Highly Available Distributed Storage Systems

Thesis by

Lihao Xu

In Partial Fulfillment of the Requirements

for the Degree of

Doctor of Philosophy

California Institute of Technology

Pasadena, California

1999

(Submitted Nov. 30, 1998)

To *Lan*

for her love

Acknowledgements

I am deeply grateful to my advisor, Prof. Jehoshua Bruck, for his continuous support, guidance and encouragement during my study at Caltech. Working with him has always been fun, not only because of his high creativity, broad knowledge and insightful vision, but more importantly because of his amiable personality. Discussions with him are not just enjoyable, they are always a good source of inspiration. I hope I will continue to have the privilege and pleasure of working with him in the rest of my career.

I am also indebted to my master's thesis advisor, Dr. Juing Fang, for his guidance during my master's degree study at Shanghai Jiao Tong University, China, and for his continuing support and encouragement during my stay at Caltech. I should also express my sincere thanks to Dr. Mario Blaum of the IBM Almaden Research Center. He gave me many valuable comments when I was working on array codes for this thesis. It is really fun to have discussions with him.

I also thank my officemates at the Parallel and Distributed Computing Lab at Caltech for making the lab a real *Paradise*, an excellent environment to work and to live. In particular, many thanks go to Michael Gibson. He carefully proofread this thesis, which improved its presentation quality significantly.

I would also like to thank Profs. K. Mani Chandy, Robert J. McEliece, P. P. Vaidyanathan, and Henk C.A. van Tilborg for serving on my thesis defense committee.

This thesis work was supported in part by the NSF Young Investigator Award CCR-9457811, by the Sloan Research Fellowship, and by DARPA through an agreement with NASA/OSAT.

I should express my deep gratitude to my parents. They have always supported me with love, care and pride. I also thank my parents-in-law for their care, particularly for their sacrifice to allow their only daughter leaving them and accompanying me during my study at Caltech. Finally, my heartfelt gratitude goes to my dear wife, Lan. Her endless love and care are the driving forces of my study, research, and life. Without her sacrifice and patience, this work would not exist. I dedicate this thesis to her.

Abstract

As the need for data explodes with the passage of time and the increase of computing power, data storage becomes more and more important. Distributed storage, as distributed computing before it, is coming of age as a good solution to make systems highly available, i.e., highly scalable, reliable and efficient. The focus of this thesis is how to achieve data reliability and efficiency in distributed storage systems.

This thesis consists of two parts. The first part deals with the *reliability* of distributed storage systems. Reliability is achieved by computationally efficient MDS array codes that eliminate single points of failure in the systems, thus providing more reliability and flexibility to the systems. Such codes can be used as general MDS error-correcting codes. They are particularly suitable for use in distributed storage systems. The second part deals with the *efficiency* of distributed storage systems. Methods are proposed to improve the performance of data server and storage systems significantly through the proper use of data redundancy. These methods are based on error-correcting codes, particularly the MDS array codes developed in the first part.

Two new classes of MDS array codes are presented: the X-Code and the B-Code. The encoding operations of both codes are optimal, i.e., their update complexity achieves the theoretical lower bound. They distribute parity bits over all columns rather than concentrating them on some parity columns. As with other array codes, the error model for both codes is that errors or erasures are columns of the array, i.e., if at least one bit of a column is an error or erasure, then the whole column is considered to be an error or erasure. Both codes are of distance 3, i.e., they can either: correct two erasures, detect two errors or correct one error. In addition to encoding algorithms, efficient decoding algorithms are proposed, both for erasure-correcting and for error-correcting. In fact, the erasure-correcting algorithms are also optimal in terms of computation complexity.

The X-Code has a very simple geometrical structure: the parity bits are constructed along two groups of parallel *parity lines* of slopes 1 and -1. This is the origin of the name X-Code. This simple geometrical structure allows simple erasure-decoding and error-decoding algorithms, using only XORs and vector cyclic-shift operations.

The significance of the B-Code not only includes all its optimality properties: MDS, optimal encoding and optimal decoding, but also its relation with a 3-decade old graph theory problem. It is proven in this thesis that constructing a B-Code of *odd* length is exactly equivalent to constructing a *perfect one-factorization* (or P1F) of a complete graph. Constructing a P1F of an *arbitrary* complete graph has remained a conjecture since the early 1960's. Though the P1F conjecture remains unsolved, the B-Code as the first real application of the P1F problem will hopefully spur more research on it. It is also conjectured in this thesis that constructing a B-Code of *any* length, even or odd, is equivalent to constructing a P1F of a complete graph. An efficient error-correcting algorithm for the B-Code is also presented, which is based on the relations between the B-Code and its dual. The algorithm might give a hint of how to develop efficient decoding algorithms for other codes.

While it is intuitive that redundancy can bring reliability to a system, this thesis gives another direction: using redundancy actively to improve performance (efficiency) of distributed data systems. The results in this direction are both theoretical and experimental. System models are extracted from experiments in real practical systems; analytical results are derived using these and are then fed back to experiments for verification.

In this thesis, a novel *deterministic* voting scheme that uses error-correcting codes is proposed. The voting scheme generalizes all known simple deterministic voting algorithms. It can be tuned to various application environments with different error rates to drastically reduce average *communication complexity*, i.e., the amount of information that must be transmitted in order to get correct voting results.

Two problems are identified to improve the performance of general data server systems, namely the *data distribution* problem and the *data acquisition* problem. Solutions to these are proposed, as are general analytical results on performance of (n, k) systems. A simple service time model of a practical disk-based distributed server system is given. This model, which is based on experimental results, is a starting point for data distribution and data acquisition schemes. These results, both experimental and analytical, can be further used for more sophisticated scheduling schemes to optimize or improve the performance of data server systems that serve multiple clients simultaneously.

Finally, some research problems related to storage systems are proposed as future directions.

Contents

List of Figures

List of Tables

Chapter 1 Introduction

This is a time of information. This is a world of information. Information is generated, processed, transmitted and stored in various forms: text, voice, image, video and multimedia types. In this thesis, all these forms will be treated as general data. As the need for data increases exponentially with the passage of time and the increase of computing power, data storage becomes more and more important. From scientific computing to business transactions, data is the most precious part. How to store the data reliably and efficiently is the essential issue, that is the focus of this thesis.

As with distributed computing, distributed storage is coming of age as a good solution to achieve scalability, fault tolerance and efficiency. Historically, since the speed of storage devices, such as tapes and disks, is much slower than the speed of computing devices, e.g., CPUs, I/O is a bottleneck in computing systems. To improve the data throughput of storage devices, RAID (*Redundant Array of Independent Disks*) was proposed[22][25] to store data over multiple storage devices in a distributed way, so that the total I/O bandwidth is sum of the bandwidths of the individual storage devices. That was the start of distributed (networked) storage. Since then, storage technologies have been advancing rapidly; the capacity of magnetic devices continuously increases and access speed constantly improves. But as with CPUs, there are physical limits to the density of disks, seek time and rotational speed of the disk drives. These limits mean that the capacity and access speed of a single storage device can *not* be improved infinitely. The need for storage capacity and access speed can be met by improving storage systems at the architectural level, i.e., using multiple distributed storage devices connected via a fast network, such as the *Fiber Channel*, which reduces data latency incurred over the network to much less than the latency time of a single storage device. A distributed structure not only can increase the capacity and speed of storage systems, but also can bring fault tolerance and scalability.

As with computing, fault tolerance (or *reliability*) is increasingly important in storage systems. Some critical data should be available and some services should be provided even when faults occur in storage units. Besides, a storage system that allows some faulty units and can be replaced on-the-fly would have great value for business transactions, such as airport management, banking systems, internet portals and internet service provider systems. Naturally, reliability of storage systems can be achieved more easily using distributed structure. Scalability is another natural feature of distributed systems: addition or replacement of components is much more flexible in a distributed system than in a central system. Thus distributed storage systems can adapt better to dynamic and growing data demands.

In this thesis, the reliability, efficiency and scalability of distributed storage systems are all considered aspects of availability. A highly available storage system has high reliability (or can tolerate more faults), high efficiency (or performance) and scalability. Achieving high availability in distributed storage systems is the main topic of this thesis.

1.1 MDS Array Codes

Reliability of storage systems is often achieved by storing redundant data in the systems using *error-control codes*. Usually in storage systems, the failure of a single storage unit can be detected by the storage controllers and then can be masked. Thus *erasure-correcting* codes are often used, since the device failures can be marked as erasures. To make redundant data most effective, i.e., to tolerate as many single storage unit failures as possible, the codes should have the MDS (*Maximum Distance Separable*) property. Additionally, the codes should have simple encoding and decoding operations so that the computation overhead can be reduced to a minimum. The well-known Reed-Solomon codes[19] are a class of powerful MDS codes, but their encoding and decoding need rather complicated finite field operations. It is useful and important to design codes that have both the MDS property and simple encoding and decoding operations. MDS array codes are a class of error-correcting codes with the both properties.

