w}_1, \mathbf{w}_2) \) from the pair \( (\mathbf{v}_1, \mathbf{v}_2) \). Reasoning as before, we see that \( (\mathbf{v}_1, \mathbf{v}_2) = f_2(\mathbf{w}_1, \mathbf{w}_2) \) for some invertible function \( f_2 : 2^{2C} \to 2^{2C} \).

However, we have already seen that \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) must each be invertible functions of \( \mathbf{w}_1 \) and thus invertible functions of each other. Therefore, any information storage scheme that allows User 1 and User 3 each to recover \( \mathbf{w}_1 \) can have only \( 2^C \) possible values of the pair \( (\mathbf{v}_1, \mathbf{v}_2) \). Since there are \( 2^{2C}(>2^C) \) possible values of \( (\mathbf{w}_1, \mathbf{w}_2) \), it follows that \( (\mathbf{v}_1, \mathbf{v}_2) \) and \( (\mathbf{w}_1, \mathbf{w}_2) \) cannot be put into one-to-one correspondence.

Therefore User 2 cannot reliably recover all \( 2^C \) bits of information. We conclude that sequential refinement is impossible for this example.

**Necessary and Sufficient Conditions**

Having seen a network for which sequential refinement is possible and a network for which it is not, let us use the ideas illustrated in the examples of the last subsection to derive necessary and sufficient conditions for sequential refinement. We assume in this section that \( k_i = n_i \) for \( 1 \leq i \leq m \); that is, we assume that no disk or link failures occur.

Let us begin by deriving necessary conditions for sequential refinement. The following definitions are motivated by the analysis of the negative example given
earlier.

**Definition:** Let $E$ be the set of *edges* between users and disks; i.e.,

$$E \triangleq \{(i, j) : \text{User } i \text{ is connected to Disk } j\}.$$

**Definition:** For $1 \leq j \leq n$, associate with Disk $j$ the *disk degree*

$$d_j \triangleq \min_{i : (i, j) \in E} \{k_i\}.$$

We shall see later that for sequential refinement, each $v_j$ can depend only on $w_1, \ldots, w_{d_j}$. See Figure 3.6 for an example illustrating the definition of the disk degree $d_j$.

![Diagram](image)

Figure 3.6: An example illustrating the definition of $d_j$

In Figure 3.6, we see that $k_{\max} = \max\{2, 3, 3, 1\} = 3$, so no user can hope to recover more than $3C$ bits of information. As in the previous examples, we assume that the users wish to reliably recover (subsets of) a binary information vector $y$, where $y = (y_1, y_2, \ldots, y_{3C})$. Let

$$w_1 \triangleq (y_1, \ldots, y_C),$$

$$w_2 \triangleq (y_{C+1}, \ldots, y_{2C}),$$

$$w_3 \triangleq (y_{2C+1}, \ldots, y_{3C}).$$
Then $U_1$ wishes to recover the pair $(w_1, w_2)$ from the pair $(v_1, v_2)$; $U_2$ wishes to recover the triple $(w_1, w_2, w_3)$ from the triple $(v_1, v_2, v_3)$; $U_3$ wishes to recover the triple $(w_1, w_2, w_3)$ from the triple $(v_2, v_3, v_4)$; and $U_4$ wishes to recover $w_1$ from $v_4$.

Notation: If $a$ is a deterministic function of $b$, then we write $a \leftarrow b$. If $a$ is an invertible function of $b$, then we may also write $a \leftrightarrow b$.

Arguing as in the earlier negative example, we see that $v_4$ must be an invertible function of $w_1$. Thus we write

$$v_4 \leftarrow w_1.$$ Similarly, by considering User 1, we see that $(v_1, v_2)$ must be an invertible function of $(w_1, w_2)$. Therefore $v_1$ and $v_2$ must individually be deterministic functions of $(w_1, w_2)$, so we write

$$v_1 \leftarrow (w_1, w_2) \quad v_2 \leftarrow (w_1, w_2).$$

Considering User 2, we see that

$$v_1 \leftarrow (w_1, w_2, w_3); \quad v_2 \leftarrow (w_1, w_2, w_3); \quad v_3 \leftarrow (w_1, w_2, w_3).$$

Finally, considering User 3, we see that

$$v_2 \leftarrow (w_1, w_2, w_3); \quad v_3 \leftarrow (w_1, w_2, w_3); \quad v_4 \leftarrow (w_1, w_2, w_3).$$

Now shifting attention to individual disks rather than individual users, we see by considering Disk 1 that

$$v_1 \leftarrow (w_1, w_2) \quad \text{and} \quad v_1 \leftarrow (w_1, w_2, w_3).$$

Clearly the second relation above is implied by the first, so we just note that

$$v_1 \leftarrow (w_1, w_2). \quad (3.1)$$

Now considering Disk 2, we see that

$$v_2 \leftarrow (w_1, w_2); \quad v_2 \leftarrow (w_1, w_2, w_3); \quad v_2 \leftarrow (w_1, w_2, w_3).$$
Thus
\[ v_2 \leftarrow (w_1, w_2). \] (3.2)

Proceeding in the same way for Disks 3 and 4, we finally obtain the following set of relations:
\[ v_1 \leftarrow (w_1, w_2); \quad v_2 \leftarrow (w_1, w_2); \quad v_3 \leftarrow (w_1, w_2, w_3); \quad v_4 \leftarrow w_1. \] (3.3)

Thus, for \( 1 \leq j \leq 4 \), we have
\[ v_j \leftarrow (w_1, \ldots, w_d) \]
for the example being considered. We shall see shortly that, in general,
\[ v_j \leftarrow (w_1, \ldots, w_d) \] for \( 1 \leq j \leq n. \)

Now let us generalize the previous arguments and consider an arbitrary network. Suppose that for some \( i \) with \( 1 \leq i \leq m \), User \( i \) is connected to \( k_i \) different disks. As before, let \( k_{\max} = \max_{1 \leq i \leq m} \{ k_i \}. \)

**Lemma 1** Suppose that a network admits sequential refinement. Then for each User \( i \), for any Disk \( j \) connected to User \( i \) (that is, for any \( j \) such that \((i, j) \in E\)), the binary vector \( v_j \) stored on Disk \( j \) is a function only of the first \( k_i \) information vectors \( w_1, \ldots, w_{k_i} \).

**Proof:** In order to simplify the notation, we assume that the User \( i \) is fixed, so we do not explicitly write the subscript \( i \). Thus User \( i \) is connected to \( k \) different disks; call them \( D_{j_1}, \ldots, D_{j_k} \). It follows that the \( k \)-tuple of \( C \)-element vectors \((v_{j_1}, \ldots, v_{j_k})\) must be an invertible function of \((w_1, \ldots, w_k)\). Thus if \((i, j) \in E\), we must have
\[ v_j \leftarrow (w_1, \ldots, w_{k_i}). \]

Therefore, for each Disk \( j \) such that \((i, j) \in E\), the vector \( v_j \) must be a function of the first \( k_i \) information subvectors \( w_1, \ldots, w_{k_i} \). \( \blacksquare \)
Examining Lemma 1, fixing attention on a particular Disk $j$, and considering every User $i$ attached to that disk, we see that $\mathbf{v}_j$ can depend only on the first $d_j$ information subvectors $\mathbf{w}_1, \ldots, \mathbf{w}_{d_j}$, where $d_j = \min_{i:j(i,j) \in E} \{k_i\}$.

Thus we have Lemma 2.

**Lemma 2** Suppose that $k_i = n_i$ for $1 \leq i \leq m$. For $1 \leq j \leq n$, if $\mathbf{v}_j$ is the binary vector stored on Disk $j$ and if $d_j = \min_{i:j(i,j) \in E} \{k_i\}$ is the minimum number of disks accessible by any user with access to Disk $j$, then $\mathbf{v}_j$ must be a function only of the first $d_j$ information subvectors $\mathbf{w}_1, \ldots, \mathbf{w}_{d_j}$.

(Note: $\mathbf{v}_j$ does not necessarily depend on all of the first $d_j$ information subvectors; the lemma simply implies that $\mathbf{v}_j$ does not depend on $\mathbf{w}_t$ for any $t > d_j$.)

Lemma 2 leads to the following condition for a network to admit sequential refinement.

**Theorem 2** In order for a network to (admit) sequential refinement, it is necessary that

$$k_i \leq \max_{j:j(i,j) \in E} d_j \quad \text{for} \quad 1 \leq i \leq m,$$

where $k_i$ is the number of disks accessible by User $i$, and where the disk degree $d_j$ is the minimum $k_i$ for any User $i$ connected to Disk $j$.

**Proof:** For sequential refinement, User $i$ must be able to recover the $k_i C$ bits of information contained in $(\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_{k_i})$. Therefore at least one of the disks to which User $i$ has access must contain information about the $k_i$th subvector, $\mathbf{w}_{k_i}$. By the previous lemma, we must have $d_j \geq k_i$ for some $j$ such that User $i$ has access to Disk $j$.

More formally, we see from Lemma 2 that

$$\mathbf{v}_j \leftarrow (\mathbf{w}_1, \ldots, \mathbf{w}_{d_j}) \quad \text{for} \quad 1 \leq n.$$

As in Lemma 1, consider an arbitrary but fixed User $i$; for simplicity, we do not explicitly write the subscript $i$ in the following argument. Now User $i$ is
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

connected to the data vectors \( v_{j1}, \ldots, v_{jk} \), where \( k = k_i \).

For sequential refinement, \((w_1, \ldots, w_k)\) must be an invertible function of \((v_{j1}, \ldots, v_{jk})\).
But each \( v_j \) is in turn a function of \((w_1, \ldots, w_{d_j})\), by Lemma 2.

Thus for each User \( i \), we have

\[
(w_1, \ldots, w_{k_i}) \leftrightarrow (w_1, \ldots, w_d),
\]

where

\[
d = \max_{j: (i, j) \in E} d_j.
\]

In particular,

\[
w_{k_i} \leftrightarrow (w_1, \ldots, w_d).
\]

Now, since the \( w \)'s are mutually independent, we must have \( k_i \leq d \) in order for \( w_{k_i} \) to be a function of \((w_1, \ldots, w_d)\).

The theorem follows immediately. ■

The theorem above can be restated by using the following definition.

**Definition:** For \( 1 \leq i \leq m \) and \( 1 \leq s \leq k_i \), let \( N_i(s) \) be the number of disks \( D_j \)
attached to User \( i \) that satisfy \( d_j \leq s \). ■

In the previous definition with \( s = k_i \), note that \( N_i(k_i) = k_i \) for all \( i \), because every Disk \( D_j \) attached to User \( i \) satisfies the inequality \( d_j \leq k_k \) by the definition of \( d_j \). Using the definition of \( N_i(s) \) above, we can restate Theorem 2 as follows.

**Corollary:** In order for a network to admit sequential refinement, it is necessary that for \( 1 \leq i \leq m \),

\[
N_i(s) \leq s \text{ when } s = k_i - 1.
\]

In fact, this corollary gives just one of a family of necessary conditions that must all hold in order for a network to admit sequential refinement. More generally, we have the following theorem.
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

Theorem 3 In order for a network to admit sequential refinement, it is necessary that \( N_i(s) \leq s \) for \( 1 \leq i \leq m \), for \( 1 \leq s \leq k_i - 1 \).

Proof: We have already (shown, proved) that the condition is necessary for \( s = k_i - 1 \). Now suppose that, in contradiction \{of\} the assertion made in this theorem, \( N_i(s) \geq s + 1 \) for some \( s \in \{1, 2, \ldots, k_i - 2\} \), for some \( i \in \{1, 2, \ldots, m\} \). Then for some \( s \), at least \( s + 1 \) of the disks \( D_j \) attached to User \( i \) have \( d_j \leq s \). Therefore, at least \( s + 1 \) of the disks \( D_j \) contain information that depends only on the information subvectors \( w_1, w_2, \ldots, w_s \). Then at most \( k_i - (s + 1) \) of the disks contain information about the remaining \((k_i - s)\) information subvectors, \( w_{s+1}, w_{s+2}, \ldots, w_{k_i} \). But (now) there are \( 2^{\binom{k_i}{s}} \) different possible values of the information \((w_{s+1}, \ldots, w_{k_i})\) and at most \( 2^{\binom{k_i}{s+1}} \) different possible states of the disks that depend on \( w_{s+1}, \ldots, w_{k_i} \).

Since \( k_i - (s + 1) < k_i - s \), we see that User \( i \) cannot always recover the information \((w_{s+1}, \ldots, w_{k_i})\) reliably, hence the network does not admit sequential refinement. It follows immediately that the condition \( N_i(s) \leq s \) must hold for each \( s \) in order for sequential refinement to be possible.

Interpretation of theorem: Consider the example of Figure 3.6. On each Disk \( j \) we can only store information about \( w_1, \ldots, w_d \). Accordingly, let us associate a vector of \( k_{\text{max}} \) \( z \)'s and \( 0 \)'s with each disk, with an \( x \) in the \( t \)th position indicating that \( v_j \) (the information stored on Disk \( j \)) is allowed to depend on \( w_t \) and a \( 0 \) in the \( t \)th position indicating that \( v_j \) is independent of \( w_t \). Then Figure 3.6 may be redrawn as in Figure 3.7.

Now for each User \( i \) \((1 \leq i \leq m)\), we can write out the row vectors of \( z \)'s and \( 0 \)'s from each of the \( k_i \) disks connected to User \( i \). (See Figure 3.8 for a continuation of the previous example.)

Note that for each User \( i \), only the first \( k_i \) columns are nonzero, since \( U_i \) depends only on \( w_1, \ldots, w_{k_i} \). Therefore, we can delete all but the first \( k_i \) columns for each User \( i \) to obtain a \( k_i \times k_i \) matrix of \( z \)'s and \( 0 \)'s. Furthermore, we can
permute the rows of each matrix (thus effectively reordering the disks) so that for each matrix, every row has at least as many $x$’s as the previous row. Thus we obtain Figure 3.9.

Theorem 3 implies that each $k_i \times k_i$ matrix obtained as above must have $x$’s all along the main diagonal in order for sequential refinement to be possible. Since this condition is met in the current example, we see that sequential refinement may be possible. (Recall that the theorem only gives necessary, not sufficient, conditions for sequential refinement.)

The preceding interpretation of Theorem 3 is given to indicate how to prove a converse to the theorem. The matrix notation suggests that we might be able to use linear algebra to store and recover information. In fact, by replacing the $x$’s with appropriate elements of a finite field, we shall see that in all cases of practical interest, the necessary conditions given in Theorem 3 are also sufficient to achieve sequential
Figure 3.9: An example illustrating Theorem 3

refinement.

Before proving a theorem about sufficient conditions for sequential refinement, let us see how we can systematically find an information storage scheme for the example of Figure 3.6 that allows sequential refinement. Our approach will be to work with the Galois field elements \( W_1 \) and \( W_2 \) having the information subvectors \( w_1 \) and \( w_2 \) as their vector representations. (Thus \( W_t \in GF(2^C) \) for \( 1 \leq t \leq 2 = k_{\text{max}} \).) Similarly, for \( 1 \leq j \leq 3 = n \) we shall represent the \( C \)-bit contents of Disk \( j \) by the Galois field element \( V_j \in GF(2^C) \) having \( v_j \) as its vector representation. Finally, we shall try to express each \( V_j \) as a suitable linear combination of \( W_1, W_2, \ldots, W_{k_{\text{max}}} \). In fact, we see by Lemma 2 that \( V_j \) should be a linear combination of \( W_1, W_2, \ldots, W_{d_j} \) only, because \( V_j \) must be independent of \( W_t \) for \( d_j < t \leq k_{\text{max}} \).

We reproduce Figure 3.6 in Figure 3.10, changing the labels to reflect the definitions of \( V_j \) and \( W_t \). The \( \alpha \)'s, \( \beta \)'s, \( \gamma \)'s and \( \delta \)'s are constants from \( GF(2^C) \) to be determined.

From Figure 3.10, we see that

- \( V_1 = (\alpha_1 \alpha_2 0)(W_1 \ W_2 \ W_3)^t \)
- \( V_2 = (\beta_1 \beta_2 0)(W_1 \ W_2 \ W_3)^t \)
- \( V_3 = (\gamma_1 \gamma_2 \gamma_3)(W_1 \ W_2 \ W_3)^t \)
- \( V_4 = (\delta_1 0 \ 0)(W_1 \ W_2 \ W_3)^t \).

Now user \( U_1 \) wishes to recover the information \((W_1, W_2)\) from \((V_1, V_2)\). But

\[
\begin{pmatrix}
V_1 \\
V_2
\end{pmatrix} =
\begin{pmatrix}
\alpha_1 & \alpha_2 \\
\beta_1 & \beta_2
\end{pmatrix}
\begin{pmatrix}
W_1 \\
W_2
\end{pmatrix}.
\]
Figure 3.10: Storing information using linear algebra

Thus if

\[
\begin{pmatrix}
\alpha_1 & \alpha_2 \\
\beta_1 & \beta_2
\end{pmatrix}
\]

is nonsingular, we can let

\[
\begin{pmatrix}
W_1 \\
W_2
\end{pmatrix} = \begin{pmatrix}
\alpha_1 & \alpha_2 \\
\beta_1 & \beta_2
\end{pmatrix}^{-1} \begin{pmatrix}
V_1 \\
V_2
\end{pmatrix}
\]

and recover \((W_1, W_2)\) from \((V_1, V_2)\). Therefore we should choose \(\alpha_1, \alpha_2, \beta_1, \beta_2\) such that

\[
\begin{pmatrix}
\alpha_1 & \alpha_2 \\
\beta_1 & \beta_2
\end{pmatrix}
\]

is nonsingular.