Array codes have been studied extensively [4][5][6][7][12][15]. A common property of these codes is that the encoding and decoding procedures use only simple XOR (*exclusive* OR) operations, which can be implemented easily in either hardware or software or both; thus these codes are more efficient than Reed-Solomon codes in terms of computation complexity. Array codes are defined over an Abelian group $G(q)$ with the addition operation $+$. For simplicity, we will assume that $q = 2$, i.e., the code is binary and the addition is just a simple bitwise XOR. In an array code, the information and parity bits are placed in an array of size $n \times l$. In a distributed storage system, the bits in a same column are stored in a same disk. If a disk fails, then the corresponding column of the code is considered to be an erasure. Thus the code can also be viewed as an (l, k, d) code over $G(q^n)$, where k is the dimension of the code, defined as $k = log_{q^n} N$ with N being the number of its codewords; and d is the distance of the code, also defined over $G(q^n)$, i.e., the columns of the array. We will use this (l, k, d) notation in the following examples, where the $G(q^n)$ is omitted, when it is clear from contexts.

Current RAID systems can tolerate at most *one* disk failure at a time, i.e., the corresponding codes are just one-parity codes of distance 2. In more and more applications, fault-tolerance of only one single disk is not enough. A system that can tolerate more than one failure at the same time would be more robust and flexible. For example, when one disk fails, the system can still have some non-stop fault-tolerance capability while the bad disk is being replaced by a good one. This level of fault tolerance requires codes with distance more than 2. The recently designed EVENODD codes are a class of MDS array codes with distance 3 [4][5]. The following example shows a (7,5,3) EVENODD code, which can tolerate 2 simultaneous disk failures.

Example 1.1 *A* $(7, 5, 3)$ *EVENODD code*

Table 1.1 shows an encoding rule of a (7,5,3) EVENODD code, and Table 1.2 is a numerical example of Table 1.1.

a_1	a_2	a_3	a_4	a_5	$a_1 + a_2 + a_3 + a_4 + a_5$	$s + a_1 + b_5 + c_4 + d_3$
b_1	b_2	b_3	b_4	b_5	$b_1 + b_2 + b_3 + b_4 + b_5$	$s + a_2 + b_1 + c_5 + d_4$
c_1	c_2	c_3	c_4	c_5	$c_1 + c_2 + c_3 + c_4 + c_5$	$s + a_3 + b_2 + c_1 + d_5$
d_1	d_2	d_3	d_4	d_5	$d_1 + d_2 + d_3 + d_4 + d_5$	$s + a_4 + b_3 + c_2 + d_1$

Table 1.1: Encoding of a (7,5,3) EVENODD code, where $s = a_5 + b_4 + c_3 + d_2$

1	0	1	1	0	1	0
0	1	1	0	0	0	0
1	1	0	0	0	0	1
0	1	0	1	1	1	0

Table 1.2: Numerical example of a (7,5,3) EVENODD code

□

As shown in the above example, EVENODD code is a $(n, n-2, 3)$ MDS code. The information bits are placed in the first $n-2$ columns, and the parity bits are placed in the last 2 columns. Notice that the parity columns can be computed *independently*. One important parameter of array codes is the average number of parity bits affected by a change of a single information bit; this parameter is called the *update complexity* in this thesis. The update complexity is particularly crucial when the codes are used in storage applications that update information frequently. It also measures the encoding complexity of the code. The lower this parameter is, the simpler the encoding operations are. If a code is described by a *parity check matrix*, then this parameter is the average *row density* — the number of nonzero entries in a row — of the parity check matrix. Research has been done to reduce this parameter or to make the density of parity check matrix of codes as low as possible [13][28]. The update complexity of $EVENODD$ codes approaches 2 as the length (number of the columns) of the codes increases. But it was proven in [5] that for any linear array code with separate information and parity columns, the update complexity is always *strictly* larger than 2 (the obvious lower bound). Then a natural question is whether the update complexity of 2 is achievable for general array codes. The answer is, fortunately, yes. The following two examples show two classes of array codes whose update complexity achieves the lower bound 2. The codes are called the X-code and the B-Code, and will be discussed in detail later in Chapter 2 and Chapter 3.

Example 1.2 *A* $(7,5,3)$ *X-Code*

Table 1.3 shows an encoding rule of a (7,5,3) X-Code, and Table 1.4 is a numerical example of Table 1.3.

a_1	a_2	a_3	a_4	a_5	a_6	a_7
b_1	b_2	b_3	b_4	b_5	b_6	b_7
c_1	c_2	c_3	c_4	c_5	c_6	c_7
d_1	d_2	d_3	d_4	d_5	d_6	d_7
e_1	e_2	e_3	e_4	e_5	e_6	e_7
$a_3+b_4+c_5$ $+d_6+e_7$	$a_4+b_5+c_6$ $+d_7+e_1$	$a_5+b_6+c_7$ $+d_1+e_2$	$a_6+b_7+c_1$ $+d_2+e_3$	$a_7+b_1+c_2$ $+d_3+e_4$	$a_1+b_2+c_3$ $+d_4+e_5$	$a_2+b_3+c_4$ $+d_5+e_6$
$a_6+b_5+c_4$ $+d_3+e_2$	$a_7+b_6+c_5$ $+d_4+e_3$	$a_1+b_7+c_6$ $+d_5+e_4$	$a_2+b_1+c_7$ $+d_6+e_5$	$a_3+b_2+c_1$ $+d_7+e_6$	$a_4+b_3+c_2$ $+d_1+e_7$	$a_5+b_4+c_3$ $+d_2+e_1$

Table 1.3: Encoding of a (7,5,3) X-Code

1	0	1	1	0	1	0
0	1	1	0	0	0	0
1	1	0	0	0	0	1
0	1	0	1	1	1	0
1	0	0	1	0	1	0
0	0	1	1	0	1	1
1	1	1	0	0	1	0

Table 1.4: Numerical example of a (7,5,3) X-Code

□

As shown in the example, the X-Code is an MDS array code of size $n \times n$, an $(n, n-2, 3)$ code.

Example 1.3 *A* $(7, 5, 3)$ *B-Code*

Table 1.5 shows an encoding rule of a (7,5,3) B-Code, and Table 1.6 is a numerical example of Table 1.5.

a_1	a_2	a_3	a_4	a_5	a_6	a_7
b_1	b_2	b_3	b_4	b_5	b_6	b_7
$a_5 + a_6 + a_7$ $+b_2 + b_4$	$a_6 + a_1 + b_7$ $+b_3 + b_5$	$a_1 + a_2 + c_7$ $+b_4 + b_6$	$a_2 + a_3 + a_7$ $+b_5 + b_1$	$a_3 + a_4 + b_7$ $+b_6 + b_2$	$a_4 + a_5 + c_7$ $+b_1 + b_3$	c_7

Table 1.5: Encoding of a (7,5,3) B-Code

1	0	1	1	0	1	0
0	1	1	0	0	0	0
0	1	0	1	1	1	1

Table 1.6: Numerical example of a (7,5,3) B-Code

□

The B-Code is an array code of size $n \times (2n + 1)$, an MDS $(l, l - 2, 3)$ code. A common structure of the X-Code and the B-Code is that parity bits are no longer placed in separate columns but mixed in with information bits. This is the key to achieving the lower bound of the update complexity. While the construction of the X-Code is fairly easy using a geometrical representation — the two parity rows are constructed using two groups of diagonals of slope 1 and -1 — the construction of the B-Code is not so obvious. Indeed the construction of the B-Code given here comes from a 3-decade old graph theory problem, the *perfect one-factorizations* of complete graphs [29]. Also, as shown in the two examples above, the column size of a B-Code is only half of that of an X-Code with the same length. In fact, the B-Code achieves the *maximum* length possible for MDS codes with the optimal update, thus the B-Code has *optimal length*, twice that of the X-Code with the same column size. In addition, the parity bits are evenly distributed over all columns, and each parity bit requires the same number of *XOR* operations. Consequently, the computational complexity for computing parity bits is *balanced*, i.e., the X-Code and the B-Code feature *balanced computation* as well. This property is quite useful in distributed storage systems, since the computational loads are naturally distributed to all disks evenly, eliminating another bottleneck. The properties of the X-Code and the B-Code are summarized in Table 1.7, together with a comparison with *Reed-Solomon* and *EVENODD* codes.

Codes \ Properties	MDS	XOR	Optimal Update	Optimal Length	Balanced Computation
Reed Solomon	Yes	No	No	Yes	No
EVENODD	Yes	Yes	No	No	No
X-Code	Yes	Yes	Yes	No	Yes
B-Code	Yes	Yes	Yes	Yes	Yes

Table 1.7: X-Code, B-Code vs. Reed-Solomon and EVENODD.