Shifting attention to User 2, we wish to invert the matrix equation

\[
\begin{pmatrix}
V_1 \\
V_2 \\
V_3
\end{pmatrix} = \begin{pmatrix}
\alpha_1 & \alpha_2 & 0 \\
\beta_1 & \beta_2 & 0 \\
\gamma_1 & \gamma_2 & \gamma_3
\end{pmatrix} \begin{pmatrix}
W_1 \\
W_2 \\
W_3
\end{pmatrix}.
\]

Thus we should choose the \(\alpha\)'s, \(\beta\)'s, and \(\gamma\)'s such that

\[
\begin{pmatrix}
\alpha_1 & \alpha_2 & 0 \\
\beta_1 & \beta_2 & 0 \\
\gamma_1 & \gamma_2 & \gamma_3
\end{pmatrix}
\]
is nonsingular.

Similarly, in order for User 3 to recover \((W_1, W_2, W_3)\),

\[
\begin{pmatrix}
\beta_1 & \beta_2 & 0 \\
\gamma_1 & \gamma_2 & \gamma_3 \\
\delta_1 & 0 & 0
\end{pmatrix}
\]

should be nonsingular.

Finally, in order for User 4 to recover \(W_1\), the matrix \((\delta_1)\) should be nonsingular; i.e., \(\delta_1 \neq 0\).

Combining the four conditions above and reordering the rows of the third matrix, we see that sequential refinement can be achieved if the \(\alpha\)'s, \(\beta\)'s, \(\gamma\)'s, and \(\delta\) can be chosen to make the matrices

\[
\begin{pmatrix}
\alpha_1 & \alpha_2 \\
\beta_1 & \beta_2 \\
\gamma_1 & \gamma_2 & \gamma_3
\end{pmatrix},
\begin{pmatrix}
\alpha_1 & \alpha_2 & 0 \\
\beta_1 & \beta_2 & 0 \\
\gamma_1 & \gamma_2 & \gamma_3
\end{pmatrix},
\begin{pmatrix}
\delta_1 & 0 & 0 \\
\beta_1 & \beta_2 & 0 \\
\gamma_1 & \gamma_2 & \gamma_3
\end{pmatrix}, (\delta_1)
\]

all nonsingular.

Note that these four matrices correspond exactly to the four matrices of \(x\)'s and \(0\)'s given in Figure 3.9 with the \(x\)'s replaced by distinct elements of \(GF(2^C)\). Because the network currently being considered satisfies the necessary conditions given in Theorem 3, there is at least some hope that we can choose the \(\alpha\)'s, \(\beta\)'s, \(\gamma\)'s, and \(\delta\) to make all four matrices nonsingular. (In contrast, the example of Figure 3.5 has the matrix \(\begin{pmatrix} x & 0 \\ x & 0 \end{pmatrix}\) associated with User 2, and the matrix \(\begin{pmatrix} \alpha & 0 \\ \beta & 0 \end{pmatrix}\) is clearly singular no matter which elements \(\alpha\) and \(\beta\) are chosen to replace the \(x\)'s.)

Returning to the example of Figure 3.10, we can see by inspection that the following simple assignment makes all four matrices nonsingular:

\[
\begin{align*}
(\alpha_1, \alpha_2) &= (1, 1) \\
(\beta_1, \beta_2) &= (0, 1) \\
(\gamma_1, \gamma_2, \gamma_3) &= (0, 0, 1) \\
(\delta_1) &= (1)
\end{align*}
\]
Therefore, we let \( V_1 = W_1 + W_2, V_2 = W_2, V_3 = W_3, V_4 = W_1 \).

Translating from Galois field arithmetic to operations on \( C \)-bit vectors, we have the assignment shown in Figure 3.11.

\[
\begin{align*}
(w_1, w_2) & \quad \bullet & (w_1, w_2, w_3) & \quad \bullet & (w_1, w_2, w_3) & \quad \bullet & (w_1) & \quad \bullet \\
U_1 & \quad \bullet & U_2 & \quad \bullet & U_3 & \quad \bullet & U_4 & \quad \bullet \\
v_1 = w_1 \oplus w_2 & \quad & v_2 = w_2 & \quad & v_3 = w_3 & \quad & v_4 = w_1 & \\
\end{align*}
\]

Figure 3.11: Realization in \( GF(2) \)

In the example just considered, it was easy to find field elements that made all four matrices nonsingular. In more complicated examples, though, we need a systematic way of verifying that appropriate field elements exist and a way of finding them when they do exist. We exhibit such a systematic approach now, first considering its application to the running example and then deriving a general theorem.

In general, each Disk \( j \) (for \( 1 \leq j \leq n \)) depends only on \( W_1, W_2, \ldots, W_{d_j} \), where \( k_i \) is the number of disks accessible by User \( i \) and where \( d_j = \min_{i,(i,j) \in E} k_i \). Associated with each Disk \( j \) is a vector \( \alpha_j \) with components in \( GF(2^C) \):

\[
\alpha_j = (\alpha_{j,1} \alpha_{j,2} \ldots \alpha_{j,d_j} \underbrace{0 \ldots 0}_{(k_{\max} - d_j) \text{ zeros}}).
\]

In the example, we refer to \( \alpha_1, \alpha_2, \alpha_3, \) and \( \alpha_4 \) as \( \alpha, \beta, \gamma, \) and \( \delta \), respectively. We have

\[
\alpha = (\alpha_1 \alpha_2 0), \quad \beta = (\beta_1 \beta_2 0), \\
\gamma = (\gamma_1 \gamma_2 \gamma_3), \quad \delta = (\delta_1 0 0)
\]

Let us arrange the four vectors in increasing order of \( d_j \). If two or more vectors (equivalently, disks) have the same value of \( d_j \), we order those vectors arbitrarily.
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

We obtain
\[
\delta = (\delta_1 \ 0 \ 0) \quad (d_j = 1),
\]
\[
\alpha = (\alpha_1 \ \alpha_2 \ 0) \quad (d_j = 2),
\]
\[
\beta = (\beta_1 \ \beta_2 \ 0) \quad (d_j = 2),
\]
\[
\gamma = (\gamma_1 \ \gamma_2 \ \gamma_3) \quad (d_j = 3).
\]

Referring to the four matrices on page 40, we see that we must choose \( \delta, \alpha, \beta, \) and \( \gamma \) such that
\[
\text{rank } \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = k_1 = 2, \quad \text{rank } \begin{pmatrix} \alpha \\ \gamma \end{pmatrix} = k_2 = 3,
\]
\[
\text{rank } \begin{pmatrix} \delta \\ \beta \\ \gamma \end{pmatrix} = k_3 = 3, \quad \text{rank}(\delta) = k_4 = 1.
\]

Equivalently, the sets of vectors \( \{ \alpha, \beta \}, \{ \alpha, \beta, \gamma \}, \{ \delta, \beta, \gamma \}, \) and \( \{ \delta \} \) should each be a linearly independent set. We construct the vectors sequentially, in increasing order of \( d_j \).

When choosing the first vector, \( \delta \), we can let \( \delta_1 \) be any nonzero element of \( GF(2^C) \); for simplicity, we take \( \delta_1 = 1 \), so \( \delta = (1 \ 0 \ 0) \). Now we choose \( \alpha = (\alpha_1 \ \alpha_2 \ 0) \) such that \( \delta \) and \( \alpha \) are linearly independent. Clearly we need only choose \( \alpha_2 \neq 0 \) in order for \( \alpha \) to be independent of \( \delta \); the component \( \alpha_1 \) can be arbitrary. For simplicity, we take \( \alpha_1 = 0, \alpha_2 = 1 \). Then
\[
\alpha = (0 \ 1 \ 0).
\]

Having chosen \( \delta \) and \( \alpha \), we next choose \( \beta = (\beta_1 \ \beta_2 \ 0) \) such that \( \{ \alpha, \beta \} \) and \( \{ \delta, \beta \} \) are both linearly independent sets of vectors.

Aside: We need not show that the set \( \{ \delta, \alpha, \beta \} \) is linearly independent, because these three vectors do not all appear together in any of the sets of vectors that are required to be independent. In fact, these three vectors could not all appear together in the same set, because then the user associated with that set of vectors would have access to at least three different disks such that \( d_j \leq 2 \). By Theorem 3, though, sequential refinement is impossible if any user has access to more than two disks with \( d_j \leq 2 \).
Returning to the task of choosing $\beta$, we see that any vector $\beta$ with $\beta_2 \neq 0$ will be linearly independent of $\delta$. In fact, there are $2^{2C}$ ways to choose $(\beta_1 \beta_2 0)$, and only $2^C$ of these ways are forbidden because they would make $\beta$ a multiple of $\delta$. Similarly, only $2^C$ of the possible choices of $\beta$ are forbidden, since $\beta$ may not be a multiple of $\alpha$ and there are just $2^C$ such multiples. Therefore, provided that $2^{2C} > 2^C + 2^C = 2^{C+1}$, it follows that there must be at least one choice of $\beta$ such that $\{\delta, \beta\}$ and $\{\alpha, \beta\}$ are each linearly independent sets. We see that $\beta = (1 1 0)$ does the job.

With $\delta$, $\alpha$, and $\beta$ chosen, it remains only to choose $\gamma$ such that $\{\alpha, \beta, \gamma\}$ and $\{\delta, \beta, \gamma\}$ are both linearly independent sets of vectors.

Because $d_j = d_3 = 3$ for $\gamma$, there are $2^{3C}$ possible values of $\gamma = (\gamma_1 \gamma_2 \gamma_3)$. Of these $2^{3C}$, exactly $2^{2C}$ are linear combinations of $\alpha$ and $\beta$ and therefore forbidden. Similarly, $2^{2C}$ are linear combinations of $\delta$ and $\beta$ and therefore forbidden. Provided that $2^{3C} > 2^{2C} + 2^{2C} = 2^{2C+1}$, there must be at least one vector $\gamma$ that satisfies the constraints. In fact, we see that $\gamma = (0 0 1)$ works.

To summarize, we may take $\delta = (1 0 0)$, $\alpha = (0 1 0)$, $\beta = (1 1 0)$, and $\gamma = (0 0 1)$. These choices lead to the information storage scheme already displayed in Figure 3.11. In this example, we see that the components of $\alpha$, $\beta$, $\gamma$, and $\delta$ can all be taken from $GF(2)$, which is a subfield of $GF(2^C)$ for every $C \geq 1$. More generally, we can split each $C$-bit disk into chunks of $r$ or $r + 1$ bits each, where $r$ is sufficiently large that for $1 \leq j \leq n$, all components of the vector $\alpha_j$ can be chosen from $GF(2^r)$. In cases of practical interest, we can make $r \ll C$.

Let us now generalize the foregoing analysis to prove a converse of Theorem 3. We begin by establishing the following lemma.

**Lemma 3** Let $\alpha_1, \alpha_2, \ldots, \alpha_L$ and $\mathbf{x}$ be vectors of length $d$ with components in $GF(2^r)$. Take $\alpha_1, \ldots, \alpha_L$ as given; $\mathbf{x}$ will be determined later. Let $S_1, S_2, \ldots, S_N$ be subsets of $\{\alpha_1, \ldots, \alpha_L\}$ with fewer than $d$ vectors. Suppose that for each of these $N$ subsets, the vectors in the subset are linearly independent. Finally, suppose that $N < 2^r$. Then the vector $\mathbf{x}$ can be chosen such that each of the $N$ augmented sets $S_1 \cup \{\mathbf{x}\}$, $S_2 \cup \{\mathbf{x}\}, \ldots, S_N \cup \{\mathbf{x}\}$ is a linearly independent set of vectors.

**Proof:** We shall show that each set $S_j$, for $1 \leq j \leq N$, eliminates at most $2^{r(d-1)}$ of the $2^{rd}$ possible values of $\mathbf{x}$. The lemma will follow. (Note: The
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

subscripts $i, j, k,$ and $n$ used in this proof are dummy variables unrelated to their eponymous counterparts used in the rest of the text.)

Begin by selecting a particular $j \in \{1, 2, \ldots, N\},$ thus fixing $S_j.$ Let the vectors in this set $S_j$ be called $\beta_1, \beta_2, \ldots, \beta_k,$ where $k \leq d - 1$ and $\{\beta_1, \beta_2, \ldots, \beta_k\} \subseteq \{\alpha_1, \alpha_2, \ldots, \alpha_L\}.$ By hypothesis, $\beta_1, \ldots, \beta_k$ are linearly independent, so

$$\text{Span}\{\beta_1, \ldots, \beta_k\} = \{a_1\beta_1 + a_2\beta_2 + \ldots + a_k\beta_k \mid a_1, a_2, \ldots, a_k \in 2^r\}$$

has $(2^r)^k$ distinct vectors. Now $S_j \cup \{x\}$ is a linearly independent set if and only if $x \notin \text{Span}\{\beta_1, \ldots, \beta_k\}.$ Thus exactly $(2^r)^k(\leq (2^r)^{d-1})$ possible values of $x$ are ruled out by the requirement that $S_j \cup \{x\}$ be a linearly independent set of vectors.

Since there are only $N$ different sets $S_j,$ at most $N(2^r)^{d-1}$ possible values of $x$ are ruled out by the requirement that $S_j \cup \{x\}$ be a linearly independent set for each $j.$ (Some values of $x,$ e.g., the zero vector, may be ruled out by more than one set of vectors.) But $x = (x_1, x_2, \ldots, x_d)$ has $(2^r)^d(= 2^{rd})$ possible values, so if $N(2^r)^{d-1} < 2^{rd},$ then there must be at least one $x$ that satisfies the conditions of the proposition. Equivalently, it suffices to have $N < 2^r.$

Now we give a definition and use Lemma 3 to establish a proposition that will lead to a converse of Theorem 3.

**Definition:** Let $\alpha_1, \alpha_2, \ldots, \alpha_n,$ be (undetermined) vectors of length $d \leq n$ with components in $GF(2^r),$ where $r \geq 1.$ Suppose that for $1 \leq j \leq n,$ the first $d_j$ components of $\alpha_j$ can be freely chosen ($d_j \leq d$), while the last $d - d_j$ components must be zero. Assume that the vectors are ordered so that $1 \leq d_1 \leq d_2 \leq \ldots \leq d_n = d,$ where $d \leq n.$ For $1 \leq k \leq d,$ say that a subset $\{\beta_1, \ldots, \beta_k\} \subseteq \{\alpha_1, \ldots, \alpha_n\}$ is a $k$-triangular subset if the following is true: At most one of the vectors in the subset has one (free) component or fewer, at most two of the vectors have two free components or fewer, and at most $k - 1$ of the vectors have $k - 1$ free components or fewer. If we do not wish to specify the number
of elements in a $k$-triangular subset, we shall simply call the set a triangular subset. ■

**Proposition 1** Suppose that $r \geq n$. Then, for vectors $\alpha_1, \ldots, \alpha_n$ defined as above, it is possible to choose the free components of $\alpha_1, \ldots, \alpha_n$ such that for each $k \in \{1, \ldots, d\}$, every $k$-triangular subset of the $n$ vectors is a linearly independent set over the field $GF(2^r)$.

**Proof:** We proceed by induction on $n$, the number of vectors.

First suppose that $n = 1$. Then there is only one vector, $\alpha_1$, and it must have exactly one component. Since $r \geq 1$, we see that $GF(2^r)$ has at least one nonzero element. Therefore we can take $\alpha_1 = (1)$. Now the only triangular subset of $\{\alpha_1\}$ is $\{\alpha_1\}$ itself; since $\alpha_1 \neq 0$, the triangular subset is a linearly independent set, and the proposition is proved for $n = 1$.

Now suppose that the proposition is true for $n = 1, 2, \ldots, L$. We shall show that the general proposition is also true for $n = L + 1$, and the proposition will follow by induction.

Consider the set of $d$-dimensional vectors $\{\alpha_1, \ldots, \alpha_{L+1}\}$, where for $1 \leq j \leq L + 1$, the first $d_j$ components can be chosen freely, while the remaining $(d - d_j)$ components must be zero. As in the definition, we have

$$1 \leq d_1 \leq d_2 \leq \ldots \leq d_L \leq d_{L+1} = d, \text{ where } d \leq n.$$

Let $n' \triangleq n - 1 = L$ and $d' \triangleq d_L$. Then $r \geq n \geq n'$, so if the conditions of the proposition apply to the set $\{\alpha_1, \ldots, \alpha_{L+1}\}$, they also apply to the set $\{\alpha_1, \ldots, \alpha_L\}$. Therefore, by the induction hypothesis, $\alpha_1, \ldots, \alpha_L$ can be chosen such that every triangular subset of $\{\alpha_1, \ldots, \alpha_L\}$ is a linearly independent set over $GF(2^r)$.

It follows from the definition of a triangular subset that every subset of a triangular subset is itself a triangular subset. Therefore every triangular subset
of \( \{\alpha_1, \ldots, \alpha_L, \alpha_{L+1}\} \) is a triangular subset \( \{\beta_1, \ldots, \beta_k\} \) of \( \{\alpha_1, \ldots, \alpha_L\} \), perhaps with the vector \( \alpha_{L+1} \) appended to the set of \( \beta \)'s.