1.2 Efficiency through Redundancy

While it is conventional wisdom that redundancy is necessary for fault tolerance, redundancy is in general regarded as a passive cost (overhead) to achieve reliability. However, in this thesis, it will be shown that in a distributed storage system, redundancy is an active part of the system in the sense that proper data redundancy can help to improve the performance (data throughput) of storage systems. Thus data redundancy will improve not only the reliability of a system, but also the efficiency of a system. A similar idea was first shown in [11], namely that redundant data can make packet routing more efficient by reducing the mean and variance of the routing delay. Recently, more scalable and efficient reliable multicast schemes have been proposed, based on data redundancy in the messages to be multicast[14]. We will show here a more systematic way of using proper redundancy, based on error-correcting codes (particularly MDS array codes), to improve the performance of storage systems.

As an example, the efficiency of majority voting can be improved by introducing redundancy to the data to be voted on. Distributed majority voting is a useful tool to maintain data consistency in storage systems, including memory-based fast storage systems [17], in addition to creation of fault-tolerant computing systems [38]. One important issue in voting is to reduce the *communication complexity*, the total number of bits that are transmitted via communication medium, thus to improve the efficiency of voting. A typical solution is to vote on a *signature* or *hash function* of the data instead of on the data itself, since the signature is a compressed form of the data that is much shorter than the data itself. But this gives a probabilistic voting result, because there is a nonzero probability of different data having the same signature. For many applications [38], *deterministic* voting schemes are needed to provide accurate voting results. Now instead of naively sending the whole data, error-correcting codes (particularly MDS array codes) can be used to significantly reduce the amount of data that needs to be sent in order to get a deterministic voting result. The following example shows the role that codes can play.

Example 1.4 *Voting using codes*

A fault-tolerant system consists of 7 nodes, each of which generates 15 bits of data. Before being written to memory (or other storage unit), the data needs to be voted on. Suppose each node generates 10,01,11,10,00,10,001. Instead of sending all these 15 bits, each node encodes these 15 bits into 21 bits, using the (7,5,3) B-Code, as in Table 1.6: 100,011,110,101,001,101,001. Then the ith ($i = 1, \cdots, 7$) node just broadcasts the 3 bits in the ith column; after receiving the total 21-bits of coded data, each node can decode the data to retrieve the 15-bits information data, then compare these decoded 15 bits of information data to its own local 15-bits of data. The result of comparison can then be sent to all other nodes using a 1-bit flag. In this example, every node broadcasts a positive flag and agrees upon the decoded data, which is the correct voting result.

Now if the 7th node's local data changes to 10,01,11,10,00,10,000 while all the other nodes' data does not change, the 7th node broadcasts its coded 3-bit part as 000 instead of 001, so each node now has the 21 coded bits as 100,011,110,101,001,101,000.

But the (7,5,3) B-Code can correct 1 column error, so the decoded 15-bits of data is still 10,01,11,10,00,10,001. All the 6 nodes except the 7th node agree with this result. Now, the voting result is still a correct one: 10,01,11,10,00,10,001.

So the correct deterministic voting result can be achieved with each node sending only 4 bits instead of 15 bits, and the communication complexity is reduced drastically. □

The above example is only a special case where data redundancy helps improve efficiency. This is even true for more general distributed storage or *data server* systems. A distributed data server system consists of multiple server nodes that are connected by reliable communication networks. Each server has its own local storage unit. The whole system provides data to clients. So a data server system can be regarded as a superset of a storage system. For a data server system, since the I/O speed of the local disks is much slower than the CPU speed of the servers, the whole performance of the system is dominated by the bottleneck of the disks. Distributing data over multiple servers can help to overcome this bottleneck and improve the data throughput of the whole system, since the data can be accessed in parallel from multiple servers at the same time.

The data a client needs is distributed over all the servers in such a way that the client can reconstruct the complete requested data after it gets data from at least k of the all n servers in the system. (Here we assume that the client can receive data from multiple servers at the same time, by buffering incoming data at its communication devices.) Such a data server system is called an (n, k) data system. The performance of a data server system can be further improved by implementing it with an (n, k) rather than a naive (n, n) system, which, of course, is only a special case of a general (n, k) system. Again, implementation of (n, k) systems can be achieved using (n, k) error-correcting codes, especially (n, k) MDS array codes. Let's demonstrate a few examples of such implementations.

Example 1.5 *Performance improvement using an (n, k) system*

Suppose $n = 6$ and the total amount of the data that a client requests is 12 Kbytes. Then in a $(6, k)$ system, the data stored on each server is $\frac{12}{k}$ Kbytes using MDS array codes, where k can range from 1 to 6. The delivery time model is as follows: the time required for each server to deliver m Kbytes of data to the client is $bm + 0.8$ msecs, where for simplicity, b is a random variable uniformly distributed over $[0.3, 0.5]$ msecs/Kbyte. The following examples show the performance for different k.

For $k = 6$, each server stores 2 Kbytes of data, and there is *no* redundant data in the whole system. Suppose the delivery time of the 2 Kbytes of data from each server is $1.50, 1.45, 1.65, 1.75, 1.55$ and 1.70 msecs respectively, then the client has to wait for 1.75 msecs until it can obtain the whole 12 Kbytes of data it requested.

Now let $k = 5$, the 12 Kbytes of data is evenly distributed over 5 servers, and the remaining server stores the parity of the data stored on the first 5 servers. In this case, each server has 2.4 Kbytes of data, and the total amount of redundant data in the system is 2.4 Kbytes. Now suppose the delivery time of the 2.4 Kbytes of data from each server is $1.55, 1.65, 1.60, 1.72, 1.70$ and 1.90 msecs respectively. Since now it can retrieve the 12 Kbytes of data from any 5 servers, the client only needs to wait for 1.72 msecs, which is shorter than the $k = 6$ case. □

This shows that data redundancy can improve the system's performance, as is already known to the database community [26]. But the following has not been studied: can more redundancy provide a greater performance improvement when the total data resource (number of the servers) of a system is fixed?

Example 1.6 *Proper redundancy in an (n, k) system*

Now let $k = 4$. This (6,4) system can use a (6,4,3) B-Code, which can be obtained simply by setting all the bits in last column of Table 1.5 to zero, changing all the other columns accordingly and deleting the last column, as shown in Table 1.8. Each server stores 3 Kbytes

a_1	a_2	a_3	a_4	a_5	a_6
b_1	b_2	b_3	b_4	b_5	b_6
$a_5 + a_6$ $+b_2 + b_4$	$a_6 + a_1$ $+b_3 + b_5$	$a_1 + a_2$ $+b_4 + b_6$	$a_2 + a_3$ $+b_5 + b_1$	$a_3 + a_4$ $+b_6 + b_2$	$a_4 + a_5$ $+b_1 + b_3$

Table 1.8: Encoding of a (6,4,3) B-Code

of data, and the total amount of redundant data in the system is now 6 Kbytes. Suppose now the delivery time of the 3 Kbytes data from each server is $1.72, 1.78, 2.20, 1.82, 2.10$ and 1.85 msecs respectively. The total time the client needs to wait is 1.85 msecs, which is worse than in the previous example. □

So in the above examples, with the above data delivery time model, a $(6, 5)$ system may give the best performance. What is the *proper* redundancy when the total number of the servers is given? Or when n is given, what is the best k so as to achieve the best system performance? This is the so-called *data distribution* problem at the server side, which will be investigated in this thesis.

Once a data distribution scheme is decided at the server side, the client should choose a way of reading data from the servers that optimizes the performance of the client-server system. This is called *data acquisition* at the client side. This problem cannot be found in current literatures either. The following example gives a flavor of this problem.

Example 1.7 *Data acquisition schemes of a* $(6, 4)$ *system*

A client requests 12 Kbytes of data from a $(6, 4)$ system. Again, use the (6,4,3) B-Code in Table 1.8 and the same delivery time model in the above two examples. Now each server stores 3 Kbytes of data. The client has two options to read the 12 Kbytes of data from the servers: 1) request 2 Kbytes of data from each of 6 servers, i.e., the 12 original data symbols in the top 2 rows in Table 1.8. In this case the client needs to gather data from all 6 servers; 2) request all 3 Kbytes of data from all of the servers, then the client only needs to wait for data from any 4 servers. Suppose the data delivery times of 2 Kbytes and 3 Kbytes of data from each server are the same as in the above two examples. The client needs 1.75 msecs if it chooses option 1 and 1.85 msecs if it chooses option 2. So in this case, option 1 gives better performance. □

All the examples in this section suggest that proper data redundancy based on error-correcting codes should be actively introduced to storage systems to improve the performance.

1.3 Main Contributions of the Thesis

This thesis consists of two parts. The first part deals with the design of MDS array codes that are computationally efficient. Such codes can be used as general MDS error-correcting codes, and are particularly suitable for distributed storage systems. The second part shows that the performance of data server and storage systems can be improved significantly by the proper use of redundant data based on error-correcting codes, particularly the MDS array codes developed in the first part.

Two new classes of MDS array codes are presented, called the X-Code and the B-Code.

The encoding operations of both codes are optimal, i.e., their update complexity achieves the theoretical lower bound. The key to achieving this lower bound is by distributing parity bits over all the columns rather than concentrating them on some parity columns. This idea which was discovered in late 1970's and largely ignored since then [37], is stated much more clearly and directly in this thesis. As with other array codes, the error model for both codes is this: if a column has at least one bit erasure (error), then this column is considered as an erasure (error) column. (Frequently, the word column will be dropped, and they will be simply called erasures or errors.) Both codes are of distance 3, i.e., they can either: correct two erasures, detect two errors or correct one error. In addition to encoding algorithms, efficient decoding algorithms are proposed, both for erasure-correcting and for error-correcting. In fact, the erasure-correcting algorithms are also optimal in terms of computation complexity.

The X-Code has a very simple geometrical structure: the parity bits are constructed along two groups of parallel *parity lines* of slopes 1 and -1. This is the origin of the name X-Code. This simple geometrical structure allows simple erasure-decoding and error-decoding algorithms, using only XORs and vector cyclic-shift operations.

The significance of the B-Code not only includes all its optimality properties: MDS, optimal encoding and optimal decoding, but also its relation with a 3-decade old graph theory problem. It is proven in this thesis that constructing a B-Code of *odd* length is exactly equivalent to constructing a *perfect one-factorization* (or P1F) of a complete graph. Constructing a P1F of an *arbitrary* complete graph has remained a conjecture since the early 1960's. Though the P1F conjecture remains unsolved, the B-Code as the first real application of the P1F problem will hopefully spur more research on it. It is also conjectured in this thesis that constructing a B-Code of *any* length, even or odd, is equivalent to constructing a P1F of a complete graph. An efficient error-correcting algorithm for the B-Code is also presented, which is based on the relations between the B-Code and its dual. The algorithm might give a hint of how to develop efficient decoding algorithms for other codes.

While it is intuitive that redundancy can bring reliability to a system, this thesis gives another direction: using redundancy actively to improve performance (efficiency) of distributed data systems. The results in this direction are both theoretical and experimental. System models are extracted from experiments in real practical systems; analytical results are derived using these and are then fed back to experiments for verification.

In this thesis, a novel *deterministic* voting scheme that uses error-correcting codes is proposed. The voting scheme generalizes all known simple deterministic voting algorithms. The main contributions related to the voting scheme include: (i) using the correcting capability in addition to the detecting capability of codes (only the detection was used in known schemes) to drastically reduce the chances of retransmission of the whole local result of each node, thus reducing the communication complexity of the voting, (ii) a proof that the scheme correctly reaches the same voting result as the naive voting algorithm in which every module broadcasts its local result to all other modules, and (iii) a method of tuning the scheme for optimal average case communication complexity by choosing the parameters of the error-correcting code, thus making the voting scheme adaptive to various application environments with different error rates.

Two problems are identified to improve the performance of general data server systems, namely the *data distribution* problem and the *data acquisition* problem. Solutions to these are proposed, as are general analytical results on performance of (n, k) systems. A simple service time model of a practical disk-based distributed server system is given. This model, which is based on experimental results, is a starting point for data distribution and data acquisition schemes. These results, both experimental and analytical, can be further used for more sophisticated scheduling schemes to optimize or improve the performance of data server systems that serve multiple clients simultaneously.

Most of the results in this thesis have been published or submitted to journals and conferences. The results related to the X-Code are in [33]. The B-Code and related issues are discussed in detail in [34]. The new voting scheme is presented in [35], and efficiency issues in data server systems are investigated in [36].

1.4 Organization

The rest of the thesis is organized as follows: Chapter 2 gives results about the X-Code, and the B-Code is discussed in Chapter 3. A generalized deterministic voting scheme using codes is presented in Chapter 4. Various issues of improving performance of data server systems using codes are addressed in Chapter 5. Chapter 6 concludes the thesis and gives some future research directions.

Chapter 2 X-Code: MDS Array Codes with Optimal Encoding

2.1 Introduction

As stated in Chapter 1, array codes have important applications in storage systems[6] [12] and have been studied extensively[4][5][7][8][15]. A common property of these codes is that the encoding and decoding procedures use only simple XOR and cyclic shift operations, and thus are more efficient than Reed-Solomon codes in terms of computation complexity [6]. In this chapter, we describe the *X-Code*, a new class of array codes of size $n \times n$ over any Abelian group $G(q)$ with an addition operation $+$, where q is the size of the group. When $q = 2^m$, the addition operation is just the usual bit-wise XOR operation. Similar to the codes in [4][7], the error model of the X-Code is that errors or erasures are columns of the array, i.e., if one symbol of a column is an error or erasure, then the whole column is considered to be an error or erasure. The same model is also used for the B-Code in the next chapter. As usual, the dimension of the code is defined to be $k = log_{q^n} N$, where N is the number of its codewords. The code can also be viewed as an (n, k, d) code over $G(q^n)$. Its distance is defined over $G(q^n)$, i.e., over the columns of the array. The X-Code is an *MDS* (*Maximum Distance Separable*) code of distance $d = 3$, i.e., $k = n - 2$, which meets the Singleton bound[19]: $d = n - k + 1$.

Although it was shown[37][8] that for general array codes of distance 3, the lower bound 2 of update complexity is achievable, the code in [37] and later its clearer form [8] are described by *parity check matrix* and not directly as array codes. The new family of array codes, called the X-Codes, has a much simpler and direct *geometrical structure* and has an update complexity of *exactly* 2.

Both the X-Codes and the codes in [37] and [8] combine information and parity symbols within columns in order to achieve optimal update complexity. The redundancy of the X-Code is obtained by adding two parity *rows* rather than two parity columns, which results in a nice property that updating one information symbol affects only *two* parity symbols, i.e., the *update complexity* is always *two*. In addition, the number of operations for computing parity symbols is the same for every column, namely, the computational load is evenly distributed among all the columns, and thus the bottleneck effects of repeated *write* operations are naturally overcome.

In summary, the main contribution of this chapter is a construction for the X-Code, a new class of MDS array codes of distance 3, with the properties of optimal update complexity and balanced computations. The simple geometrical structure of the the X-Code makes its decoding very efficient, both for *two erasures* and for *one error*.

This chapter is organized as follows. In Section 2.2, the encoding scheme of the X-Code is described, and a proof of its *MDS* property is presented. In Section 2.3, an efficient decoding algorithm for correcting two erasures and an efficient algorithm for correcting one error are provided. Section 2.4 concludes the chapter and presents some future research directions.

2.2 X-Code Description

In the X-Code, information symbols are placed in an array of size $(n-2) \times n$. Like other array codes [4][5][7][15], parity symbols are constructed by adding (with the group addition operation $+$) the information symbols along several *parity check lines* or *diagonals* of some given *slopes*. But instead of being put in separate columns, the parity symbols of the X-Code are placed in *two* additional *rows*. So the coded array is of size $n \times n$, with the first $n-2$ rows containing information symbols, and the last two rows containing parity symbols. Notice that each column has information symbols as well as parity symbols, i.e., information symbols and parity symbols are mixed in each column. By the structure of the code, if two columns are erased, the number of remaining symbols is $n(n-2)$, which is equal to the number of original information symbols, making it possible to recover the two column *erasures*, i.e., missing columns.

2.2.1 Encoding Procedure

Let $C_{i,j}$ be the symbol at the ith row and jth column. The parity symbols of the X-Code are constructed according to the following encoding rules:

$$C_{n-2,i} = \sum_{k=0}^{n-3} C_{k,\langle i+k+2\rangle_n}$$

$$C_{n-1,i} = \sum_{k=0}^{n-3} C_{k,\langle i-k-2\rangle_n} \tag{2.1}$$

where $i = 0, 1, \cdots, n-1$, and $\langle x\rangle_n = x \bmod n$. Geometrically speaking, the two parity rows are just the checksums along diagonals of slopes 1 and -1 respectively. The following example gives a construction of the X-Code of size 5×5.

Example 2.1 *X-Code of size 5×5*

The first parity row is calculated along the diagonals of slope 1, with the last row being an imaginary 0-row, as follows:

△	♣	♢	♡	♠		1	0	0	1	1
♠	△	♣	♢	♡		0	1	0	1	1
♡	♠	△	♣	♢		0	0	1	0	1
♢	♡	♠	△	♣	←1st parity check row→	0	0	1	1	0
♣	♢	♡	♠	△	←imaginary 0-row →	0	0	0	0	0

The second parity row is calculated along the diagonals of slope -1, as follows:

△	♣	♢	♡	♠		1	0	0	1	1
♣	♢	♡	♠	△		0	1	0	1	1
♢	♡	♠	△	♣		0	0	1	0	1
♡	♠	△	♣	♢	←2nd parity check row→	1	1	0	1	1
♠	△	♣	♢	♡	←imaginary 0-row→	0	0	0	0	0

Then the complete codeword is

1	0	0	1	1
0	1	0	1	1
0	0	1	0	1
0	0	1	1	0
1	1	0	1	1

□

From the construction of the X-Code, it is easy to see that the two parity rows are obtained independently; more specifically, each information symbol affects exactly *one* parity symbol in each parity row. All parity symbols depend only on information symbols, *not* on each other. So, updating one information symbol results in updating only *two* parity symbols. Thus the X-Code has the optimal encoding (or update) property, i.e., its update complexity of 2 matches the lower bound for any code of distance 3.