Thus every triangular subset of \( \{\alpha_1, \ldots, \alpha_{L+1}\} \) consists of a linearly independent set \( \{\beta_1, \ldots, \beta_k\} \) of vectors from \( \{\alpha_1, \ldots, \alpha_L\} \), perhaps with the vector \( \alpha_{L+1} \) appended to the set of \( \beta \)'s. If we can choose \( \alpha_{L+1} \) so that \( \alpha_{L+1} \) lies outside the span of \( \{\beta_1, \beta_2, \ldots, \beta_k\} \) for every linearly independent subset \( \{\beta_1, \ldots, \beta_k\} \subseteq \{\alpha_1, \ldots, \alpha_L\} \) with fewer than \( d \) vectors, then we will have proved that the proposition holds for \( n = L + 1 \).

It is almost time to use Lemma 3. Let \( N \) be the total number of linearly independent subsets of vectors from \( \{\alpha_1, \ldots, \alpha_L\} \) having at most \( d - 1 \) vectors. Then \( N \) is upper-bounded by the total number of subsets of \( \{\alpha_1, \ldots, \alpha_L\} \) (linearly independent or otherwise) having at most \( d - 1 \) elements. Including the empty set as one of the subsets that must be considered, we have

\[
N \leq \binom{L}{0} + \binom{L}{1} + \cdots + \binom{L}{d-1} = \binom{n-1}{0} + \binom{n-1}{1} + \cdots + \binom{n-1}{d-1},
\]

where \( \binom{a}{b} \triangleq \frac{a!}{b!(a-b)!} \) denotes the binomial coefficient. Since \( \sum_{k=0}^{L} \binom{L}{k} = 2^L \) and \( d \leq n \), we see that

\[
N \leq \sum_{k=0}^{d-1} \binom{n-1}{k} \leq \sum_{k=0}^{n-1} \binom{n-1}{k} = 2^{n-1}.
\]

Therefore, if \( r \geq n \), then \( 2^r \geq 2^n > 2^{n-1} \geq N \), and Lemma 3 implies that \( \alpha_{L+1} \) can be chosen to be linearly independent of every linearly independent set of \( d - 1 \) or fewer vectors from \( \{\alpha_1, \ldots, \alpha_L\} \). Thus the condition \( r \geq n \) suffices to prove the proposition.

If \( d \ll n \), then we can obtain tighter bounds on \( \sum_{k=0}^{d-1} \binom{n-1}{k} \) and relax the
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

condition that \( r \geq n \). We obtain the following proposition.

**Proposition 2** If \( r \geq 1 \) and \( d \leq \frac{n}{2} \), then the conclusion of Proposition 1 holds if \( r \geq d \log_2(n) \).

**Proof:** The proof is essentially the same as the proof of the previous proposition, but we use a tighter bound on \( \sum_{k=0}^{d-1} \binom{n-1}{k} \). Examining the proof of Proposition 1 we see that the conclusion will follow if we can show that

\[
\sum_{k=0}^{d-1} \binom{\hat{n}-1}{k} < 2^r
\]

for all \( \hat{d} \leq d \) and \( \hat{n} \leq n \). We take two steps to show that the inequality above actually holds. First we note that

\[
\sum_{k=0}^{d-1} \binom{\hat{n}-1}{k} \leq \sum_{k=0}^{d-1} \binom{n-1}{k} \leq \sum_{k=0}^{d-1} \binom{n-1}{k}
\]

for all \( \hat{d} \leq d \) and \( \hat{n} \leq n \). (The proof is easy.) Then we show that

\[
\sum_{k=0}^{d-1} \binom{n-1}{k} < 2^r \text{ if } r \geq 1, \ d \leq n/2, \ \text{and} \ r \geq d \log_2 n. \quad (*)
\]

The inequality (*) is derived below. For \( n \geq 2d \), we have \( d \geq 1, \ n \geq 2, \) and

\[
\sum_{k=0}^{d-1} \binom{n-1}{k} \leq \sum_{k=0}^{d-1} \frac{(n-1)^k}{k!} \leq \frac{(n-1)^{d-1}}{(d-1)!} \sum_{j=0}^{d-1} \binom{d-1}{n-1}^j < \frac{(n-1)^{d-1}}{(d-1)!} \sum_{j=0}^{\infty} \left( \frac{1}{2} \right)^j
\]

\[
= \frac{2(n-1)^{d-1}}{(d-1)!} < 2(n-1)^{d-1} = \frac{2(n-1)^d}{n-1}.
\]

If \( n \geq 3 \), then \( n-1 \geq 2 \), so

\[
\sum_{k=0}^{d-1} \binom{n-1}{k} < \frac{2(n-1)^d}{n-1} \leq \frac{2(n-1)^d}{2} < n^d = 2^{d \log_2 n} \leq 2^r,
\]
as desired. If $n = 2$, then we see directly that $d = 1$, $n = 2$, and
\[
\sum_{k=0}^{d-1} \binom{n-1}{k} = \binom{1}{0} = 1 < 2 \leq 2^r,
\]
since $r \geq 1$ by assumption. Thus when $r \geq 1$ and $d \leq \frac{n}{2}$, Proposition 2 holds if
\[ r \geq d \log_2(n). \]

We are now prepared to prove a converse of Theorem 3. The necessary conditions for sequential refinement given in that theorem are also sufficient conditions, provided that the capacity of each disk is large enough.

First let us recall some definitions. There is a network of $m$ users and $n$ disks, each disk having capacity $C$ bits. For $1 \leq i \leq m$, User $i$ is connected to $k_i$ disks, all of which are always accessible. (There are no disks or link failures.) Let $k_{\text{max}} = \max_{1 \leq i \leq m} \{k_i\}$. By $E$, we mean the set of edges $\{(i,j)\}$ such that User $i$ is connected to Disk $j$.

We say that the storage network admits sequential refinement if it is possible to encode and store $k_{\text{max}}C$ bits of information such that each user can recover the first $k_iC$ bits of this information from the $k_i$ disks to which he has access.

For $1 \leq j \leq n$, we associate with Disk $j$ the quantity $d_j = \min_{(i,j) \in E} \{k_i\}$, the minimum degree of the nodes representing the users connected to Disk $j$.

For $1 \leq i \leq m$ and $1 \leq s \leq k_i$, we let $N_i(s)$ be the number of disks $D_j$ attached to User $i$ for which $d_j \leq s$.

In the following definition we define triangular subsets of disks by analogy with the earlier definition of triangular subsets of vectors.

Definition: If, for a given $i \leq \{1, \ldots, m\}$, we have $N_i(s) \leq s$ for all $s \in \{1, \ldots, k_i - 1\}$, then we say that the set of disks attached to User $i$ is a triangular subset of the set of all disks.

Using the definitions above, we can prove the following theorem.

Theorem 4 Consider a given network with $m$ users and $n$ disks, each disk having capacity $C$ bits, where $C \geq 1$. Suppose that no disk or link failures occur. In order for the network to admit sequential refinement, it is necessary that for each User $i$
(1 ≤ i ≤ m), the set of disks attached to User i be a triangular subset of the set of all disks.

If \( C \geq n \), or if \( k_{\text{max}} \leq \frac{n}{2} \) and \( C \geq k_{\text{max}} \log_2 n \), then in order for the network to admit sequential refinement, it is also sufficient that for each User i, the set of disks attached to User i be a triangular subset of the set of all disks.

**Proof:** The necessary condition follows immediately from Theorem 3, using the definition of a triangular subset of disks.

To prove the sufficient condition, we let \( d = k_{\text{max}} \) and for \( 1 \leq j \leq n \), we associate with Disk j a \( d \)-dimensional vector \( \alpha_j \) over \( GF(2^C) \). For each \( j \), we can choose the first \( d_j \) components of \( \alpha_j \), but the final \( d - d_j \) components must be zero (\( d_j \leq d \) for all \( j \)). We refer to the \( k_{\text{max}} C \) bits of information to be stored in the network as \( W = (W_1, W_2, \ldots, W_d) \), where \( 1 \leq t \leq d = k_{\text{max}} \), \( W_t \in GF(2^C) \) is the finite field element corresponding to the \( t^{th} \) \( C \)-bit block of information stored in the disk network. On each Disk j we shall store the \( C \)-bit vector corresponding to \( \alpha_j W^t \). If, for each User i, the set of disks attached to User i is a triangular subset of the set of all n disks, then it follows that for each \( i \in \{1, \ldots, m\} \), the set of vectors \( \{\alpha_j : (i, j) \in E\} \) is a triangular subset of the set \( \{\alpha_1, \ldots, \alpha_n\} \). Now, by Proposition 1 and Proposition 2, the components of the \( \alpha_j \)'s can be chosen such that every triangular subset of the vectors (including every set \( \{\alpha_j : (i, j) \in E\} \)) is a linearly independent set over \( GF(2^C) \). But for each \( i \in \{1, \ldots, m\} \),

\[
d_j \triangleq \min_{(i', j') \in E} \{k_{i'}\} \leq k_i \text{ for all } j \text{ such that } (i, j) \in E.
\]

Therefore, for each \( i \in \{1, \ldots, m\} \), the \( k_i \) vectors in the set \( \{\alpha_j : (i, j) \in E\} \) all have zeros in the last \( d - k_i \) positions. Thus each User i can recover \( (W_1, \ldots, W_{k_i}) \) by solving a linearly independent set of \( k_i \) equations in the \( k_i \) unknowns \( W_1, \ldots, W_{k_i} \). Thus we see that the conditions given in this theorem are sufficient to show that a network admits sequential refinement.

**Note:** In practice, we have \( C \gg n \); that is, there are many more bits per disk.
than there are disks. In this case, we break each disk into \( r \)-bit chunks and do all arithmetic in \( GF(2^r) \), where \( r \ll C \) is as small as possible while still satisfying the requirements imposed on \( C \) in Theorem 4.

### 3.2.2 Disk or Link Failures Allowed

In Subsection 3.2.1 we found necessary and sufficient conditions for a storage network to admit sequential refinement. We assumed that no disk or link failures could occur. Let us now derive necessary and sufficient conditions for sequential refinement when disk or link failures are allowed, still assuming that each disk has capacity \( C \) bits. It is not hard to see that the case with \( f \) disk failures allowed is equivalent to the case in which each user is allowed \( f \) link failures.

Both disk failures and link failures can be accounted for by saying that for \( 1 \leq i \leq m \), User \( i \) is connected to \( n_i \) disks, at least \( k_i \) of which must be accessible at any given time. See Figure 3.12 for an example with \( m = 3 \), \( n = 4 \), \( k_1 = 1 \), \( n_1 = 2 \), \( k_2 = 2 \), \( n_2 = 3 \), \( k_3 = n_3 = 3 \). As before, we define

\[
d_j = \min_{i:(i,j) \in E} \{k_i\} \text{ for } 1 \leq j \leq n.
\]

Note that the disk degree \( d_j \) is still defined in terms of \( k_i \) (the number of disks to which User \( i \) is guaranteed access at any given time), not in terms of \( n_i \) (the total number of disks connected to User \( i \)).

![Diagram](image)

**Figure 3.12:** Sequential refinement example with link failures allowed

For \( 1 \leq i \leq m \), User \( i \) is only guaranteed access to \( k_i \) disks, each of capacity \( C \)
bits. Therefore User \( i \) cannot hope to recover more than \( k_i C \) bits reliably. It makes sense, then, to leave the earlier definition of sequential refinement unchanged. We say that a storage network admits complete sequential refinement of information, or just sequential refinement, if we can encode and store \( k_{\text{max}} C \) bits of information in the network such that for \( 1 \leq i \leq m \), User \( i \) can reliably recover the first \( k_i C \) bits of this information.

In Subsection 3.2.1, we showed that for sequential refinement without link failures, the set of \( k_i \) disks attached to each User \( i \) had to be a triangular subset of the set of all \( n \) disks. (See page 48 for the definition of a triangular subset of disks.) We shall see that when link failures (or disk failures) are allowed, the necessary and sufficient condition for sequential refinement is that for \( 1 \leq i \leq m \), every \( k_i \)-element subset of the \( n \) disks attached to User \( i \) should be a triangular subset.

**Definition:** For \( 1 \leq i \leq m \), say that User \( i \) is fully triangular if every \( k_i \)-element subset of the \( n \) disks attached to User \( i \) is a triangular subset of disks.

**Theorem 5** Consider an arbitrary storage network in which link or disk failures can occur. If the storage network admits sequential refinement, then each user must be fully triangular.

**Proof:** Lemma 2 shows that for \( 1 \leq j \leq n \), the data stored on Disk \( j \) cannot depend on any but the first \( d_j C \) bits of the \( k_{\text{max}} C \) bits of information stored in the network. Then Theorem 3 follows as before when applied to any \( k_i \) of the \( n \) disks attached to User \( i \). Theorem 5 follows immediately.

Now a converse to Theorem 5 can be derived in the same way that the converse of Theorem 5 was derived.

**Theorem 6** Assume that each disk in a storage network has the same capacity \( C \geq 1 \), where \( C \) is measured in bits. If \( C \geq n \), or if \( k_{\text{max}} \leq n/2 \) and \( C \geq k_{\text{max}} \log_2 n \), then in order for a network to admit sequential refinement, it suffices to have all \( m \) users fully triangular.
CHAPTER 3. MULTIPLE USERS WITH COMMON INFORMATION

Proof: The proof is essentially the same as the proof of the second half of Theorem 4.

Note: To decide whether a particular User $i$ is fully triangular, it is not necessary to check all $k_i$-element subsets of the $n_i$ disks attached to User $i$. (There are $\binom{n_i}{k_i}$ such subsets.) Instead, we can simply arrange the $n_i$ disks attached to User $i$ in increasing order of $d_j$ and then check whether the $k_i$ disks with the smallest values of $d_j$ constitute a triangular subset.

Example: Returning to the example of Figure 3.12, we see that the network can be characterized as in Table 3.1. In this example, User 1 is fully triangular since

<table>
<thead>
<tr>
<th>User $i$</th>
<th>$k_i$</th>
<th>$n_i$</th>
<th>${d_j : (i,j) \in E}$</th>
<th>$k_i$ smallest values of $d_j$ : ${i,j} \in E$</th>
</tr>
</thead>
<tbody>
<tr>
<td>User 1</td>
<td>1</td>
<td>2</td>
<td>${1,1}$</td>
<td>${1}$</td>
</tr>
<tr>
<td>User 2</td>
<td>2</td>
<td>3</td>
<td>${1,1,2}$</td>
<td>${1,1}$</td>
</tr>
<tr>
<td>User 3</td>
<td>3</td>
<td>3</td>
<td>${1,2,3}$</td>
<td>${1,2,3}$</td>
</tr>
</tbody>
</table>

Table 3.1: An example for which sequential refinement is impossible

$\{1\}$ corresponds to a triangular subset of disks. User 2 is not fully triangular, because the multiset $\{1,1\}$ of $d_j$'s does not correspond to a triangular set of disks. User 3 is fully triangular, because the multiset $\{1,2,3\}$ corresponds to a triangular set of disks. Because User 2 is not fully triangular, the network in Figure 3.12 does not admit sequential refinement.

3.3 Other Problems to Consider

In Section 3.2 we found necessary and sufficient conditions for a data storage network to admit complete sequential refinement of information. If a network does not admit complete sequential refinement, we might ask to what extent sequential refinement is possible. For example, given a network of users and disks and an ordered $m$-tuple
of integers \((C_1, C_2, \ldots, C_m)\) with \(C_{\text{max}} \triangleq \max_{1 \leq i \leq m} C_i\), we might ask whether it is possible to encode and store \(C_{\text{max}}\) bits of information in the network such that each User \(i\) can reliably recover the first \(C_i\) bits of this information. We might ask such a question first for reliable disks and links and then for unreliable disks and links.

More generally, for an arbitrary network, we might try to find a general characterization of the set of all \(m\)-tuples \((C_1, \ldots, C_m)\) that can be achieved as above. This set could be called the **sequential refinement capacity region** of the network.

The problems already mentioned can also be posed when the disks in the network do not all have the same storage capacity. It is possible that multi-user problems involving different disk capacities can be solved using techniques developed for the single-user problem.

One more class of problems to consider involves users storing independent information rather than common information. In the next chapter we consider a number of problems involving the storage and recovery of independent information.
Chapter 4

Storage of Independent Information

4.1 Introduction

So far we have considered a number of problems involving different users who each wish to recover (a subset of) the same information from a storage network. However, it is often the case that different users instead want to store independent information. In ordinary time-sharing computation, for example, each user stores his own files on some set of disks, and many different users share a single disk.

In this chapter we shall consider a several problems involving multiple users who wish to store and reliably recover independent information from a network of storage units, usually referred to as disks for simplicity. (The problem of storing information is easily solved using a write-only memory. Some difficulties arise, however, when we insist upon reliable recovery of the information.) In general, we shall assume that there are \( m \) users and \( n \) disks in all. The users \( U_1, \ldots, U_m \) wish to reliably recover \( X_1, \ldots, X_m \) bits of independent information, respectively. The disks, labeled \( D_1, \ldots, D_n \), have respective capacities \( C_1, \ldots, C_n \) bits. We shall often assume that all disks have the same capacity \( C \). In a slight abuse of notation, we shall sometimes refer to user \( U_i \)'s information as \( U_i \), and to the contents of disk \( D_i \) as \( D_i \). If node failures (i.e., disk failures) are allowed, then \( f \) is the maximum number of disk failures
allowed. If link failures are allowed, then for \( 1 \leq i \leq m \), the quantity \( f_i \) represents the maximum number of link failures that User \( i \) can tolerate. (For example, if \( f_1 = 2 \) and \( f_3 = 5 \), then User 1 can recover his stored information even if he loses his links to any two of the disks to which he personally has access, and User 2 can recover her stored information even if she loses her links to any five of the disks to which she has access.)