It is also easy to see that the X-Code is a cyclic code in terms of columns, i.e., cyclically shifting columns of a codeword of the X-Code results in another codeword of the X-Code.

In addition, notice that each column has two parity symbols, each of which is the checksum of $n-2$ information symbols. Thus computing parity symbols at each column needs $2(n-3)$ group additions. This balanced computation property of the X-Code is very useful in applications that require evenly distributed computations.

2.2.2 The MDS Property

In this section, we state and prove the *MDS property* of the X-Code.

Theorem 2.1 (MDS Property)

The X-Code has column distance of 3, i.e., it is MDS, if and only if *n is a prime number.*

Proof: Let us start with the *sufficient* condition, namely, prove that for any prime number n, the X-Code is *MDS*.

First observe that the X-Code is a linear code, thus proving that the code has distance of 3 *is equivalent to* proving that the code has *minimum column weight* w_{min} of 3, i.e., a valid codeword of the X-Code has at least 3 nonzero columns. (A column is called a nonzero column if at least one symbol in the column is nonzero.) We will prove this by contradiction.

From the construction of the X-Code, checksums are obtained along diagonals of slope 1 or slope -1, so it is impossible to have only *one* nonzero column, thus $w_{min} > 1$.

Now suppose $w_{min} = 2$. Without loss of generality, we can assume the nonzero columns are the 0th and kth columns where $1 \leq k \leq n-1$, because of the column cyclic property of the X-Code. Denote the ith symbol of the 0th and kth columns a_i and b_i respectively.

Observe that any one diagonal of slope 1 or -1 only traverses $n-1$ columns, then among the diagonals of slope 1, the diagonal crossing a_{n-1-k} does *not* cross any symbol of the kth column, and the diagonal crossing b_{k-1} does *not* cross any symbol of the 0th column, so $a_{n-1-k} = 0$ and $b_{k-1} = 0$. Because the diagonals of slope -1 have the same property, we can also get $a_{k-1} = 0$ and $b_{n-1-k} = 0$ (or $b_{n-1} = 0$ if $k = 1$).

Starting from $a_{k-1} = 0$, we get $b_{2k-1} = 0$, since they are in the same diagonal of slope 1; then we get $a_{3k-1} = 0$, since it is on the same diagonal of slope 1 with $b_{2k-1}, \cdots$, and so on, we have

$$a_{k-1} = a_{3k-1} = a_{5k-1} = \cdots = a_{(n-2)k-1} = 0$$

and

$$b_{2k-1} = b_{4k-1} = b_{6k-1} = \cdots = b_{(n-1)k-1} = 0$$

where all indices are *mod* n.

Similarly, starting from $a_{n-1-k} = 0$, we have

$$a_{n-1-k} = a_{n-1-3k} = \cdots = a_{n-1-(n-2)k} = 0$$

and

$$b_{n-1-2k} = b_{n-1-4k} = \cdots = b_{n-1-(n-1)k} = 0$$

again, all indices above are *mod* n.

We can describe the above 4 sets of entries in the array as follows. Let $A_0 = \{\langle(2m+1)k-1\rangle_n : m = 0, 1, \cdots, \frac{n-3}{2}\}$, and $A_1 = \{\langle n-(2l+1)k-1\rangle_n : l = 0, 1, \cdots, \frac{n-3}{2}\}$, let $B_0 = \{\langle 2mk-1\rangle_n : m = 1, 2, \cdots, \frac{n-1}{2}\}$, and $B_1 = \{\langle n-2lk-1\rangle_n : l = 1, 2, \cdots, \frac{n-1}{2}\}$, notice that none of the sets includes $n-1$, since n is prime. This can also be seen from the construction of the X-Code, since the $(n-1)$th row is just an imaginary all-0 row and it does not need to be considered. An illustration of the above sets for n = 5 and k = 2 is as follows:

A_0		B_1		
A_0		B_1		
A_1		B_0		
A_1		B_0		

Since n is prime, for any $1 \le k \le n-1$, $\gcd(n,k) = 1$, $\|A_0\| = \|A_1\| = \frac{n-1}{2}$, and if there were m and l such that

$$(2m+1)k - 1 \equiv n - (2l+1)k - 1 \bmod n \tag{2.2}$$

then,

$$2(m+l+1)k \equiv 0 \bmod n \tag{2.3}$$

but $1 \le m+l+1 \le n-2, \gcd(m+l+1,n)=1, \gcd(2k,n)=1$, so it is impossible to have such a pair of m and l, i.e., $\|A_0 \cap A_1\| = 0$. Notice that $n-1 \equiv (2\frac{n-1}{2}+1)k-1 \bmod n$, so

$$A_0 \cup A_1 = \{0,1,\cdots,n-2\}$$

Similarly,

$$B_0 \cup B_1 = \{0,1,\cdots,n-2\}$$

So all the first $n-1$ symbols in the 0th and the kth columns are 0's, obviously the last symbols in the 0th and the kth columns should be also 0's. This is a contradiction. Thus, $w_{min} \ge 3$, but it is easy to see there is a codeword of column weight 3, so $w_{min} = 3$. This concludes the proof for the sufficient condition.

On the other hand, from the equation Eq. (2.3), if n were not a prime number, then it could be factored into two factors n_1 and n_2. Thus we got a solution (k,l,m) for the equation Eq. (2.3) or Eq. (2.2), where $k=n_1$, and $m+l+1=n_2$, and $2 \le k \le n-1$. This means there is a codeword of weight 2, or equivalently the distance of the code is no greater than 2. This contradicts the fact that the code is of distance 3. So n being a prime number is also a necessary condition to the *MDS* property of the X-Code. □

Remarks:

1. For the sufficient condition, we can always find a diagonal of one slope which traverses only one of the two columns. Thus the traversed symbol must be 0. Starting from this 0-symbol and using the diagonal of the other slope crossing this symbol, we can determine that the symbol crossed by the diagonal in the other column must be also 0. So this saw-like recursive procedure can proceed until it hits a parity symbol at one of the two columns, since a parity symbol can only lie in one diagonal. We call this saw-like recursion a *decoding chain*. Since there are four parity symbols in the two columns, there are at most four decoding chains. (A simple calculation can show that there are two decoding chains when $k=1$ and four decoding chains otherwise.) If n is prime, the procedure of getting the decoding chains will stop with all the symbols

in the two columns being 0s. Since this procedure is deterministic once the positions of the two columns are given, it also provides an efficient erasure decoding algorithm.

2. In the code construction above, we use diagonals of slope 1 and -1. This choice of slopes is not unique. In fact, for $s = 1, \cdots, \frac{n-1}{2}$, codes constructed by the pair of slopes $(s, -s)$ are *MDS* if and only if n is prime. The proof is similar to the case where the slope pair is (1,$-$1). It seems that other slope pairs do not provide advantages over (1,$-$1), so in this paper we will focus on the X-Codes generated by the slope (1,$-$1).

2.3 Efficient Decoding Algorithms

In this section, we present decoding algorithms for correcting two erasures or one error of the X-Code. Neither the encoding algorithm of the code or decoding algorithms require any finite field operations. Instead, the only operations needed are additions and cyclic shifts, both of which can be implemented very efficiently in software and/or hardware. It is clear how to correct one erasure, since the erasure can be easily recovered along one of the diagonals. So we will proceed with correcting two erasures.

2.3.1 Correcting Two Erasures

First notice that in an array of size $n \times n$, if two columns are erasures, then the key unknown symbols of the two erased columns are the information symbols. So the number of unknown symbols is $2(n-2)$. On the other hand, in the remaining array, there are $2(n-2)$ parity symbols that include all the $2(n-2)$ unknown symbols. Hence, correcting the two erasures is only a problem of solving for $2(n-2)$ unknowns from $2(n-2)$ linear equations. Since the X-Code is of distance 3, it can correct two erasures; thus the $2(n-2)$ linear equations must be linearly independent, i.e., the linear equations are solvable. Now notice that a parity symbol can *not* be affected by more than one information symbol in the same column, so each equation has at most two unknown symbols, with some having only one unknown symbol. This drastically reduces the complexity of solving the equations.

Suppose the erasure columns are the ith and jth $(0 \leq i < j \leq n-1)$ columns. Since each diagonal traverses only $n-1$ columns, if a diagonal crosses a column at the last row, no symbols of that column are included in this diagonal. This determines the position of the parity symbol that includes only one symbol from the two erasure columns, thus this symbol can be immediately recovered from a simple checksum along this diagonal. From this symbol, we can get a decoding chain as discussed in Remark 1 in Section 2.2. Using this decoding chain and the other one (if $j-i=1$) or three (if $j-i>1$), all unknown symbols can be recovered.