We shall consider the following problems:

I. No disk or link failures
   
   (A) All users have access to the same set of disks
       
       1. Only one disk
       2. Arbitrary number of disks
   
   (B) Different users have access to different subsets of the \( n \) disks

II. Node or link failures allowed

   (A) \( m \) users, each with access to all \( n \) disks

   (B) \( m \) users, each with access to at least \( f_i \) private disks

### 4.2 Cases Without Disk or Link Failures

We begin with a definition and a very simple problem.

**Definition:** Let the *storage capacity region*, or simply the *capacity region*, of a network be the set of \( m \)-tuples \( (X_1, \ldots, X_m) \) such that users \( U_1, \ldots, U_m \) can simultaneously recover \( X_1, \ldots, X_m \) bits of independent information.

**Problem 1** Suppose that users \( U_1, \ldots, U_m \) all have access to a single (reliable) disk \( D \) with capacity \( C \) bits. How can we characterize the capacity region \( (X_1, \ldots, X_m) \) such that users \( U_1, \ldots, U_m \) can simultaneously recover \( X_1, \ldots, X_m \) bits of information, respectively, from the disk?
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

Solution: As usual, we assume that the $X_i$'s are all integers. If $X_1 + X_2 + \cdots + X_m \leq C$, then it is clear that User $i$ can simply reserve $X_i$ bits on the disk for his own use. Therefore in order for $(X_1, \ldots, X_m)$ to be achievable, it is sufficient to have $\sum_{i=1}^{m} X_i \leq C$. Conversely, we shall show that it is also necessary to have $\sum_{i=1}^{m} X_i \leq C$.

Proof of necessity: Recall that users $U_1, \ldots, U_m$ store independent information. As usual, we assume that for each user $U_i$, all $2^{X_i}$ possible sequences of $X_i$ bits can actually occur. Since our storage scheme must work for all possible information sequences, it must work with probability 1 if, for each $U_i$, all $2^{X_i}$ possible information sequences occur with equal probability, so that $H(U_i) = X_i$. Because each user must be able to recover his information reliably from $D$, it follows that $H(U_i|D) = 0$ for all $i$ and that $H(U_1, \ldots, U_m|D) = 0$. Now

$$I(U_1, \ldots, U_m; D) = H(U_1, \ldots, U_m) - H(U_1, \ldots, U_m|D)$$

$$= H(U_1) + \cdots + H(U_m) - 0$$

$$= X_1 + \cdots + X_m,$$

where the first equality is the definition of mutual information and the second equality follows from independence.

But we also have

$$I(U_1, \ldots, U_m; D) = H(D) - H(D|U_1, \ldots, U_m)$$

$$\leq C - H(D|U_1, \ldots, U_m)$$

$$\leq C$$

where the first inequality holds because $D$ can take on at most $2^C$ different values and the second inequality holds because conditional entropy, like entropy, is always nonnegative.

Therefore $X_1 + \cdots + X_m \leq C$. ■

Thus we have the following lemma.

Lemma 4 The capacity region $\{(X_1, \ldots, X_m)\}$ for Problem 1 is precisely the set of $(X_1, \ldots, X_m)$ for which $X_1 + \cdots + X_m \leq C$. ■
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

It is unlikely that anyone would find Lemma 4 counterintuitive, but we have given the proof in detail because intuition will be less reliable in some of the problems that we shall consider later. In the meantime, let us consider one more simple problem.

**Problem 2** Suppose that users $U_1, \ldots, U_m$ all have access to the same set of $n$ disks, labeled $D_1, \ldots, D_n$, with respective capacities $C_1, \ldots, C_n$ bits. How can we characterize the set of $m$-tuples $(X_1, \ldots, X_m)$ such that users $U_1, \ldots, U_m$ can simultaneously recover $X_1, \ldots, X_m$ bits of information, respectively, from the $n$ disks?

**Solution:** This problem is exactly equivalent to Problem 1 with a single disk $D$ of storage capacity $C = C_1 + C_2 + \cdots + C_n$. Lemma 4 immediately yields the following lemma.

**Lemma 5** The capacity region $\{(X_1, \ldots, X_m)\}$ for Problem 2 is precisely the set of $(X_1, \ldots, X_m)$ for which $X_1 + \cdots + X_m \leq C_1 + \cdots + C_n$. ■

Having solved two simple problems, let us now consider a more interesting problem.

**Problem 3** Consider a storage network with users $U_1, \ldots, U_m$ and disks $D_1, \ldots, D_n$, where disk $D_j$ has capacity $C_j$ bits and each user $U_i$ has access to some subset $S_i$ of the $n$ disks. How can we characterize the capacity region $(X_1, \ldots, X_m)$ such that users $U_1, \ldots, U_m$ can simultaneously recover $X_1, \ldots, X_m$ bit of information, respectively, from the $n$ disks? (See Figure 4.1 for an example with $m = 3$ and $n = 4$.)

In Figure 4.1, we have $S_1 = \{D_1, D_2\}$, $S_2 = \{D_1, D_3\}$, $S_3 = \{D_2, D_3, D_4\}$, $D_1 = 140$, $D_2 = 230$, $D_3 = 170$, and $D_4 = 210$. Of course, this is a toy example, but the solution would be essentially unchanged if all capacities were in gigabits rather than bits.

To solve Problem 3 we shall use results from network flow theory to find a region $\{(X_1, \ldots, X_m)\}$ that can be achieved by letting each user reserve his own disk space. Then we shall use entropy-based arguments to show that this achievable region is the best possible. (Note that it is not immediately obvious that it is optimal for each
user to reserve certain bits on the disks for his own use. The results of the previous chapters suggest that it might be better to store functions of different information bits, e.g., exclusive-ors of different bits, than to store “raw” information bits on the disks.)

In order to find an achievable region \( \{(X_1, \ldots, X_m)\} \), we assume for now that each user simply reserves different bit positions on the various disks. (Propositions 1 and 2 will be derived using this assumption.) In this case, it is no longer necessary to think of the stored commodity as information; we might just as well be storing some liquid in containers of capacity 140, 230, 170, and 210, respectively. In fact, we might as well assume that the bit positions that any given user \( U_i \) reserves on any particular disk \( D_j \) are contiguous. Now, supposing that User \( i \) has \( X_i \) units of liquid to store, we consider the liquid flow in the network. Referring to Figure 4.1, we see that we can convert the original problem into a network flow problem by adding an auxiliary source node to supply the \( m \) users with liquid and an auxiliary terminal node to collect all the liquid from the \( n \) storage sites.

Thus we obtain Figure 4.2, where the arrows indicate direction of flow and the number above each link indicates the maximum flow allowed in that link. The links connecting users to disks may be thought of as having infinite capacity. However, the flow in these links will be limited by the constraints on the flow in other links.

In Figure 4.2, if we can assign a flow value to each link such that the three leftmost links are saturated (with flows \( X_1, X_2, \) and \( X_3 \), respectively), then we can immediately convert this set of flow values into a solution to the original problem. (See Figures 4.3
Thus we see that \((X_1 = 210, X_2 = 110, X_3 = 390)\) is achievable. It is clear that Figures 4.3 and 4.4 can be generalized to handle arbitrary \(m, n, X_1, \ldots, X_m, C_1, \ldots, C_n\).

Happily, there are already efficient techniques known for solving network flow problems. (See, for example, [14, pp. 178–180] and [54, pp. 118-120].) The problems that we consider can be solved in time \(O((m + n)^3)\); i.e., there exists a constant \(K\) such that \(K(m + n)^3\) operations suffice to determine whether a particular \(m\)-tuple
(X_1 = 210) U_1 → 80 → D_1 (C_1 = 140)
(X_2 = 110) U_2 → 60 → 50 → D_2 (C_2 = 230)
(X_3 = 390) U_3 → 80 → 110 → 200 → D_3 (C_3 = 170)
(D_4 (C_4 = 210)

Figure 4.4: An achievable \((x_1, x_2, x_3)\) for the storage problem

\((X_1, \ldots, X_m)\) is feasible for a given network of \(m\) users and \(n\) disks and, if so, to allocate bits to disks appropriately. For even more efficient techniques, see Gusfield, Martel, and Fernandez-Baca [31].

For any given network of disks and users with given \(m, n, C_1, \ldots, C_n\), we can use the max-flow, min-cut theorem (Theorem 7) to characterize the set of all \(m\)-tuples \((X_1, \ldots, X_m)\) achievable by having each user reserve an integer number of bit positions on the various disks. We shall also see from simple entropy-based arguments that no information storage scheme can do any better. Thus for any given storage network we shall obtain an exact characterization of the set of achievable \(m\)-tuples \((X_1, \ldots, X_m)\), where \(X_i\) is the number of bits of information that User \(i\) can store and reliably recover from the network.

Before stating the max-flow, min-cut theorem, let us formally define a few concepts that we have been using informally.

**Definition:** An abstract directed graph \(G_d(V, E)\), or simply a directed graph or digraph \(G_d\), consists of a set \(V\) of elements called nodes together with a set \(E\) of ordered pairs of the form \((i, j)\), \(i, j \in V\), called the arcs or directed edges of \(G_d\). Node \(i\) is called the initial node and node \(j\) the terminal node of \((i, j)\). Together they are the endpoints of \((i, j)\).
For each ordered pair \((i, j)\), there is at most one arc going from node \(i\) to node \(j\) (and at most one arc in the reverse direction).

Let us suppose that some commodity is stored at some initial node(s), and delivered to some terminal node(s). We can think of each arc of a directed graph as a route along which the commodity can flow.

Let us define by \(b(i, j)\) the bound (or capacity) of the arc from \(i\) to \(j\); that is, let \(b(i, j)\) be the maximum amount of the commodity that can flow directly from node \(i\) to node \(j\). It is generally assumed that the commodity is discrete and that \(b(i, j)\) is an integer. (Note that if there is no arc leading from \(i\) to \(j\), then we may take \(b(i, j)\) to be zero.)

With the previous definition in mind, we define the concepts of network and network flow.

**Definition [54, p. 23]:** A network \(N = (s, t, V, E, b)\) is a directed graph \((V, E)\) together with a source \(s \in V\) with 0 indegree (that is, \(s\) is not the terminal node of any arc), a terminal \(t \in V\) with 0 outdegree (that is, \(t\) is not the initial node of any arc), and with a bound (or capacity) \(b(u, v) \in \mathbb{Z}^+\) for each \((u, v)\) in the edge set \(E\). (See Figure 4.5.)

![Figure 4.5: An example showing capacities of arcs](image)
A flow $f$ in $n$ is a vector in $\mathbb{R}^{|E|}$ (one component $f(u,v)$ for each arc $(u,v) \in E$) such that:

1. $0 \leq f(u,v) \leq b(u,v)$ for all $(u,v) \in E$ (i.e., the flow in each arc is no more than that arc's capacity).

2. $\sum_{u:(u,v) \in E} f(u,v) = \sum_{u:(v,u) \in E} f(v,u)$ for all $v \in V - \{s,t\}$ (i.e., for all nodes but the source and terminal, the total flow out of the node equals that total flow into the node).

The value of the vector-valued flow $f$, sometimes denoted by $|f|$, is the following quantity:

$$|f| = \sum_{u:(s,u) \in E} f(s,u).$$  \hspace{1cm} (4.7)

For example, in the Figure 4.6, we show a legitimate flow for $N$; its value is $|f| = 5$. Note that the flow in each arc is no more than the capacity of that arc. Also note that nodes $v_1$, $v_2$, $v_3$, and $v_4$ have the same outgoing flow as incoming flow.

An important concept in dealing with flow problems is that of a cut.

**Definition** (See [54, p. 118]): For nodes $s$ and $t$, an $s$-$t$ cut is a partition $(W, \overline{W})$ of the nodes of $V$ into sets $W$ and $\overline{W}$ such that $s \in W$ and $t \in \overline{W}$. The capacity of
an s-t cut is
\[ C(W, \overline{W}) = \sum_{(i,j) \in E} b(i, j). \] (4.8)

In words, the capacity of a cut is the sum of the capacities of the "forward" arcs: those which go from nodes in \( W \) to those in \( \overline{W} \). We would expect that the value of an s-t flow cannot exceed the capacity of an s-t cut, since all the s-t flow must pass through the forward arcs of a cut.

In fact, the following theorem can be proved.

**Theorem 7 (Max-flow, min-cut)** (due to Ford and Fulkerson; see [54]): The value of any s-t flow is no greater than the capacity \( C(W, \overline{W}) \) of any s-t cut. Furthermore, the value of the maximum flow equals the capacity of the minimum cut, and a flow \( f \) and a cut \((W, \overline{W})\) are jointly optimal if and only if

\[
\begin{align*}
f(x, y) &= 0 \quad \text{for } (x, y) \in E \text{ such that } x \in \overline{W} \text{ and } y \in W \\
f(x, y) &= b(x, y) \quad \text{for } (x, y) \in E \text{ such that } x \in W \text{ and } y \in \overline{W}.
\end{align*}
\]

Now we show how to transform our original distributed information storage problem into a network flow problem so that we can apply the max-flow, min-cut theorem just stated.

**Recall:** At this point, we are assuming that each user simply reserves certain bit positions on the various disks for his own use. It is not the case, for example, that different users exclusive-or some of their bits together and store the result.

Given a set of users \( U_1, \ldots, U_m \) who wish to store \( X_1, \ldots, X_m \) bits, respectively, of independent information on disks \( D_1, \ldots, D_n \) with respective capacities \( C_1, \ldots, C_n \), we have a graph like that of Figure 4.7 (where \( m = 3 \) and \( n = 4 \)).

We transform the undirected graph above into a flow network by adding an auxiliary source \( s \) and auxiliary terminal \( t \), by making the arcs between nodes directed from users to disks, and by choosing the capacities (not necessarily the flows) of the different arcs as in Figure 4.8.

**Proposition 3** Users \( U_1, \ldots, U_m \) can reliably recover \( X_1, \ldots, X_m \) bits of information from the disks \( D_1, \ldots, D_n \) by reserving space on the disks if and only if the maximum \( s-t \) flow in the associated flow network equals \( X_1 + X_2 + \cdots + X_m \).
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

Figure 4.7: A storage network

Figure 4.8: A flow network equivalent to the previous storage network

Proof: \((\Rightarrow)\): Suppose that users \(U_1, \ldots, U_m\) can reliably recover \(X_1, \ldots, X_m\) bits of information. Then it follows readily that the users can store information such that for \(1 \leq i \leq m\),

\[
\sum_{j=1}^{n} \text{(Number of bits that User } i \text{ stores on disk } j \text{)} = X_i. \quad (4.9)
\]

Letting \(f(U_i, D_j)\) be the “flow” of information, in bits, directly from User \(i\) to disk \(j\), we see that

\[
f(U_i, D_j) = \text{(Number of bits that User } i \text{ stores on disk } j\text{)}. \quad (4.10)
\]
Then
\[ \sum_{j=1}^{n} f(U_i, D_j) = (\text{Total number of bits stored by } U_i) = X_i \] (4.11)

and
\[ \sum_{i=1}^{m} f(U_i, D_j) = (\text{Total number of bits stored on disk } j) \leq C_j, \] (4.12)
since we have assumed that all the information can be reliably recovered.

Thus, with the capacities of the arcs as shown in the flow network, we see that the total flow in each arc is less than or equal to the arc's capacity, as required for a feasible flow. Since the total flow out from each node \( U_i \) is \( X_i \), the total flow into each node \( U_i \) must also be \( X_i \). Thus for each \( i \in \{1, 2, \ldots, m\} \), the arc from the source \( s \) to node \( U_i \) must be saturated, containing flow \( X_i \). Therefore the maximum \( s-t \) flow in the network is at least \( X_1 + X_2 + \cdots + X_m \).

Choosing a cut such that \( W = \{s\} \) and \( \bar{W} = V - \{s\} \), we see by the max-flow, min-cut theorem that the maximum \( s-t \) flow in the network is at most \( X_1 + X_2 + \cdots + X_m \). Therefore the maximum \( s-t \) flow in the network is exactly \( X_1 + X_2 + \cdots + X_m \), as we wished to show.

**Proof (\( \Leftarrow \))**: Suppose that the maximum flow in the network represented in Figure 4.8 is \( X_1 + X_2 + \cdots + X_m \). Then every arc leading from the source, \( s \), must be utilized fully and each arc leading to the terminal, \( t \), must have flow less than or equal to its capacity.

Reversing the argument in the first half of the proof, we see that we can use the max-flow realization from Figure 4.8 to obtain a bit-allocation scheme in Figure 4.7 that allows each \( U_i \) to recover \( X_i \) bits of information. \[ \blacksquare \]

Using the equivalence of the information-storage problem represented in Figure 4.7 and the max-flow problem represented in Figure 4.8, we can use the max-flow, min-cut theorem on the max-flow problem to obtain necessary and sufficient conditions for the feasibility of any vector \( (X_1, \ldots, X_m) \). Recall, however, that these conditions are necessary only under the assumption that each user simply reserves an integer
number of bit positions on the disks for his own use. It is conceivable at this point that other \( m \)-tuples \((X_1, \ldots, X_m)\) may be achievable using more sophisticated information storage schemes.