Now let us calculate the starting parity symbols of the decoding chains. First consider the diagonals of slope 1. Suppose the xth symbol of the ith column is the only unknown symbol in a diagonal. This diagonal hits the jth column at the $(n-1)$th row, and hits the first parity row at the yth column, i.e., the three points $(x,i),(n-1,j)$ and $(n-2,y)$ are on the same diagonal of slope 1, thus the following equations hold:

$$\begin{cases} (n-1)-x \equiv j-i \bmod n \\ (n-1)-(n-2) \equiv j-y \bmod n \end{cases}$$

Since $1 \le j-i \le n-1$, and $0 \le j-1 \le n-2$, the solutions for x and y are

$$\begin{cases} x = \langle (n-1)-(j-i) \rangle_n = (n-1)-(j-i) \\ y = \langle j-1 \rangle_n = j-1 \end{cases}$$

So from the parity symbol $C_{n-2,j-1}$, we can immediately get the symbol $C_{(n-1)-(j-i),i}$ in the ith column. Similarly, the symbol $C_{(j-i)-1,j}$ in the jth column can be solved directly from the parity symbol $C_{n-2,\langle i-1\rangle_n}$.

Symmetrically with the diagonals of slope -1, the symbol $C_{(j-i)-1,i}$ in the ith column can be solved from the parity symbol $C_{n-1,\langle j+1\rangle_n}$, and the symbol $C_{(n-1)-(j-i),j}$ in the jth column can be solved from the parity symbol $C_{n-1,i+1}$.

A formal algorithm for correcting the two erasures ith and jth $(0 \le i < j \le n-1)$ columns of the X-Code can be described as follows:

Algorithm 2.1 (Correcting Two Erasures)

Use each of the four parity symbols $C_{n-2,j-1}$, $C_{n-2,\langle i-1\rangle_n}$, $C_{n-1,\langle j+1\rangle_n}$ and $C_{(n-1)-(j-i),j}$ as the starting point of a decoding chain; in each decoding chain use the saw-like recursion method to recover unknown symbols until a parity symbol at one of the two erasure columns is hit, then start a new decoding chain, as discussed in Section 2.2.

A pseudo-code description of the algorithm is as follows:

1. Init_Slope_Set = { 1, 1, -1, -1 }

$Init\_Par\_Col\_Set = \{j-1, \langle i-1\rangle_n, \langle j+1\rangle_n, i+1\}$;

$Init\_Sym\_Col\_Set = \{i, j, i, j\}$;

$Init\_Sym\_Row\_Set = \{(n-1)-(j-i), (j-i)-1, (j-i)-1, (n-1)-(j-i)\}$;

$i = -1$;

2. $i++$;

If $i == 4$ **Then**

Compute $P_0[i], P_0[j], P_1[i], P_1[j]$ according to the encoding rule Eq. (2.1);

Stop;

Else

$Slope = Init\_Slope\_Set[i]$;

$Par\_Col = Init\_Par\_Col\_Set[i]$;

$Sym\_Col = Init\_Sym\_Col\_Set[i]$;

$Sym\_Row = Init\_Sym\_Row\_Set[i]$;

End If

3. **If** $Par\_Col == i$ **Or** $Par\_Col == j$ **Then**

Goto 2;

Else

If $Slope == 1$ **Then**

$C_{Sym\_Row, Sym\_Col} = P_0[Par\_Col] + \sum_{k=0, k \neq Sym\_Row}^{n-3} C_{k, \langle Par\_Col+k+2\rangle_n}$;

Else

$C_{Sym\_Row, Sym\_Col} = P_1[Par\_Col] + \sum_{k=0, k \neq Sym\_Row}^{n-3} C_{k, \langle Par\_Col-k-2\rangle_n}$;

End If

End If

4. $Slope = -Slope$;

$Par\_Col = \langle Sym\_Col - Slope * (Sym\_Row + 2)\rangle_n$;

If $Sym\_Col == i$ **Then**

$Sym\_Col = j$;

Else

$Sym\_Col = i$;

End If

$Sym\_Row = \langle n - 2 - Slope * (Par\_Col - Sym\_Col)\rangle_n$;

Goto 3;

□

Step 1 of the algorithm computes those positions of the four parity symbols that contain only one unknown symbol. Steps 2 through 4 include the saw-like recursive procedure described above. Step 3 is just the checksum calculation along a diagonal of slope *Slope* crossing the parity symbol $P_0[Par\_Col]$ (or $P_1[Par\_Col]$). This recovers the unknown symbol C_{Sym_Row,Sym_Col} if the parity symbol is not in one of the erasure columns; otherwise, it just restarts with another parity symbol obtained in Step 1. Step 4 uses the symbol that was just found to calculate the position of the next unknown symbol.

The correctness of the algorithm can be deduced from the proof of Theorem 1 and Remark 1 in Section 2.2. The complexity of the algorithm is easy to analyze. Each iteration solves one unknown symbol and requires $(n-3)$ additions. So to correct two erasure columns, the decoding algorithm needs $2n(n-3)$ additions, just the same as that of the encoding algorithm.

The following is a simple example to show how the decoding algorithm works. To be more general, we use symbols rather than numerical values.

Example 2.2 *Correcting Two Erasures of a* 5×5 *X-Code*

Without loss of generality, we assume the last (i.e., the 4th) column is one of the erasures, and because of the symmetry of the code, we only need to examine the cases where the other erasure is the 3rd or the 2nd column.

Case 1. $i = 3,\ j = 4$

Then the remaining array is:

a_0	a_1	a_2	$?(a_3)$	$?(a_4)$
b_0	b_1	b_2	$?(b_3)$	$?(b_4)$
c_0	c_1	c_2	$?(c_3)$	$?(c_4)$
$d_0 = a_2 + b_3 + c_4$	$d_1 = a_3 + b_4 + c_0$	$d_2 = a_4 + b_0 + c_1$	$?(d_3)$	$?(d_4)$
$e_0 = a_3 + b_2 + c_1$	$e_1 = a_4 + b_3 + c_2$	$e_2 = a_0 + b_4 + c_3$	$?(e_3)$	$?(e_4)$

After omitting the obvious checksum calculations, the decoding chain for the erasures would be as follows:

$$a_4(d_2) \rightarrow b_3(e_1) \rightarrow c_4(d_0)$$

$$a_3(e_0) \rightarrow b_4(d_1) \rightarrow c_3(e_2)$$

Each chain above represents a recursion starting from a parity symbol, and in each term of the chain, $x(y)$ means that the symbol x can be recovered from the parity symbol y. Obviously, d_3, d_4, e_3 and e_4 can be easily computed after all others are known.

Case 2. $i = 2$, $j = 4$

Then the remaining array is follows:

a_0	a_1	$?(a_2)$	a_3	$?(a_4)$
b_0	b_1	$?(b_2)$	b_3	$?(b_4)$
c_0	c_1	$?(c_2)$	c_3	$?(c_4)$
$d_0 = a_2 + b_3 + c_4$	$d_1 = a_3 + b_4 + c_0$	$?(d_2)$	$d_3 = a_0 + b_1 + c_2$	$?(d_4)$
$e_0 = a_3 + b_2 + c_1$	$e_1 = a_4 + b_3 + c_2$	$?(e_2)$	$e_3 = a_1 + b_0 + c_4$	$?(e_4)$

Now the decoding chain becomes:

$$c_2(d_3) \to a_4(e_1)$$
$$b_4(d_1)$$
$$b_2(e_0)$$
$$c_4(e_3) \to a_2(d_0)$$

Again, d_2, d_4, e_2 and e_4 are easy to get after all other symbols are obtained.

□

2.3.2 Correcting One Error

To correct one error, the key is to locate the error position. This can be done by computing two *syndrome* vectors from the two parity rows. Since the error is a column error, it is natural to compute the syndromes with respect to *columns* rather than with respect to *rows* as in the encoding procedure. Once the error location is found, the value of the error can be easily computed along the diagonals of either slope.

Suppose $R = [r_{i,j}]_{0 \le i,j \le n-1}$ is the error-corrupted array. Construct two arrays $U = [u_{i,j}]_{0 \le i,j \le n-1}$ and $V = [v_{i,j}]_{0 \le i,j \le n-1}$ from R, where for $0 \le j \le n-1$,

$$u_{i,j} = v_{i,j} = r_{i,j}, \quad 0 \le i \le n-3 \tag{2.4}$$

$$u_{n-2,j} = r_{n-2,j}, v_{n-2,j} = r_{n-1,j} \tag{2.5}$$

$$u_{n-1,j} = v_{n-1,j} = 0 \tag{2.6}$$

i.e., U and V are constructed by copying the $n-1$ information rows and parity rows accordingly from R, then adding an imaginary 0-row at the last row. From U and V, compute two *syndrome* vectors S_0 and S_1 as follows:

$$S_0[i] = \sum_{k=0}^{n-1} u_{i+k,k} \tag{2.7}$$

$$S_1[i] = \sum_{k=0}^{n-1} v_{i-k,k} \tag{2.8}$$

all indices above are mod n.