**Proposition 4** \((X_1, \ldots, X_m)\) is achievable by reserving space on disks if and only if for all \( W \subseteq \{U_1, \ldots, U_m\} \),

\[
\sum_{i:U_i \in W} X_i \leq \sum_{j:(U_i, D_j) \in \mathcal{E}} C_j.
\]

**Proof (Outline):** Using the previous results, we can show that

\((X_1, \ldots, X_m)\) is achievable by reserving space on disks \(\iff\)

\[
C(W, \overline{W}) \geq \sum_{i=1}^{m} X_i \text{ for all } W \subseteq \{U_1, \ldots, U_m\} \text{ (see Figure 4.8)} \iff
\]

\[
\sum_{j:(U_i, D_j) \in \mathcal{E}, \text{ for some } U_i \in W} C_j + \sum_{i:U_i \in W} X_i \geq \sum_{i=1}^{m} X_i \text{ for all } W \subseteq \{U_1, \ldots, U_m\} \iff
\]

\[
\sum_{i:U_i \in W} X_i \leq \sum_{j:(U_i, D_j) \in \mathcal{E}, \text{ for some } U_i \in W} C_j \text{ for all } W \subseteq \{U_1, \ldots, U_m\}.
\]

The proposition has a simple interpretation. For each subset \( W = \{U_{i_1}, \ldots, U_{i_k}\} \) of the set of users \( \{U_1, \ldots, U_m\} \), think of the \( k \) users in the subset as a single superuser. Then the set of disks available to this superuser is just the set of disks accessible to at least one user in \( W \). In order for \((X_1, \ldots, X_m)\) to be feasible, the total information that the superuser wishes to store and reliably recover must be no greater than the total capacity of the disks available to this superuser.

Conversely, the proposition implies that if, for each subset \( W \subseteq \{U_1, \ldots, U_m\} \), the corresponding superuser wishes to store no more information than can be stored on the disks to which he has access, then the \( m \)-tuple \((X_1, \ldots, X_m)\) is feasible.

**Remark:** For large values of \( m \), the conditions of the proposition are difficult to use, because there are \( 2^m \) different conditions to check. In practice, we would instead use one of several polynomial-time max-flow algorithms (see \([31, 14]\)) to determine whether a given \( m \)-tuple \((X_1, \ldots, X_m)\) is feasible.
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

Recall: Proposition 4 assumes that each user simply reserves bit positions on the various disks for his own use. However, it is conceivable that more information could be stored by, for example, allowing the bits stored on the disks to be exclusive-or's of bits from the information stored by several different users.

Now we show that every information-storage scheme must satisfy the set of $2^m$ inequalities above. Thus it will follow that $(X_1, \ldots, X_m)$ is achievable for a given network for some storage scheme if and only if the $2^m$ subset-induced inequalities are all satisfied.

**Proposition 5** Consider a network with $m$ users $U_1, \ldots, U_m$ and $n$ disks $D_1, \ldots, D_n$ with respective integer capacities $C_1, \ldots, C_n$ bits. Allow each bit stored on every disk to be an arbitrary function of the users' information. Then, if each User $i$ can store and reliably recover $X_i$ bits from the network ($X_i$ an integer), the following $2^m$ inequalities must hold:

$$\sum_{i: U_i \in W} X_i \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j \quad \text{for all } W \subseteq \{U_1, \ldots, U_m\}. \quad (4.13)$$

**Proof:** For each subset $W \subseteq \{U_1, \ldots, U_m\}$, let $U_W$ be a superuser with access to all the disks accessible to any user in the subset $W$. For each $j \in \{1, 2, \ldots, n\}$, there are just $2^{C_j}$ different possible states of disk $j$. It follows immediately that $U_W$ can distinguish at most $\prod_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} 2^{C_j}$ different states. Since the entropy is maximized by a uniform distribution, it follows that

$$H(U_W) \leq \log_2(\prod_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} 2^{C_j}) \quad (4.14)$$

$$= \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j \text{ bits.} \quad (4.15)$$

By assumption, though, $U_1, \ldots, U_m$ have independent information. Therefore, by additivity,

$$H(U_W) = H(\{U_i : U_i \in W\}) = \sum_{i: U_i \in W} H(U_i) = \sum_{i: U_i \in W} X_i. \quad (4.16)$$
Therefore
\[
\sum_{i: U_i \in W} X_i = H(U_W) \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j. \quad \blacksquare \tag{4.17}
\]

Combining the last two propositions, we obtain the following theorem.

**Theorem 8** Consider a network with \( m \) users \( U_1, \ldots, U_m \) and \( n \) disks \( D_1, \ldots, D_n \) with respective integer capacities \( C_1, \ldots, C_n \). Then for integer \( X_i \)'s, the \( m \) users \( U_1, \ldots, U_m \) can store and reliably recover \( X_1, \ldots, X_m \) bits of independent information, respectively, if and only if for all \( W \subseteq \{ U_1, \ldots, U_m \} \),
\[
\sum_{i: U_i \in W} X_i \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j. \quad \blacksquare \tag{4.18}
\]

Thus for any network, we can completely characterize the set of integer-valued \( m \)-tuples \( (X_1, \ldots, X_m) \) such that users \( U_1, \ldots, U_m \) can store and reliably recover \( X_1, \ldots, X_m \) bits, respectively, of independent information. We have seen that when disk failures are not allowed, it is optimal for each user simply to reserve a set of bit positions for his own use. In the next section, we consider the more difficult problems that arise when disks (nodes) or communication links between users and disks are allowed to fail.

### 4.3 Node and Link Failures Allowed

In the last section we saw how to determine the "capacity region" \( \{(X_1, \ldots, X_m)\} \) of any storage network for which the \( m \) users \( U_1, \ldots, U_m \) store independent information, provided that no disk or link failures occur. In the present section we shall determine the capacity regions of some networks in which disk or link failures are allowed.

#### 4.3.1 Maximally Connected Networks

We continue our study of information storage capacity by supposing that every user is connected to all \( n \) disks (having capacities \( C_1, \ldots, C_n \) bits) and that \( f \) disk failures
are allowed, where $0 < f < n$. (Equivalently, we can suppose instead that every user is allowed $f$ link failures.) We assume that $C_j \gg \log_2 n$ for all $j$ to ensure that we can store information using the Galois-field techniques described in Chapter 2.

**Definition:** Given any information storage network we say that an $m$-tuple $(X_1, \ldots, X_m)$ is *achievable* if $U_1, \ldots, U_m$ can simultaneously and reliably recover $X_1, \ldots, X_m$ bits of independent information from the storage network. ■

We define the capacity region for a given information storage network asymptotically in order to avoid some technical difficulties concerning the boundary of the capacity region.

**Definition:** Given a network of $m$ users and $n$ disks, we say that an $m$-tuple $(X_1, \ldots, X_m)$ is in the *capacity region* if and only if for some positive real number $A$, the $m$-tuple $([AX_1], \ldots, [AX_m])$ is an achievable $m$-tuple when the original disk capacities $C_1, \ldots, C_n$ are replaced by $[AC_1], \ldots, [AC_n]$, respectively. With this definition the $X_i$’s need not be integers. Points on the *boundary* of the capacity region may or may not be achievable in practice. ■

**Interpretation of Definition:** The capacity region is essentially just the set of all achievable $m$-tuples $(X_1, \ldots, X_m)$. We define the capacity region as above so that it is unchanged by scaling; the capacity region is the same whether information is measured in bits, bytes, or megabytes. The definition allows us to ignore the minor discrepancies that can exist between the necessary and sufficient conditions on achievability that can exist if, for example, the $X_i$’s do not have a common divisor greater than $\log_2 n$. In practice, the change in each $X_i$ caused by this sanitized definition is on the order of $k_i \log_2 n$ bits, where $n$ is the total number of disks and $k_i$ is the minimum number of disks to which User $i$ has access at any given time. When $C_j \gg n \log_2 n$ for all $j$, there is no significant difference between the capacity region and the set of achievable $m$-tuples $(X_1, \ldots, X_m)$. ■
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

Having defined the capacity region for an arbitrary information storage network, let us now find the storage capacity of a maximally connected network, in which each of the m users is connected to all n disks. We suppose that f disk failures are allowed, where 0 < f < n.

**Theorem 9** Consider any information storage network having m users, each connected to the same n disks. The capacity region of such a maximally connected network with m users and n disks, with disk capacities \( C_1, \ldots, C_n \), is the set

\[
\{(X_1, \ldots, X_m) : \sum_{i=1}^{n} X_i \leq \sum_{j=1}^{k} C_j \text{ and } X_i \geq 0 \text{ for all } i\},
\]

where the \( C_j \)'s are assumed to be ordered so that

\[ C_1 \leq C_2 \leq \ldots \leq C_n \]

and where \( k = n - f \) is the number of disks that are guaranteed to be accessible at any given time.

**Proof:** Even if the users could communicate with each other, they could only reliably recover a total of \( C_1 + \cdots + C_k \) bits. Using the asymptotic definition of storage capacity given earlier, we see in the same way that if \( (X_1, \ldots, X_m) \) is in the capacity region, then for some real \( A \), we have

\[ [AX_1] + \cdots + [AX_m] \leq [AC_1] + \cdots + [AC_k]. \] (4.19)

Therefore

\[ AX_1 + \cdots + AX_m \leq AC_1 + \cdots + AC_k, \] (4.20)

so

\[ X_1 + \cdots + X_m \leq C_1 + \cdots + C_k. \] (4.21)

Thus for \( (X_1, \ldots, X_m) \) to be in the capacity region, it is necessary to have

\[ \sum_{i=1}^{m} X_i \leq \sum_{j=1}^{k} C_j \text{ and } X_i \geq 0 \text{ for all } i. \]
Conversely, suppose that $\sum_{i=1}^{m} X_i < \sum_{j=1}^{k} C_j$; that is, suppose that $(X_1, \ldots, X_m)$ is in the interior of the alleged capacity region. Then

$$\sum_{i=1}^{m} X_i = (1 - \epsilon) \sum_{j=1}^{k} C_j$$

for some $\epsilon > 0$.

Recalling the conditions given in Chapter 2 under which Galois-field techniques can be used to store information, we define $r = \lceil \log_2 n \rceil$. We shall be working with $r$-bit chunks of information.

Now for some sufficiently large value of $A$, if the disks have capacities $r[AC_1], \ldots, r[AC_n]$ bits, then each user $U_i$ can reserve $r[AX_i(C_j/\sum_{i=1}^{k} C_i)]$ bit positions on each disk $D_j$. It is easy to show that

$$\sum_{i=1}^{m} r[AX_i(C_j/\sum_{i=1}^{k} C_i)] \leq r[AC_j]$$

for all sufficiently large $A$.

Then each user $U_i$ can use the Galois-field techniques of Chapter 2 to reliably recover at least

$$r \sum_{j=1}^{k} [AX_i(C_j/\sum_{i=1}^{k} C_i)]$$

bits of information.

Since each term in the last sum is an integer, the sum itself is an integer. Also,

$$\sum_{j=1}^{k} [AX_i(C_j/\sum_{i=1}^{k} C_i)] \geq \sum_{j=1}^{k} AX_iC_j/\sum_{i=1}^{k} C_i = AX_i. \quad (4.22)$$

Since the sum on the left side of the last equation must be an integer, it follows that each user $U_i$ can recover at least $r[AX_i]$ bits of information when the disks have capacities $r[AC_1], \ldots, r[AC_n]$.

But $r[AX_i] \geq [(rA)X_i]$ and $r[AC_j] \leq [(rA)C_j]$, so we can let $A' = rA$ and conclude that with disks of storage capacities $[A'C_1], \ldots, [A'C_n]$ bits, each user $U_i$ can reliably recover at least $[A'X_i]$ bits of information.

Therefore, if $\sum_{i=1}^{m} X_i < \sum_{j=1}^{k} C_j$, then $(X_1, \ldots, X_m)$ is in the capacity region of the storage network. This completes the second half of the theorem.

Thus we see that the capacity region is given by

$$\{(X_1, \ldots, X_m) : \sum_{i=1}^{m} X_i \leq \sum_{j=1}^{k} C_j \quad \text{and} \quad X_i \geq 0 \text{ for all } i\}. \quad (4.23)$$
Points on the boundary of the capacity region may or may not be achievable.

4.3.2 Users with Private Disks

In the last subsection we defined the capacity region for an arbitrary information storage network and characterized this region for the case in which each user is connected to every disk. In this subsection we characterize the capacity region for another class of problems. As before, we assume that the users wish to store independent information.

Suppose that \( m \) users are connected to arbitrary subsets of a set of \( n \) disks, each disk having capacity \( C \) bits. The subsets of disks can be overlapping. Assume that each user \( U_i \) is connected to \( n_i \) disks and can suffer as many as \( f_i \) link failures. (Alternatively, we can assume that \( f \) disk failures can occur; this case is equivalent to the problem in which \( f_1 = \ldots = f_m = f \).) Let \( k_i = n_i - f_i \) be the number of disks to which user \( U_i \) is guaranteed access at any given time.

Finally, suppose that each user \( U_i \) has private access to \( p_i \) disks, where \( p_i \geq f_i \); that is, suppose that each user \( U_i \) is connected to at least \( f_i \) disks that are not connected to any other user. These private disks are included in the total of \( n_i \) disks attached to \( U_i \). Some of the links that fail may be links to private disks.

See Figure 4.9 for an example.

\[
\begin{align*}
(f_1 = 1) & \quad \{U_1 \rightarrow C, C, C, \ldots, C\} \quad p_1 = 2 \\
(f_2 = 1) & \quad \{U_2 \rightarrow C, C, C, \ldots, C\} \quad p_2 = 1
\end{align*}
\]

Figure 4.9: Users with private disks
Given any network of the form described above, we define its corresponding communal network as the network in which all private disks—and the links to them—are removed. For the communal network, it is assumed that no disk or link failures occur. See Figure 4.10 for the communal network corresponding to the network of Figure 4.9.

![Diagram showing communal network derived from previous network](image)

Figure 4.10: The communal network derived from the previous network

We shall prove the following theorem.

**Theorem 10** Consider the problem described above. For $1 \leq i \leq m$, each user $U_i$ has private access to $p_i$ disks, where $p_i \geq f_i$ and where $f_i$ is the maximum number of link failures that user $U_i$ can suffer. Let $W$ refer to an arbitrary subset of $(U_1, \ldots, U_m)$, the set of all users. For $1 \leq i \leq m$, let $X_i$ be the number of bits that $U_i$ stores in the network and let $Y_i = X_i - (p_i - f_i)C$.

The capacity region for this information storage network is the set of $m$-tuples

$\{(X_1, \ldots, X_m) : \text{for all } W \subseteq \{U_1, \ldots, U_m\}, 0 \leq \sum_{i: U_i \in W} Y_i \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j\}$, (4.24)

where $C_j = C$ for $1 \leq j \leq n$.

Equivalently, the capacity region is the capacity region $\{(Y_1, \ldots, Y_m)\}$ associated with the communal network (derived using the max-flow, min-cut theorem), but with each user $U_i$ in the original network being allowed to store an additional $(p_i - f_i)C$ bits of information because of his privately held disks.
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

Proof: Rather than using the precise definition of the capacity region as in Theorem 9 and multiplying each $X_i$ and $C_j$ by some large constant $A$, we simply work with the original values of $X_i$ and $C_j$ and find the set of achievable $m$-tuples $(X_1, \ldots, X_m)$. In order to avoid obscuring the main ideas of the proof, we gloss over the minor complications that can arise if, for example, $X_i$ is not divisible by $k_i$. The proof can be made precise by proceeding as in the proof of Theorem 9.

First let us show that the region given in the present theorem is an outer bound on the capacity region. We show that for each user $U_i$ to recover $X_i$ bits of information reliably, it is necessary that

$$
\text{for all } W \subseteq \{U_1, \ldots, U_m\}, 0 \leq \sum_{i: U_i \in W} Y_i \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j.
$$

(4.25)

To see that these constraints are necessary, suppose that for $1 \leq i \leq m$, user $U_i$ loses access to $f_i$ of his private disks. (That is, all of the links that fail are connected to private disks.) In this case, every user $U_i$ can store at most $(p_i - f_i)C$ bits of information on the remaining $(p_i - f_i)$ disks to which he still has access. By Theorem 8, if users $U_1, \ldots, U_m$ store $Y_1, \ldots, Y_m$ bits on the common disks of the communal network, then $Y_1, \ldots, Y_m$ must satisfy the conditions given in the current theorem. Then

$$
X_i \leq Y_i + (p_i - f_i)C \text{ for each user } U_i,
$$

as we wished to show.

Therefore the conditions on $X_1, \ldots, X_m$ given in this theorem are necessary for an $m$-tuple $(X_1, \ldots, X_m)$ to be achievable and thus to lie in the capacity region.

Now let us prove the converse. Given an $m$-tuple $(X_1, \ldots, X_m)$, consider the associated $m$-tuple $(Y_1, \ldots, Y_m)$ with $Y_i = X_i - (p_i - f_i)C$ for $1 \leq i \leq m$.

If for all subsets $W \subseteq \{U_1, \ldots, U_m\}$,

$$
0 \leq \sum_{i: U_i \in W} Y_i \leq \sum_{j: D_j \text{ is attached to } U_i \text{ for some } U_i \in W} C_j,
$$

(4.26)

where $C_j = C$ for all $j$, then we shall see that users $U_1, \ldots, U_m$ can simultaneously and reliably recover $X_1, \ldots, X_m$ bits of independent information from the network.
CHAPTER 4. STORAGE OF INDEPENDENT INFORMATION

The storage scheme is straightforward. Since the \( Y_i \)'s satisfy the constraints above, the max-flow, min-cut theorem implies that each user \( U_i \) can reserve \( Y_i \) bit positions in the communal network for his own use. It is also clear that each \( U_i \) should make full use of the \( p_i \) private disks to which he alone has access. Recall that user \( U_i \) is connected to \( n_i \) disks in all, so \( n_i - p_i \) of these disks are in the communal network.

Now each \( U_i \) can be considered independently of all other users. Each user \( U_i \) is effectively connected to \( p_i \) disks of capacity \( C \) and to at most \( n_i - p_i \) additional disks, each with capacity less than or equal to \( C \). These additional disks have total capacity \( Y_i \) bits.

Using the Galois-field information storage scheme given in Chapter 2 for disks of unequal capacities, user \( U_i \) can recover the same amount of information that he could recover if he were told ahead of time that he would lose access to his \( f_i \) most capacious disks.

Thus each user \( U_i \) can reliably recover \((p_i - f_i)C + Y_i\) bits of information, as we wished to show. This completes the proof of the converse. ■

Note: It is not necessary in the previous theorem for all disks to have the same capacity. Reviewing the proof, we see that we can characterize the capacity region in the same way if, for example, every privately held disk has capacity at least as great as every common disk.

Even if there is no such constraint on the \( C_i \)'s, we can still use the proof of Theorem 10 to find inner and outer bounds on the capacity region. However, these bounds are no longer guaranteed to be tight.

4.4 Other Problems to Consider

In this chapter we have considered several classes of problems involving users who store independent information at a set of storage sites, where the storage sites are called disks for concreteness. The networks that we have considered all have capacity regions that can be simply characterized.

It seems likely that Theorem 9 can be modified for the case in which different
users are allowed different numbers of link failures. There is also reason to hope that Theorem 10 might be generalized for the case in which users do not necessarily have access to their own private disks.

Ultimately we might hope to obtain good bounds on the capacity regions for very general problems involving users connected to arbitrary subsets of disks having different disk capacities, with each user being allowed a different number of link failures.
Chapter 5

Floating Parity for Disk Arrays

5.1 Introduction

The problems described in the previous chapters deal primarily with storage efficiency. When users are continually updating the information stored on a set of disks, however, the performance of the disk system can become a more important consideration than storage efficiency. Conventional techniques for creating fault-tolerant disk arrays can significantly increase the time spent modifying old data.

Computer systems use many different methods to prevent loss of data in the event of disk failures. One of the best ways to implement such protection while conserving disk space is a parity technique described by Lawlor [42], Clark et al. [15], and Patterson et al. [55]. This technique requires fewer disks than duplexing (the duplication of all data) but still achieves much faster recovery times than checkpoint-and-log techniques.

The parity technique is a cost-effective way to maintain a fault-tolerant disk array. Its main drawback is that it requires four disk accesses to update a data block—two to read old data and parity, and two to write new data and parity. Due to its relatively poor update performance, the parity technique can degrade disk system performance with respect to a system that does not use this technique.

Menon and Kasson [49] describe four related schemes to improve the write performance of disk arrays that use the parity technique. The schemes all improve write
CHAPTER 5. FLOATING PARITY FOR DISK ARRAYS

performance by sacrificing some storage efficiency and by relaxing the requirement that the modified data and parity blocks be written back into their original locations. In all of the strategies, the updated block can be written to a free location after a delay that is much shorter than the time required for a full revolution of the disk. The average time to update a block is thus improved, and the number of disk accesses is reduced from four to three or even to two.

The four schemes mentioned above can all be analyzed in essentially the same way. In this chapter and the next, we focus on the floating-parity-track technique, which appears to be the best of the four schemes. For simplicity in comparing the floating-parity-track method with the straightforward parity technique, we assume low I/O rates (so that there is no device queuing) and compute only the portion of the service time due to rotational delay of the disks, or disk latency. (The average radial delay, or seek time, is about the same for both techniques.) We find that the floating-parity-track method achieves substantially better single-block update performance than the conventional parity technique, with minimal loss of storage efficiency.

5.2 Ordinary Parity Technique

Before considering the floating-parity technique introduced by Menon and Kasson, let us first examine the conventional parity technique. In Figure 5.1 we illustrate the parity technique on a disk array consisting of three data disks and one parity disk.

<table>
<thead>
<tr>
<th>Data Disk 1</th>
<th>D1</th>
<th>D2</th>
<th>D3</th>
<th>D4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Disk 2</td>
<td>D5</td>
<td>D6</td>
<td>D7</td>
<td>D8</td>
</tr>
<tr>
<td>Data Disk 3</td>
<td>D9</td>
<td>D10</td>
<td>D11</td>
<td>D12</td>
</tr>
<tr>
<td>Parity Disk</td>
<td>P1</td>
<td>P2</td>
<td>P3</td>
<td>P4</td>
</tr>
</tbody>
</table>

Figure 5.1: A parity example with three data tracks and one parity track

We show only one track (consisting of four blocks in a row) from each of the disks.
We call this set of four tracks – three data and one parity – a family\(^1\), and we refer to the number of data tracks in a family as the family size. Note that each track is actually circular, so that the first and last blocks in each row are adjacent.

Since there is one parity track for every three data tracks, the storage efficiency (storage used for data as a fraction of total storage) is \(3/(3+1)\), or 0.75. \(P1\) contains the parity, or exclusive OR (XOR), of the data blocks \(D1, D5,\) and \(D9\). Similarly, \(P2\) is the XOR of the data blocks \(D2, D6,\) and \(D10\), and so on. Such a disk array can survive any single disk crash; if any disk fails, its data can be reconstructed from the other three disks by exclusive-OR operations.

Whenever the disk controller receives a request to write a data block, it must also update the corresponding parity block for consistency. E.g., if \(D3\) is altered, the new value of \(P3\) is calculated as follows:

\[
P3_{\text{new}} = D3_{\text{old}} \oplus D3_{\text{new}} \oplus P3_{\text{old}}.
\]

Since the parity must be updated each time the data is modified, this array requires four disk accesses to update a data block and its associated parity block:

1. Read the old data.
2. Read the old parity.
3. Write the new data.
4. Write the new parity.

In the above situation, the controller writes the new data and parity in the same locations as the old information; these locations are a full disk rotation away just after they are read. The main drawback with this parity technique is that it requires four accesses to update a data block. This chapter describes a modified parity scheme, the floating-parity-track method devised by Menon and Kasson [49], which improves write performance by relaxing the requirement that the modified parity block be written back into its original location. With this modification, the updated parity

\(^1\)In [49], this set is called a "column," but we shall be using the word "column" in a different context.
block can be written to a free location after a delay which is much shorter than the
time required for a full disk revolution.

5.3 Terminology

We use the following definitions in this chapter and the next:

\[T\] number of parity blocks per family
\[F\] number of free parity tracks per family (need not be an integer)
\[m\] number of blocks per track
\[d\] number of disks in the array
\[n\] family size (number of data tracks per family)
\[TS\] size of tables needed in the disk controller
\[D\] delay (in blocks) between reading and writing a parity block
\[E[D]\] expected, or average, delay (in blocks) between reading and writing a parity block

Derived quantities:

\[n/(n + 1)\] storage efficiency with no blocks left free
\[v = Fm\] number of vacant blocks in track (for \(T=1\))
\[TS/(\text{amount of data on disks})\] table overhead
\[L = T - F\] number of \textit{loaded} parity tracks per family
\[R = 1/E[D]\] reciprocal of the expected delay
5.4 Schemes for Improving the Write Performance of the Parity Technique

5.4.1 Overview

In this chapter we shall first describe the double-parity technique, a special case of the floating-parity-track method with $T = 2$ and $F = 1$, and then we shall describe the general floating-parity-track method. After describing the general method, we shall analyze its write update performance in Chapter 6 by computing $E[D]$, the expected delay (in blocks) between reading and writing a parity block.

5.4.2 Double Parity

With the double-parity method, we reserve two parity tracks on a disk to protect data from $n$ other disks, but we use only half of the blocks in the parity tracks at any given time. The parity now occupies $2/n$ times as much space as the data, for a storage efficiency of $n/(n+2)$.

Operation

To understand how the double-parity method works, consider a single family with three data tracks and with four blocks per track. (See Figure 5.2.)

```
| B1 | B2 | B3 | B4 |
| B5 | B6 | B7 | B8 |
| B9 | B10| B11| B12|
| P1 | P2 | P3 | P4 |
| P5 | P6 | P7 | P8 |
```

Figure 5.2: The double-parity method with 3 data tracks

Initially we can store the parity of blocks B1, B5, and B9 in P1; of blocks B2,
B6, and B10 in P3; of blocks B3, B7, and B11 in P5; and of B4, B8, and B12 in P7. Data blocks B1, B5, B9, and parity block P1 constitute a group, and there are four groups per family in the foregoing example. The controller must have a means of knowing which parity blocks are being used and which are available. The controller therefore keeps a bit map, called a free space table, for every family in the array. In our example, each bit map occupies 8 bits, one for each parity block in the family, to indicate which blocks are being used and which are available. When the system is initialized, the bit map might look like this:

1  0  1  0
1  0  1  0

It will be useful to have a term for the set of parity locations in the same sector of the same cylinder (but on different tracks) of a disk. In keeping with the bit-map representation, we shall call this set of parity locations a column. Thus there are as many columns per cylinder as there are blocks per track.²

To keep track of which parity is stored where, the controller stores one parity-address table per family. The parity-address table corresponding to the above bit map would be

1, 3, 5, 7,

indicating that the parity for the first group of blocks is in P1, the parity for the second group is in P3, the parity for the third group is in P5, and the parity for the fourth group is in P7.

Consider an update to data block B7. The controller first determines the block’s family so that it knows which parity-address and free-space tables to access. Then it calculates that block B7 is part of Group 3, and accesses the third entry in the parity-address table. This entry is 5, which tells the controller that the parity for block B7 is currently in location P5. The controller now examines the free-space table, looking for the nearest free block into which the new value of parity may be

²Note that parity locations of a given column all occupy the same physical sector of a disk. The parity value for a logical group, however, can be located anywhere on the parity cylinder.
written. The nearest free block may be on either of the two parity tracks. (Two parity locations in the same column correspond to blocks in the same sector of the disk.) From the free-space table, the controller determines that P2 and P6 are both currently free and that either location may be used to store the new value of parity for the third group. It chooses P6 since this location is adjacent to the current location, P5.

The controller now proceeds as follows:

1. Read the old value from B7.

2. Write the new value of B7 whenever appropriate.

3. Read the old value of P5.

4. In the gap between P5 and P6, exclusive-OR the old values of P5 and B7 with the new value to be written to B7.

5. Write the result of Step 4 to location P6.

The reading of P5 and the writing of P6 are done in a single access to the disk containing the parity tracks for the family being updated. The controller still must wait for a full revolution of the disk containing the data B7 before it can write back the new value.\(^3\) After the controller modifies the parity, it updates the free-space table to

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
0 & 1 & 1 & 0 \\
\end{array}
\]

and the parity-address table to

1, 3, 6, 7

to indicate that the parity of Group 3 is now in P6 and that location P5 is now available for use.

\(^3\)When the number of write requests exceeds some threshold, the controller will actually release the path to the data disk in order to process other requests. A second access will then be required to write back the new value of block B7.
CHAPTER 5. FLOATING PARITY FOR DISK ARRAYS

Performance

Note that at any given time, half of the parity blocks in a family are free. In our example, if P2 and P6 were both occupied, the controller would determine whether P3 or P7 was available. The controller would repeat this procedure until it found a column with at least one block free.

We shall see in the next chapter that, on average, the controller must try about $\sqrt{2}$ (=1.414...) columns before finding a free block. (The expected delay becomes exactly $\sqrt{2}$ blocks as $m$, the number of columns, approaches infinity.) Thus we see that even for moderate values of $m$, the expected delay is much less than the $m$-block delay (i.e., a complete revolution) required for the traditional parity scheme.

Note that the worst case for performance is when the free blocks are all bunched together in adjacent columns. (See the example below with $m = 8$.)

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

In this case, the controller might have to search as many as $m/2$ columns (half a revolution) before finding a free block. As we shall see later, however, patterns like that above occur very infrequently and break up once they have occurred.

Storage Efficiency and Table Overheads

In [49], the storage efficiency (the fraction of all disks which can be used to store data) and the table overhead (the ratio of the table size to the amount of data stored) are computed for arrays of 5 and 10 disks, each with 1000 cylinders, 16 tracks per cylinder, 64 blocks per track, and 512 bytes per block. The results are reproduced in Table 5.1.

The larger the family size, the higher the storage efficiency and the lower the table overheads. For a family size of 8, the double-parity method has high storage efficiency and low table overhead.


5.4.3 Floating Parity Track

The floating-parity-track scheme offers higher storage efficiency than the double-parity method. In this new scheme, each cylinder on every disk in the array is designated either as a parity cylinder (containing only parity tracks) or as a data cylinder (containing only data tracks). Every parity cylinder contains parity information for \((T - F)\) families, where ordinarily \(T\) is the number of tracks per cylinder and \(F\) is the number of tracks left free. Each family consists of \(n\) data tracks from \(n\) different disks and a parity track (of \(m\) blocks) on yet another disk. The \(m\) blocks in a parity track are always in the same cylinder of the same disk, but they need not all be on the same physical track, as will be apparent shortly. Ordinarily the space in a parity cylinder could store parity information for \(T\) families, but the floating-parity-track method leaves \(mF\) free blocks (i.e., \(F\) free tracks, where \(F\) need not be an integer) in the parity cylinder and uses them to enhance performance.

Operation

Consider 3 families, each with 3 data tracks and a parity track. Given a family number and knowledge of the placement algorithm used to store families, we can generate the location of the parity track. See Figure 5.3 for an example with 3 tracks per cylinder and 4 blocks per track. We show all 3 families whose parity is stored in the same parity cylinder.

Initially, \(B_1\) through \(B_{12}\) and \(P_1\) through \(P_4\) together constitute the first family; \(B_{13}\) through \(B_{24}\) and \(P_5\) through \(P_8\) constitute the second family; and \(B_{25}\) through \(B_{36}\) and \(P_9\) through \(P_{12}\) constitute the third family. At the start, the parity for
CHAPTER 5. FLOATING PARITY FOR DISK ARRAYS

Family 1 | Family 2 | Family 3
--- | --- | ---
B5 B6 B7 B8 B17 B18 B19 B20 B29 B30 B31 B32
B9 B10 B11 B12 B21 B22 B23 B24 B33 B34 B35 B36

The 3 families above contain 9 data tracks in all.

P1 P2 P3 P4
P5 P6 P7 P8
P9 P10 P11 P12
P13 P14 P15 P16
P17 P18 P19 P20

This is a parity cylinder with
5 tracks, 3 with parity and 2
left unused.

Figure 5.3: Floating parity with three families

blocks B1, B5, and B9 is stored in P1; for blocks B2, B6, and B10, in P2; for
B3, B7, and B11, in P3; and for B4, B8, and B12, in P4. Blocks P5 through
P8 contain parity for the data blocks B13 to B24, all of which are in the second
family. Finally, P9 through P12 contain the parity for the family of data blocks B25
through B36.

The parity blocks P13 through P20 are initially free. The controller uses a 20-bit
free-space table (one bit for each block in the parity cylinder) for every parity cylinder
on every disk to indicate which blocks are empty and which are occupied. When the
controller initializes the area, the free-space table for one cylinder might look like the
following:

```
1 1 1 1
1 1 1 1
1 1 1 1
0 0 0 0
0 0 0 0
```
For each parity cylinder in the disk array, the controller also stores a *parity-address table* as follows:

1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12.

The parity-address table contains \( m(T - F) \), or \( mL \), entries, one for each group of blocks in every family whose parity is stored in the parity cylinder. In our example, the controller stores parity for three different families in the same parity cylinder, and there are four groups of blocks in each of these three families, requiring 12 entries in the parity-address table. The table indicates that parity for the first group of blocks from the first of the three families is in \( P_1 \), parity for the second group of blocks from the first family is in \( P_2 \), and so on. The last entry indicates that the parity for the last group of blocks from the third family is in \( P_{12} \).

Consider an update of block \( B_7 \). The controller first determines the block’s family, so it knows which parity cylinder contains the parity information for block \( B_7 \). Thus it knows which parity-address and free-space tables to access. It also determines that the family of interest is the first of the three families whose parity is stored on that parity cylinder. Then it calculates that block \( B_7 \) is part of Group 3, and it accesses the third entry in the parity-address table. This entry is 3, which tells the controller that the parity for block \( B_7 \) is currently in location \( P_3 \). From the free-space table, the controller ascertains whether at least one of the blocks \( P_4, P_8, P_{12}, P_{16}, \) and \( P_{20} \) is free, and establishes that \( P_{16} \) and \( P_{20} \) are free. Assuming for the sake of definiteness that the controller chooses the free track with the lowest number, it stores the new parity information in position \( P_{16} \). If none of those five blocks had been free, the controller would have moved to the next column and checked whether one of \( P_1, P_5, P_9, P_{13}, \) and \( P_{17} \) was free, and so forth.

The controller now proceeds as follows:

1. Read the old data from block \( B_7 \).

2. Write the new value of \( B_7 \) whenever appropriate.

3. Read the old value of \( P_3 \).
4. In the gap between P3 and P16, exclusive-OR the old values of P3 and B7 with the new value to be written to B7.