It is easy to see that the two syndrome vectors are the column checksums along the diagonals of slope 1 and -1 respectively, and that they should be all-zero vectors if there is no error in the array R. If there is one error column in the array R, then the two syndromes are just cyclicly-shifted versions of the error vector with respect to the position of the error column. Thus the location of the error column can be determined simply by a cyclic equivalence test, which tests if one vector is equal to some cyclic shift of another vector. The following example shows how a single error column is reflected in two syndromes for an X-Code of size 5.

Example 2.3 *Syndrome Computation for a* 5×5 *X-Code*

Suppose the 3rd column is an error column, then the two syndrome vectors (S_0 and S_1 respectively) and their corresponding error arrays are as in Fig. 2.1.

$$\begin{array}{|c|c|c|c|c||c|} \multicolumn{5}{c}{} & \multicolumn{1}{c}{S_0} \\ \hline 0 & 0 & 0 & e_0 & 0 & e_3 \\ \hline 0 & 0 & 0 & e_1 & 0 & 0 \\ \hline 0 & 0 & 0 & e_2 & 0 & e_0 \\ \hline 0 & 0 & 0 & e_3 & 0 & e_1 \\ \hline 0 & 0 & 0 & 0 & 0 & e_2 \\ \hline \end{array} \qquad \begin{array}{|c|c|c|c|c||c|} \multicolumn{5}{c}{} & \multicolumn{1}{c}{S_1} \\ \hline 0 & 0 & 0 & e_0 & 0 & e_2 \\ \hline 0 & 0 & 0 & e_1 & 0 & e_4 \\ \hline 0 & 0 & 0 & e_2 & 0 & 0 \\ \hline 0 & 0 & 0 & e_4 & 0 & e_0 \\ \hline 0 & 0 & 0 & 0 & 0 & e_1 \\ \hline \end{array}$$

Figure 2.1: Syndrome computation for a 5×5 X-Code

The two syndromes are actually just the original error column vector (cyclic-)shifted in two different directions for the same number of positions. When they are shifted back, they differ in at most one position; the number of the positions shifted gives the location of the error column. □

The above example almost gives the decoding algorithm for one error correction. A formal algorithm for correcting one error is

Algorithm 2.2 Correcting One Error

Compute two syndrome vectors S_0 and S_1 from the possibly-error-corrupted array R according to the equations Eq. (2.4) through Eq. (2.8). If the two syndromes are both all-zero vectors, then there is no error in the array R; otherwise if there exists such an i that after cyclically down-shifting S_0 by i positions and cyclically up-shifting S_1 by i positions, the first $n-2$ components of the two shifted vectors are equal and the last components of both are zeros then the ith column of the array R is an error column. If no such an i exists, then there is more than one error column in the array R.

A pseudo-code description of the algorithm is as follows:

1. Compute two syndrome vectors S_0 and S_1 from the possibly-error-corrupted array R according to the equations Eq. (2.4) through Eq. (2.8);

2. $i = 0$;

3. **If** $S_0[0..n-3] == S_1[0..n-3]$ **And** $S_0[n-1] == S_1[n-1] == 0$ **Then**

    The error position is the ith column, and the error value is

    $E = (S_0[0], S_0[1], \cdots, S_0[n-3], S_0[n-2], S_1[n-1])$;

  Else If $i == n$ **Then**

    Declare decoding failure : more than one error occurred;

  Else

    $S_0 = S_0^{(1)}, S_1 = S_1^{(-1)}$;

    $i++$;

    Goto 3;

  End If

□

In the above algorithm, for a vector V, denote its transpose V^T; let

$$V = (V[0], V[1], \cdots, V[n-1])^T,$$

then $V^{(1)}$ (or $V^{(-1)}$) is the *down-* (or *up-*) shifted vector from V, i.e.,

$$V^{(1)} = (V[n-1], V[0], \cdots, V[n-2])^T,$$

and

$$V^{(-1)} = (V[1], \cdots, V[n-1], V[0])^T;$$

also $V^{(i)} = (V^{(i-1)})^{(1)}$, $V^{(-i)} = (V^{-(i-1)})^{(-1)}$.

Before proving the correctness of the algorithm, we give a numerical example.

Example 2.4 *Correcting One Error of a* 5×5 *X-Code*

Suppose the possibly-error-corrupted array R is :

0	0	0	1	0
0	0	0	0	0
0	0	0	0	0
0	0	0	1	0
0	0	0	0	0

then U and V, the two constructed arrays from R, and their corresponding syndromes S_0 and S_1 are shown in Fig. 2.2.

U					S_0
0	0	0	1	0	1
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	1	0	0
0	0	0	0	0	0

V					S_1
0	0	0	1	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	1
0	0	0	0	0	0

Figure 2.2: Correcting one error of a 5×5 X-Code

Repeat Step 3 of the algorithm until $i = 3$, then we get $S_0 = (1,0,0,1,0)^T$ and $S_1 = (1,0,0,0,0)^T$, so $S_0[0..2]$ equals to $S_1[0..2]$, and $S_0[4] = S_1[4] = 0$. Thus we declare that the error occurs at the 3rd column and that the error value is $E = (1,0,0,1,0)$, i.e., the uncorrupted array should be an all-zero array. □

Now we give a correctness proof of the algorithm.

Proof: If one error occurs at the ith column, and its value is $e = (e[0], e[1], \cdots, e[n-2], e[n-1])^T$, then the two syndromes (Eq. (2.4) through Eq. (2.7)) are:

$$
\begin{aligned}
S_0 &= ((e[0], \cdots, e[n-3], e[n-2], 0)^T)^{(-i)} \qquad (2.9)\\
S_1 &= ((e[0], \cdots, e[n-3], e[n-1], 0)^T)^{(i)} \qquad (2.10)
\end{aligned}
$$

thus

$$
\begin{aligned}
S_0^{(i)} &= (e[0], \cdots, e[n-3], e[n-2], 0)^T \qquad (2.11)\\
S_1^{(-i)} &= (e[0], \cdots, e[n-3], e[n-1], 0)^T \qquad (2.12)
\end{aligned}
$$

Since the X-Code is an MDS code of column distance 3, it can correct one error, which means the location of a single column error can always be found unambiguously. A unique i can be found such that the two shifted syndrome vectors may differ only in the second last component, and their last components are both 0 (Eq. (2.11) and Eq. (2.12)). Once the error location i is found, the error value is obtained directly from Eq. (2.11) and Eq. (2.12). $\square$

The above algorithm needs $2n(n-2)$ additions to compute the two syndrome vectors, and on average n cyclic equivalence test operations to get the error location.

2.4 Summary

The X-Code, a new class of $n \times n$ *MDS* array codes of distance 3, is presented in this chapter. The significant difference of these codes from all other known array codes is that the parity (redundancy) symbols are placed in two independent rows rather than columns. Additionally, the X-Code has a very simple geometrical structure. Encoding and decoding of the code may be accomplished using only additions (*XOR*s). We have proven that the X-Code is *MDS if and only if* n is prime. For all prime numbers n, the X-Code achieves the lower bound of the update complexity. It also has balanced computation at each column, which might be very helpful in storage systems and distributed computing systems. Finally decoding algorithms for correcting two erasures or one error are given.

One future research problem is to find new MDS codes with optimal update complexity 1) for each positive integer length rather than only for prime lengths, and 2) for distance greater than 3. Our preliminary research shows that only for a few lengths n can the X-Code be easily extended to have larger distance by simply using more parity rows and taking more slopes; in general this is not the case. Extended diagonals, i.e., a set of symbols not necessarily on a straight line of some fixed slope, might be helpful in extending the X-Code to both more general lengths and distances.

Chapter 3 Low Density MDS Codes and Factors of Complete Graphs

3.1 Introduction

In this chapter, we describe another new family of MDS array codes of distance 3 with optimal update complexity. This family is called the B-Code. The B-Code is of size $n \times l$ over an Abelian group $G(q)$ with an addition operation $+$, where $l = 2n$ or $2n+1$, and q is size of the group $G(q)$. As the X-Code in Chapter 2, the B-Code uses only group additions for its encoding and decoding operations as well. The error model is also the same as that of the X-Code: erasures (errors) are column erasures (errors). Its distance is also defined over columns.