5. Write the result from Step 4 to location P16.

The controller reads P3 and writes P16 in a single access to the disk containing the parity tracks for the family being updated. After the parity is updated, the controller updates the free-space table as follows:

```
 1 1 1 1 1 1 1 0 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0
```

The controller also updates the parity-address table to

1, 2, 16, 4, 5, 6, 7, 8, 9, 10, 11, 12,

indicating that the parity of the third group in the first family is now in P16 and that location P3 is now available for use.

**Performance**

At any given time, a parity cylinder has \(mF\) of its \(mT\) blocks free, so the probability that any particular block is free is \((mF)/(mT) = F/T\). Consider once again the update of P3. The new parity can be written one block away if any of P4, P8, P12, P16, and P20 is free. In general, the controller can write the new parity after a delay of a single block if any of the \(T\) blocks in the next column are free.

We shall see that in all cases of interest, the controller will, on average, try no more than 2 columns before finding a free block; i.e.,

\[ E[D] \leq 2. \]
CHAPTER 5. FLOATING PARITY FOR DISK ARRAYS

Note that the worst case for performance occurs when the free blocks are all bunched together in adjacent columns. (See the example below with $T = 4$ tracks per cylinder, $m = 8$ blocks per track, and $F = 1$ free track.)

\[
\begin{array}{cccccccc}
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
\end{array}
\]

In this case, the controller might have to search as many as $m - (m/T)$ columns (almost a full revolution of the disk) before finding a free block. We shall see, however, that patterns like the one above are infrequent and short-lived.

Storage Efficiency and Table Overheads

In [49] the storage efficiency and table overhead are computed for arrays of 5 and 10 disks, each with 1000 cylinders, 16 tracks per cylinder, 64 blocks per track, and 512 bytes per block. The results are reproduced in Table 5.2.

<table>
<thead>
<tr>
<th></th>
<th>$n=8, d=10, \frac{F}{T}=1$</th>
<th>$n=4, d=5, \frac{F}{T}=1$</th>
<th>$n=4, d=5, \frac{F}{T}=2$</th>
<th>$n=4, d=5, \frac{F}{T}=3$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Storage (bytes)</td>
<td>$4.6 \times 10^9$</td>
<td>$2.1 \times 10^9$</td>
<td>$2.0 \times 10^9$</td>
<td>$2.0 \times 10^9$</td>
</tr>
<tr>
<td>Storage Efficiency</td>
<td>0.88</td>
<td>0.789</td>
<td>0.778</td>
<td>0.765</td>
</tr>
<tr>
<td>Table Storage (bytes)</td>
<td>$1.56 \times 10^6$</td>
<td>$1.40 \times 10^6$</td>
<td>$1.39 \times 10^6$</td>
<td>$1.37 \times 10^6$</td>
</tr>
<tr>
<td>Table Overhead</td>
<td>$3.38 \times 10^{-4}$</td>
<td>$6.75 \times 10^{-4}$</td>
<td>$6.80 \times 10^{-4}$</td>
<td>$6.85 \times 10^{-4}$</td>
</tr>
</tbody>
</table>

Table 5.2: Storage efficiency and table overheads for floating parity

A larger family size yields higher storage efficiency and lower table overhead. Increasing the number of tracks left free leads to slightly lower storage efficiency and slightly higher table overhead.
5.5 Summary

We have reviewed the floating-parity-track technique devised by Menon and Kasson [49]. This technique improves the update performance of disk arrays by reducing the delay between reading a parity block and writing it back onto the disk. This improvement in performance can be achieved without sacrificing a great deal of storage space.

In the next chapter we shall analyze the floating-parity-track technique theoretically to determine how much the technique actually reduces the average delay between reading and writing a parity block.
Chapter 6

Analysis of the Floating-Parity Scheme

6.1 Introduction

In this chapter we theoretically analyze the performance of the previously described floating-parity-track technique for improving the write performance of fault-tolerant disk arrays. We begin by analyzing relatively simple cases and then consider progressively more general situations.

Initially we assume that every updated parity block is always written back into the same track from which it was read. Thus there is effectively just one parity track \( T = 1 \). We also assume that for every transition, each parity block is equally likely to be the parity block updated (no-skew assumption). This property is assumed to hold independent of which blocks have recently been updated (independence assumption).

Under these assumptions, we can apply standard results from Markov chain theory to show that \( E[D] \), the expected delay in blocks between reading and writing a parity block, is approximately \( 1/F \), where \( F \) is the fraction of each parity track that is left free. For example, if half of each track is left free, then the expected delay is approximately 2 blocks.

Using the result obtained above, we compute \( E[D] \) in more general cases, first dropping the no-skew assumption, then allowing parity blocks to be written to any of
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

$T$ different tracks, where $T$ is allowed to be 2 ("double parity") or any positive integer (general floating parity). Finally, we allow the skew at each time to depend on the last $M$ parity blocks updated, for arbitrarily long memory $M$; that is, we consider arbitrary $M$th-order Markov skew.

In all cases of practical interest, we find that the expected delay, $E[D]$, is less than 2 blocks, a small fraction of an entire track since there are typically 64 blocks per track. If floating parity is not used, the delay between reading and writing a parity block corresponds to an entire track, i.e., an entire disk revolution. We conclude that the floating-parity-track technique described in the previous chapter is an effective way to reduce the delays associated with fault-tolerant disk arrays due to disk latency.

6.2 Single Track ($T = 1$)

6.2.1 Without Skew

We wish to compute $E[D]$, the expected delay in blocks from the time that a parity block is read until the time that a free location is found into which its updated value can be written. We hope to show that this average delay is much less than a full disk revolution, i.e., that $E[D] \ll m$.

Let us begin by considering the case in which we use just one parity track per family, leaving the track partially empty in order to improve performance. Suppose, for example, that $T = 1$ (one parity track), $F = 1/2$ (i.e., half of the blocks in the track are free at any given time), and $m = 4$ (four blocks per track). Also suppose that there are 2 tracks per cylinder. Then our initial bit map might be as follows:

\[
\begin{array}{cccc}
1 & 0 & 1 & 0 \\
1 & 0 & 1 & 0 \\
\end{array}
\]

The solid line in the figure above indicates that parity blocks do not cross track boundaries; although there are two tracks per parity cylinder, the fact that $T = 1$ means that the two tracks do not interact.
A possible evolution of the top row of the bit map is shown in the example below. Associated with each transition is the delay, \( D \), the number of blocks that the controller must examine before finding a free block position in the track that contains the parity block being updated.

**Notation:** For each transition, the starting position of the parity block to be moved is indicated by a single dot over the "1" in the initial bit map, and the resulting position of the block is indicated by a double dot over the "1" in the subsequent bit map. The delay (the number of block positions that the block must move to find a vacancy) is indicated above the arrow leading from one state to the next. Recall that each track is actually circular; the first and last positions are adjacent. We assume that blocks move from left to right.

**Example:**

\[
\begin{array}{cccccccc}
  i & 0 & 1 & 0 & D=1 & 0 & 1 & 1 & 0, & 0 & 1 & 1 & 0 & D=2 & 0 & 0 & 1 & 1
\end{array}
\]

Now that the parity-update procedure is clear, let us calculate \( E[D] \), the expected delay (in blocks) when \( T = 1 \). Assume for now that all parity blocks are updated with equal probability; i.e., assume that there is no skew.

The limiting value of the expected delay, \( E[D] \), can be derived using standard results from the theory of Markov chains. Here we give a few preliminary definitions and then outline the derivation.

**Definition:** Let \( v \) be the number of vacant blocks in a parity track. Then \( v = Fm \), and \( m - v \) is the number of positions in the track occupied by parity blocks.

Consider the *state* of the parity-track system to be the positions of the \( v \) zeros and the \( m - v \) ones in the bit map. Then the number of different possible states is

\[
\binom{m}{v} = \frac{m!}{v!(m-v)!}
\]

**Example:** \( m = 3, F = 1/3 \Rightarrow v = mF = 1. \)
We have 3 possible states: 110 or 101 or 011.

\[
\binom{m}{v} = \frac{3!}{1!2!} = \frac{6}{(1)(2)} = 3, \text{ as required.} \quad \blacksquare
\]

We can construct a Markov chain—a sequence of transitions from one state to another—by determining the probability that any given state will lead to any other given state when a parity block is updated and moved.

**Example** (continued): If we begin in the state 110, then because both parity blocks are updated with equal probability, we have

\[
\begin{align*}
&110 \rightarrow 01\bar{1} \quad \text{with probability } 1/2, \\
&110 \rightarrow 10\bar{1} \quad \text{with probability } 1/2.
\end{align*}
\]

Similarly, if we start in state 101, we have two possible (and equally likely) transitions:

\[
\begin{align*}
&101 \rightarrow 01\bar{1} \quad \text{with probability } 1/2, \\
&101 \rightarrow 1\bar{1}0 \quad \text{with probability } 1/2.
\end{align*}
\]

Finally, if we start in state 011, we have two possible (and equally likely) transitions:

\[
\begin{align*}
&011 \rightarrow \bar{1}01 \quad \text{with probability } 1/2, \\
&011 \rightarrow \bar{1}10 \quad \text{with probability } 1/2.
\end{align*}
\]

We can represent all possible transitions and their probabilities in a transition matrix, \( P \), as follows:

<table>
<thead>
<tr>
<th>Old state</th>
<th>New state</th>
</tr>
</thead>
<tbody>
<tr>
<td>110</td>
<td>10 0 1/2 1/2</td>
</tr>
<tr>
<td>101</td>
<td>1/2 0 1/2</td>
</tr>
<tr>
<td>011</td>
<td>1/2 1/2 0</td>
</tr>
</tbody>
</table>
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

Then

\[
P = \begin{pmatrix}
0 & 1/2 & 1/2 \\
1/2 & 0 & 1/2 \\
1/2 & 1/2 & 0
\end{pmatrix}.
\]

In general, \( P_{x,y} \), or \( P(x,y) \), is the probability of going to state \( y \), given that we start in state \( x \). If \( x = s_i \) (the \( i^{th} \) state) and \( y = s_j \) (the \( j^{th} \) state), then \( P(x,y) \) is found in the \( i^{th} \) row and \( j^{th} \) column of the transition matrix \( P \). Note that \( P(x,y) \) is a conditional probability, not a joint probability.

Using properties of the Markov chain defined above, we can compute the expected delay, \( E[D] \), as follows. (See Hoel, Port, and Stone [33] for definitions and basic results pertaining to Markov chains.)

For one track without skew, the Markov chain defined on the states is irreducible; i.e., the state does not get trapped in any proper subset of the possible states. It then follows that there is a unique stationary distribution on the set of possible states. (A stationary distribution on a Markov chain is an equilibrium distribution, i.e., a probability distribution over the possible states that is unchanged after a single (random) transition.)

For the case with one parity track and no skew, it is easy to show that the transition matrix \( P \) for the Markov chain is doubly stochastic. That is, each row and column of \( P \) sums to 1. This fact implies that the stationary distribution of the Markov chain is uniform; that is, all possible states occur with equal probability.

In addition to being irreducible and having a doubly stochastic transition matrix, the Markov chain defined on the states is aperiodic. Very roughly, this means that the state does not cycle indefinitely with any fixed period. Under this condition, the probability distribution on the state must converge to the (uniform) stationary distribution. In other words, the stationary distribution is also the steady-state distribution.

The previous results imply that all states are equally likely in steady state. Since there is no skew, every parity block is also equally likely to be updated on any given turn. It is then straightforward to find the expected delay, \( E[D] \), as the average number of block positions that must be searched before a vacancy is found for the
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

block being updated. An induction argument shows that in steady state,

$$E[D] = m/(v + 1) \text{ blocks,}$$  \hspace{1cm} (6.1)

where \( m \) is the number of columns (i.e., the number of blocks per track) and \( v \) is the total number of vacant block positions.

Note that as \( m \to \infty \), in steady state,

$$E[D] = m/(v + 1) = m/(mF + 1) \to 1/F,$$  \hspace{1cm} (6.2)

where \( F = v/m \) is the fraction of parity blocks that are left free.

6.2.2 Single Track, With Skew

The computation just outlined presumes that there is no skew, i.e., that for each block update, all parity blocks are equally likely to be updated. Now we drop the assumption that each data block is updated with the same probability. More precisely, we drop the assumption that each parity block (or equivalently, each logical group of data blocks) is updated with equal probability. We assume only that each parity block (equivalently, each logical group of data blocks) is updated with nonzero probability.

**Example:** Let \( m = 3 \), \( L = 2/3 \). Then \( F = 1 - L = 1/3 \), so \( v = Fm = 1 \). A possible state of the system is shown below:

\[ \text{P1 P2 0} \]

We suppose that parity block \( \text{P1} \) is chosen with probability \( p_1 = 0.6 \), while \( \text{P2} \) is chosen with probability \( p_2 = 0.4 \). \[ \blacksquare \]

Now we must distinguish among the different blocks, since they are not chosen with equal probability. The total number of states of the system is

$$|S| = m(m - 1)(m - 2) \cdots (v + 1) = \frac{m!}{v!}$$  \hspace{1cm} (6.3)

**Example (continued):** In the example above, \( m!/v! = 3!1! = 6 \). The six states are shown below:
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

\[ \begin{array}{c|c|c|c|c|c|c|c|}
\text{P1} & \text{P2} & \text{0} & \text{P1} & \text{0} & \text{P2} & \text{0} & \text{P1} & \text{P2} \\
\text{P2} & \text{P1} & \text{0} & \text{P2} & \text{0} & \text{P1} & \text{0} & \text{P2} & \text{P1} \\
110 & 101 & 011 & \\
\end{array} \]

As usual, the parity track is circular, so the first and last blocks in the track are adjacent. Vacant positions are represented by zeros. Beneath each pair of configurations having the same vacant positions is the bit-map representation of the state that would be used if the parity blocks were not distinguishable.

As in the case without skew, we can construct a state-transition matrix whose \((i,j)^{th}\) entry is the probability of going to state \(j\) in a single transition, given that we begin in state \(i\).

Using an obvious shorthand to represent the six states of this system, we obtain the following state-transition matrix:

\[
\begin{array}{c|ccccccc}
\text{Old state} & \text{New state} & 120 & 210 & 102 & 201 & 012 & 021 \\
120 & 0 & 0 & 0.4 & 0 & 0 & 0.6 \\
210 & 0 & 0 & 0 & 0.6 & 0.4 & 0 \\
102 & 0.4 & 0 & 0 & 0 & 0.6 & 0 \\
201 & 0 & 0.6 & 0 & 0 & 0 & 0.4 \\
012 & 0 & 0.4 & 0.6 & 0 & 0 & 0 \\
021 & 0.6 & 0 & 0 & 0.4 & 0 & 0 \\
\end{array}
\]

For general \(m\) and \(v\) with the state defined so as to distinguish among different parity blocks, we repeat the analysis outlined for the no-skew case. As before, we find that the Markov chain of interest is irreducible and has a uniform stationary distribution. The Markov chain is not always aperiodic, but its period is always 1 or 2. These conditions suffice to show that the limiting expected delay is the same as in
the no-skew case; i.e., in steady state,

$$E[D] = \frac{m}{v + 1}.$$  \hspace{1cm} (6.4)

### 6.3 Multiple Tracks ($T > 1$)

#### 6.3.1 Without Skew

**Double Parity ($T = 2, \ L = 1$)**

In the last section we found the expected delay between reading and writing a parity block when the floating-parity-track method is used on each track individually. Now let us analyze the method's performance in more complicated cases.

As before, we wish to compute $E[D]$, the expected delay (in blocks) for the floating-parity-track method. We hope to show that the average delay is much less than one disk revolution (i.e., that $E[D] \ll m$, where $m$ is the number of blocks per track).

We begin by considering the case $T = 2, \ L = T - F = 1$ (i.e., 1 track “free” and 1 track “loaded”). This special case corresponds to the double-parity method described earlier. The analysis of the general case will be essentially the same as the analysis of this particular case.

As in the previous section, we define a system state and an associated Markov chain.

**Definition:** For the floating-parity-track method with $T = 2$ and $L = T - F = 1$, define the state of the relevant Markov chain to be the number of parity blocks in each column (0, 1, or 2) at a given time.

**Example:** If $m = 4$, then one possible state is $s = (1 \ 0 \ 2 \ 1)$, which can arise from any of the following four bit maps:

$$
\begin{pmatrix}
0 & 0 & 1 & 0 \\
1 & 0 & 1 & 1 \\
\end{pmatrix}, \quad
\begin{pmatrix}
0 & 0 & 1 & 1 \\
1 & 0 & 1 & 0 \\
\end{pmatrix}, \quad
\begin{pmatrix}
1 & 0 & 1 & 0 \\
0 & 0 & 1 & 1 \\
\end{pmatrix}, \quad
\begin{pmatrix}
1 & 0 & 1 & 1 \\
0 & 0 & 1 & 0 \\
\end{pmatrix}.
$$
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

Note that the sum of the column values of \( s \) is equal to \( Lm = 1(4) = 4 \), the number of loaded tracks times the number of blocks per track.

As usual, we define a Markov chain over the set of all possible states, and we define a transition matrix \( P \) which gives the probability of moving from any initial state to any target state as a result of updating a single parity block.

Using properties of the Markov chain defined above, we can compute \( E[D] \), the expected delay in blocks, as follows.

For two tracks without skew, there is a unique stationary distribution on the set of possible states. This distribution is not uniform, but it has the simple form given in Theorem 11.

**Theorem 11** When \( T = 2 \), \( L = 1 \), and \( m \geq 3 \), there is a unique steady-state distribution on \( S \), the set of all states. Let \( S \) be the random variable representing the state. For the state \( s = (x_1, x_2, \ldots, x_m) \), where \( x_i \) is the number of parity blocks in Column \( i \), we have

\[
\Pr\{S = s\} = cp_{x_1}p_{x_2} \ldots p_{x_m}, \quad \text{where}
\]

\[
p_0 = 1 - \frac{\sqrt{2}}{2} \approx .293 \quad \text{if } x_i = 0,
\]

\[
p_1 = \sqrt{2} - 1 \approx .414 \quad \text{if } x_i = 1,
\]

\[
p_2 = 1 - \frac{\sqrt{2}}{2} \approx .293 \quad \text{if } x_i = 2
\]

and \( c \) is a normalizing constant chosen so that \( \sum_{s \in S} \Pr\{S = s\} = 1 \).

**Proof (Outline):** The relevant Markov chain is easily shown to be irreducible, so it has a unique stationary distribution. Thus we need only verify that the distribution given in this theorem is a stationary distribution. The verification is not difficult.

The Markov chain defined on the states is irreducible and aperiodic, so the probability distribution on the states converges to the (unique) stationary distribution. I.e., the stationary distribution is also the steady-state distribution.
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

Now given any number of columns \( m \geq 3 \), we can use the steady-state distribution found above together with the results from the single-track case (Section 6.2) to demonstrate the following theorem.

**Theorem 12** For \( T = 2 \), \( L = 1 \) (double parity) with no skew and with \( m \geq 3 \) (i.e., with at least 3 blocks per parity track), in steady state,

\[
E[D] = \frac{\sum_{m_2=0}^{\left\lfloor \frac{m}{2} \right\rfloor} \left\{ \frac{m!(\frac{1}{2})^{m_2}}{(m_2)!^2(m-2m_2)!} \left[ \frac{m-2m_2}{m-m_2} + \frac{2m_2}{m-m_2+1} \right] \right\}}{\sum_{m_2=0}^{\left\lfloor \frac{m}{2} \right\rfloor} \frac{m!}{(m_2)!^2(m-2m_2)!}}.
\]

**Proof (Outline):** Let \( m_2 \) be the number of columns that contain 2 parity blocks at a particular time. For any fixed value of \( m_2 \), the results for the single-track case can be modified slightly to show that

\[
E[D] = \frac{(m - 2m_2)}{(m - m_2)} + \frac{(2m_2)}{(m - m_2 + 1)}.
\]

We then take a weighted average of the expressions above as \( m \) ranges over its possible values; the weight that we use is just the probability that exactly \( m \) columns will be full.

See Figure 6.1 for the value of the expected delay, \( E[D] \), as a function of the number of columns, \( m \). Note that \( E[D] \rightarrow 1.414... \) as \( m \rightarrow \infty \) and that the expected delay is always less than 2 blocks, a small fraction of a track.

As the number of blocks per track becomes large (i.e., as \( m \rightarrow \infty \)), the asymptotic equipartition property (a form of the law of large numbers) implies that the Markov chain is almost certain to be in one of its “typical” states. I.e., if \( m_0 \), \( m_1 \), and \( m_2 \) are the numbers of columns containing 0, 1 and 2 blocks, respectively, then as \( m \rightarrow \infty \),

\[
\frac{m_0}{m} \rightarrow p_0 = 1 - \frac{\sqrt{2}}{2} \approx .293,
\]

\[
\frac{m_1}{m} \rightarrow p_1 = \sqrt{2} - 1 \approx .414, \text{ and}
\]

\[
\frac{m_2}{m} \rightarrow p_2 = 1 - \frac{\sqrt{2}}{2} \approx .293.
\]
For any given value of \((m_0, m_1, m_2)\), however, all states of this given "type" (i.e., all states with the given number of 0-block, 1-block, and 2-block columns) are equally likely. It then follows readily that \(E[D] \to 1/(1-p_2) = \sqrt{2} \approx 1.414\) blocks as \(m \to \infty\), after a sufficient number of block updates.

**Multiple Tracks, Without Skew: General \(T\) and \(L\)**

Having computed the expected delay \(E[D]\) in the case of double parity \((T = 2, L = 1)\), we extend our results to the case of an arbitrary number of tracks, \(T\), an arbitrary number of which \((L)\) are occupied, or "loaded," at any given time. \(L\) need not be an integer, but \(mL\), the total number of occupied block positions, must be an integer.

As usual, we define a system state and a Markov chain on the set of all possible states.

**Definition:** Given a parity cylinder with \(T\) tracks and \(m\) blocks per track with \(L\) tracks (that is, \(mL\) blocks, where \(L\) need not be an integer) occupied by ("loaded" with) parity blocks at any given time, define the state of the Markov

Figure 6.1: Expected delay for double parity
chain associated with the floating-parity-track method to be the number of
parity blocks currently in each column (either 0, 1, 2, ..., or T).

As in the previous sections, we define a Markov chain over the set of all possible
states, and we define a transition matrix $P$ which gives the probability of moving
from any initial state to any target state as a result of updating a single parity block.

Using properties of the Markov chain defined above, we can compute $E[D]$, the
expected delay in blocks, in several stages. First we find that there is a unique
stationary distribution on the set of all possible states, similar to that found for
the special case of double parity. This distribution can be characterized in terms of
a certain probability vector $(p_0, p_1, \ldots, p_T)$, where the $p_i$'s satisfy a set of $(T + 1)$
equations in $(T + 1)$ unknowns. More specifically, we have the following theorem,
valid for all values of $m$, $L$, and $T$ of practical interest.

**Theorem 13** If $m \geq 3$ and $mL \geq T+1$, then there is a unique stationary
distribution on $S$, the set of all states. If the state $s$ is equal to $(x_1, x_2, \ldots, x_m)$, where $x_i$ is
the number of parity blocks in the $i^{th}$ column at a particular time, then $Pr\{S = s\} = c p_{x_1} p_{x_2} \cdots p_{x_m}$, where for $1 \leq i \leq m$ we have $x_i \in \{0,1,\ldots,T\}$ and $p_{x_i} \in
\{p_0,p_1,\ldots,p_T\}$.

The $p_i$'s are determined by the following set of $(T+1)$ equations in $(T+1)$ unknowns:

(*) $p_0 + p_1 + \ldots + p_T = 1$ (since each column must have either 0,1,..., or $T$ parity
blocks)

(**) $0p_0 + 1p_1 + 2p_2 + \ldots + Tp_T = L$ (since the expected number of blocks in the parity
cylinder must be $mL$)

(***) $\frac{p_1}{p_0} = \frac{2p_2}{p_1} = \ldots = \frac{Tp_T}{p_{T-1}}$. (These equations ensure that the distribution is
stationary.)

Finally, $c$ is a normalizing constant chosen so that $\sum_{s\in S} Pr\{S = s\} = 1.$
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

Proof (Sketch): The existence of a unique stationary distribution is shown as in the problems considered earlier. Any vector \((p_0, p_1, \ldots, p_T)\) satisfying (*)\((**)\), and (***) in the previous theorem can be shown to yield a stationary distribution when substituted into the formula for \(\Pr\{S = s\}\).

For values of \(m\), \(L\), and \(T\) of practical interest, the Markov chain defined above is aperiodic, so it has a unique steady-state distribution, equal to the stationary distribution.

Given any number of columns \(m \geq 3\), we can use the results from the single-track case to compute the expected delay as a weighted average of the quantity

\[
\left( \frac{Lm - Tm_T}{Lm} \right) \left( \frac{m}{m - m_T} \right) + \left( \frac{Tm_T}{Lm} \right) \left( \frac{m}{m - m_T + 1} \right) \rightarrow \frac{m}{m - m_T} \quad \text{as} \quad m \rightarrow \infty,
\]

where \(m\) is the (variable) number of columns that are fully occupied with \(T\) parity blocks.

As the number of blocks per track becomes large (i.e., as \(m \rightarrow \infty\)), the asymptotic equipartition property implies that the Markov chain is almost certain to be in one of its “typical” states. That is, if \(m_0, m_1, \ldots, m_T\) are the numbers of columns containing 0, 1 \ldots, \(T\) blocks, respectively, then \(m_j/m \rightarrow p_j\) for \(0 \leq j \leq T\).

For any given value of \((m_0, m_1, \ldots, m_T)\), all states of this “type” (i.e., all states characterized by \((m_0, m_1, \ldots, m_T)\)) are equally likely. It then follows readily that in steady state, \(E[D] \rightarrow 1/(1 - p_T)\) blocks as \(m \rightarrow \infty\). It can be shown without difficulty that if the vector \((p_0, p_1, \ldots, p_T)\) satisfies (*), (**), and (***)\), then \(p_T = 1 - R\), where \(R\) satisfies the equation

\[
(1 - R) \sum_{j=0}^{T-1} \frac{T!}{(T-j-1)!} \left( \frac{R}{L} \right)^j = L.
\]

On the other hand, a continuity argument shows that there must be a value of \(R\) between 0 and 1 that satisfies the equation above. Computing the values of \(p_T\) for various values of \(T\), letting \(L = T - 1\) (i.e., leaving one track free), we obtain Figure 6.2. Note that in all cases, \(E[D] \leq 2\) blocks, a small fraction of the delay associated with an entire disk revolution.
6.3.2 Multiple Tracks \((T > 1)\), With Skew

We shall see that the results of the previous subsection, derived for the no-skew case, hold even when there is skew.

As in the previous sections, we begin our search for the expected value of the delay by defining a system state and an associated Markov chain. When the parity blocks are not necessarily updated with equal probability, the state must distinguish among the different parity blocks; it is not enough simply to count the number of parity blocks per column.

**Definition:** Let the *state* of the relevant Markov chain be the *set* of parity blocks in each of the \(m\) columns, without regard for the tracks which the blocks occupy.

**Example:** Suppose that \(T = 3\) parity tracks, \(L = 2\) tracks loaded (so that \(F = T - L = 1\) track free), and \(m = 3\) columns (i.e., 3 blocks per track). Then there are \(mL = 6\) different parity blocks, which we call \(P_1, P_2, ..., P_6\).
CHAPTER 6. ANALYSIS OF THE FLOATING-PARITY SCHEME

If the blocks are arranged as follows,

\begin{equation}
\begin{array}{ccc}
P1 & P3 & P5 \\
P2 & P4 & P6 \\
\end{array}
\end{equation}

then we write the state as \( s = (\{1, 2\}, \{3, 4\}, \{5, 6\}) \).

If the blocks are arranged as below,

\begin{equation}
\begin{array}{ccc}
P5 & P1 & P3 \\
- & P6 & P4 \\
- & - & P2
\end{array}
\end{equation}

then we write the state as \( s = (\{5\}, \{1, 6\}, \{2, 3, 4\}) \). (Note that we do not care about the order of the blocks within a column, so we represent the third column as \( \{2, 3, 4\} \) and not as \( \{3, 4, 2\} \).)

Using properties of the Markov chain defined above, we can compute \( E[D] \), the expected delay in blocks, as follows. With the state defined to distinguish among the different parity blocks, the Markov chain converges to a unique steady-state distribution, and this steady-state distribution is uniform over all possible states. Thus for the general floating-parity problem, it is in some sense more natural to consider the “microstates” defined above (for the case with skew) than to consider the “macrostates” defined previously (for the case without skew).

Because the steady-state distribution on microstates is uniform, hence independent of the skew, it follows immediately that the distribution on macrostates must also be independent of the skew (though not uniform). It is then straightforward to show that \( E[D] \), the expected delay in blocks, is the same with skew as without. (As always, though, there is the restriction that all blocks must be updated with nonzero probability.)

6.3.3 Dependent Case

We have seen in the previous sections that results derived assuming no skew have turned out to be true even in the case of skew, provided that all blocks are updated with nonzero probability. The proofs for the cases without skew were readily adapted
to the more general cases with skew simply by redefining the state, i.e., by going from "macrostates" to "microstates."

Having dispensed so readily with the requirement that parity blocks be updated with the same probability, we might also try to dispense with the requirement of independence. Suppose that we allow the skew at any given time to be affected by the recent past. In particular, suppose that the block-update probability distribution at any time is a function of the last $M$ blocks to be updated, where $M$ is the memory of the system.

It can be shown that the earlier analysis goes through without major changes if the state is redefined once more to include the last $M$ parity blocks updated as well as the columns in which all parity blocks are currently stored. The formulas derived earlier for $E[D]$, under the assumption of independence, are still valid in the case of $M^{th}$-order Markov skew, when the probability of a particular parity block being updated is allowed to be a function of the last $M$ parity blocks chosen.

6.4 Summary of Analysis

We have analyzed the performance of the floating-parity-track technique described in Menon and Kasson [49] for preventing the loss of data in the event of a disk failure. This technique seeks to reduce the delay incurred by the straight-forward parity technique described by Lawlor [42], Clark et al. [15], and Patterson [55]. We have compared the two techniques by comparing the average time required to update a parity block.

For the straightforward technique, the time required is essentially equal to the time for the disk to make one complete revolution. Since there are $m$ blocks per track, we can write the expected delay for this technique as

$$E[D] = m \text{ blocks.}$$

For the floating-parity technique, we have shown in this chapter that in all cases of interest (e.g., if we leave at least one parity track free), the average delay is negligible compared to the time required for the disk to make a complete revolution. We have
seen that generally

\[ E[D] \leq 2 \text{ blocks} = \frac{2}{m} \text{ revolutions,} \]

where typically \( m \) is on the order of 64 blocks per track.

Thus the floating-parity-track method does indeed significantly reduce the delay associated with the straightforward parity technique. Furthermore, this improvement in performance can be obtained with a very small increase in storage space if \( T \), the number of parity tracks in a cylinder, is much greater than one.

### 6.5 Other Problems to Consider

The floating-parity-track technique analyzed in this report might be improved in several ways. So far we have only tried to reduce the delay between reading a parity block and writing its new value back to its parity cylinder. However, we still must wait half a disk revolution on average before reading the old parity value. This delay might be reduced by duplicating parity on different disks. Because the parity occupies a relatively small fraction of the total disk space, it might be acceptable to double the space allocated for parity in order to improve performance.

As noted in [49], read caching may also be used to improve write performance in disk arrays, since such caches can often eliminate the need to read data and parity from the disk prior to writing new data and parity. Write caching with nonvolatile storage may be employed too, to improve disk array subsystem performance.

Until now, we have assumed that each data block has only one parity block to serve as a check. However, we might extend the previously mentioned techniques for use with more powerful parity schemes which can tolerate the simultaneous failure of two or more disks. Such an extension would make the schemes described in Chapter 2 more attractive to system designers.
Chapter 7

Summary

7.1 Conclusions

The previous chapters have addressed a number of problems involving multiple users who simultaneously store information in a network of storage sites. Different storage sites—often called disks for concreteness—have been allowed to have different capacities. Different users have been allowed access to different (perhaps overlapping) sets of disks and have been permitted to store either common or independent information. We have considered cases in which different users can tolerate different numbers of disk or link failures.

The problems that we have already identified may be thought of as the extreme points of general theory of network information storage. The general theory would include all of these extreme points as special cases. Based on the results already obtained, we are hopeful that even the most general problems may be susceptible to solution.

The general results on reliable information storage derived in the first four chapters are specialized in Chapters 5 and 6 to a particular configuration of disks widely used in practice. In this configuration, there is effectively a single user with access to an array of \( n \) disks, with at most disk allowed to be inaccessible at any given time.

Although the information storage scheme for the above configuration is simple in principle, there are practical difficulties, because of the need to update parity-check
CHAPTER 7. SUMMARY

blocks every time data blocks are modified. As we have seen in Chapter 5, a storage technique called floating parity track, developed by Menon and Kasson [49], may be used to reduce the delays associated with updating data and parity blocks.

The theoretical analysis in Chapter 6 of the floating-parity technique described in the previous chapter shows that the average delay between reading and writing a parity block can typically be reduced from a full disk revolution (approximately 10 milliseconds) to less than 1/32 of a disk revolution (approximately 300 microseconds). As indicated in [49], this improvement in performance can be achieved with minimal loss of storage efficiency.

7.2 Suggestions for Further Research

Chapters 2, 3, and 4 identify a number of problems related to distributed information storage. Ultimately we might hope to solve the general problem in which different users store both common information and independent information, have access to arbitrary overlapping subsets of disks with arbitrary capacities, and can tolerate different numbers of disk and link failures. A more modest goal might be to find the storage capacity region for an arbitrary network of disks all having the same capacity, where the users store independent information and can tolerate different numbers of link failures.

An interesting area of research lies in the gap between the problems described in this thesis and the file-distribution problem considered by Naor and Roth [53]. In this dissertation we have taken network topologies and disk storage capacities as given and have tried to determine how much information each user can reliably recover. Naor and Roth also take the network topology as given, but they choose the disk capacities so that each user can recover a given amount of information. Their goal is to find the minimum total disk capacity that allows each user to recover the information.

In the setup of Naor and Roth, it is assumed that no link or node failures occur. It may be possible to extend their results, however, to allow link and node failures. Such an extension would help to bridge the gap between the classes of problems considered by Naor and Roth and those considered in this dissertation.
CHAPTER 7. SUMMARY

In Chapter 6, we have theoretically analyzed the performance of the floating-parity-track technique of Menon and Kasson [49] for improving the update performance of disk arrays. It would be useful, however, to extend the analysis to take into account the effects of read and write caching and of multi-block updates. The complications introduced by caching might make exact analysis difficult for the more general problem. Simulation should still provide insight, however, into the performance of the floating-parity-track technique.
Bibliography


BIBLIOGRAPHY