The novelty of this chapter is to use a graph approach to describe the code, making the design of the code easier and more direct. Figure 3.1(a) shows $\hat{B}_6$, the *dual B-Code* of length 6. In addition to the usual representation of a code as an array of information

a_1	a_2	a_3	a_4	a_5	a_6
$a_2 + a_3$	$a_3 + a_4$	$a_4 + a_5$	$a_5 + a_6$	$a_6 + a_1$	$a_1 + a_2$
$a_4 + a_6$	$a_5 + a_1$	$a_6 + a_2$	$a_1 + a_3$	$a_2 + a_4$	$a_3 + a_5$

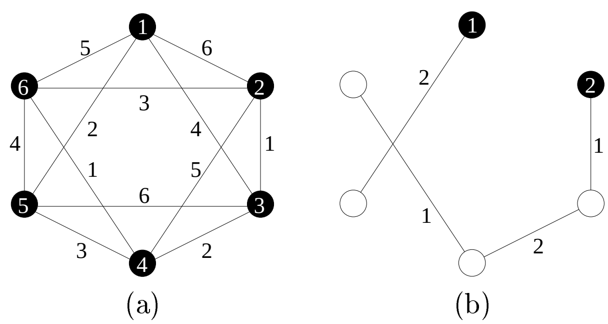


Figure 3.1: $\hat{B}_6$, the dual *B-Code* of length 6, is a 3×6 MDS array code or a $(6, 2, 5)$ MDS code over $G(2^3)$. The a_i's are the information bits. (a) the graph representation of $\hat{B}_6$, (b) a decoding path for the erasure of columns 3, 4, 5 and 6 (i.e., only columns 1 and 2 are available).

and parity bits, the B-Code can be represented by a labeled graph in which every vertex corresponds to an information bit and each edge represents a parity bit: each parity bit is simply the sum of the two information bits that constitute the edge. The edges and vertices of the graph are labeled with a column index: the ith column of the code consists of the information bit a_i and the parity bits with the column index i. The same notation will be used hereafter in this chapter. $\hat{B}_6$ has distance 5 and can therefore tolerate any erasure of 4 columns. Figure 3.1(b) shows a decoding path for the erasure of columns 3 through 6. Use a_2 (from column 2) together with parity $a_2 + a_3$ (from column 1) to recover a_3. Use the latter along with parity $a_3 + a_4$ (from column 2) to recover a_4, etc. For any 4-columns erasure, such a decoding path exists.

By using this new graph description, it will be proven in this chapter that constructing a B-Code is equivalent to a 3-decade old graph theory problem, the *perfect one-factorizations* of complete graphs[29], denoted **P1F**. Using results on P1F, we can construct two infinite families of B-Codes, one of which can be shown to be the construction of [37]. In addition, there are a number of values for which P1Fs exist that are not in the two infinite families; these result in constructions of the B-Codes of all lengths up to 49. The existence of perfect one-factorizations for *every* complete graph with an even number of nodes is a 35-year old conjecture in graph theory. An affirmative answer to this conjecture will provide the B-Code constructions of arbitrary length. Alternately, the construction of the B-Codes of arbitrary odd length will provide an affirmative answer to the conjecture.

The main contributions of this chapter are:

1. proving the *equivalence* of the perfect one-factorization of complete graphs and the MDS code constructions;
2. providing *constructions* for a new class of low-density MDS array codes;
3. proving that in general, the dual of an MDS array code is still MDS.

The chapter is organized as follows. In Section 3.2, we describe the B-Code and its dual using a new graph representation. In Section 3.3, we reveal the relation between the B-Code and the P1F problem. We also give efficient *erasure* and *error* decoding algorithms for the B-Code. In Section 3.4, we further discuss the equivalence between the B-Code and P1F. In Section 3.5, we conclude the chapter and present some future research directions.

3.2 B-Code and its Dual

As already described, a B-Code is an MDS code of size $n \times l$, with distance 3. The MDS property of the B-Code implies that out of nl bits, exactly $2n$ bits should be parity bits. In this section, we describe the B-Code and its dual code using graphs. We also prove that in general the dual of an MDS array code is also MDS.

3.2.1 Structure of the B-Code

Let B_l denote the B-Code of length l, where $l = 2n$ or $2n+1$. For B_{2n}, the first $n-1$ rows are information rows, and the last row is a parity row, i.e., all the bits in the first $n-1$ rows are information bits, while the $2n$ bits in the last row are parity bits. The structure of B_{2n+1} can be derived from that of B_{2n} simply by adding one more information column as the last column. Their structures are shown in Figure 3.2.

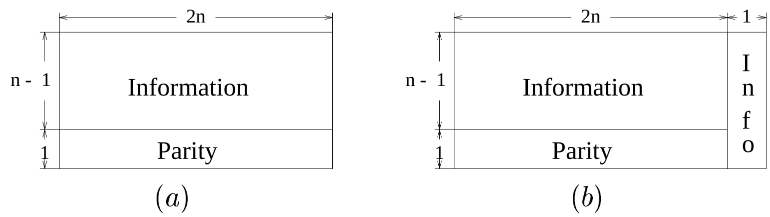


Figure 3.2: Structures of (a) B_{2n} and (b) B_{2n+1}.

Intuitively, if the roles of the information and parity bits of the B-Code are exchanged, i.e., the parity bits are placed in the entries which originally were for the information bits and vice versa, then we get the dual code of the B-Code for length l, denoted $\hat{B}_l$. We will soon give a more rigorous definition of the dual code for general array codes, and prove that the dual of a general MDS array code is also MDS. In particular, the dual B-Code is also an MDS array code; it has distance $l-1$, i.e., the dual B-Code can be recovered from any two of its columns. Figure 3.3 shows the structures of $\hat{B}_{2n}$ and $\hat{B}_{2n+1}$.

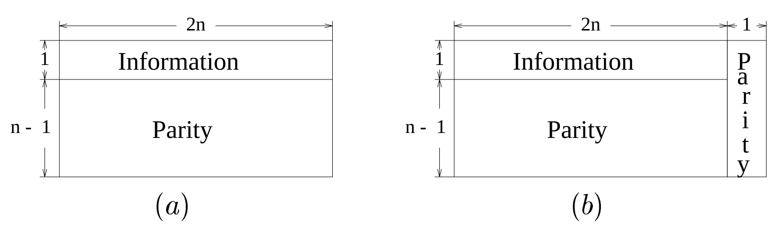


Figure 3.3: Structures of (a) $\hat{B}_{2n}$ and (b) $\hat{B}_{2n+1}$

3.2.2 Dual Array Codes

Array codes are linear codes which can be described by *parity check* or *generator* matrices. Consider an array code of size $n \times l$ over $G(q)$. A codeword of this code can be represented by a vector of length nl over $G(q)$: it consists of l blocks, each of which includes n components. The correspondence between the vector description and the array description is obvious: the ith block of the vector corresponds to the ith column of the array, and the n components within a block are just the n symbols within the corresponding column. A codeword c of a 2×4 array code is shown in both array form and vector form in Figure 3.4.

a_0	a_1	a_2	a_3
p_0	p_1	p_2	p_3

(a)

$$c = (\ a_0 \quad p_0 \mid a_1 \quad p_1 \mid a_2 \quad p_2 \mid a_3 \quad p_3\ )$$

(b)

Figure 3.4: A codeword of 2×4 code in (a) array form and (b) vector form

Using this vector form, an array code of size $n \times l$ with nr parity bits can be described by its parity check matrix $\mathbf{H}$, of size $nr \times nl$, or its generator matrix $\mathbf{G}$, of size $n(l-r) \times nl$; here r is the number of parity (*redundant*) columns as if some columns consist of only parity bits. Like for other 1-dimensional linear block codes, it is easy to observe that for a codeword c of the array code and an information vector m of length $n(l-r)$, the identities that $c = m\mathbf{G}$ and $c\mathbf{H}^T = \mathbf{0}$ still hold, or equivalently $\mathbf{G}\mathbf{H}^T = \mathbf{0}$. In Figure 3.4, let the a_i's be information bits and p_i's be parity bits. Specifically, when $p_i = a_{(i+1)mod4} + a_{(i+2)mod4}$, for $i = 0, 1, 2, 3$, we get a B-Code of length 4, i.e. B_4, with $n = 2$, $l = 4$ and $r = 2$. Its parity check matrix can be described as follows

$$\mathbf{H} = \left(\begin{array}{cc|cc|cc|cc} 0 & 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{array}\right)$$

Accordingly, its generator matrix is as follows

$$\mathbf{G} = \left(\begin{array}{cc|cc|cc|cc} 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 1 & 1 & 0 \end{array}\right)$$

Using the vector form of array codes, we can define dual of array codes as for a conventional 1-dimensional linear block code, i.e.,

Definition 3.1 (*dual array code*) Let C be a linear array code of size $n \times l$ over $G(q)$, then its dual code $C^{\perp}$ is defined as $C^{\perp} = \{ \mathbf{u} \in G(q)^{nl}\colon\ \mathbf{u} \cdot \mathbf{v} = 0$ for all $\mathbf{v} \in$ C $\}$, where $\cdot$ is the conventional *dot product* of vectors.

It follows that, as with 1-dimensional linear block codes, the parity check matrix of an array code is the generator matrix of its dual code. One would expect that other properties of dual codes that hold for 1-dimensional linear block codes also hold for array codes. In particular, the dual of MDS array code is also MDS. ([8] gives a proof for the above statement, but it implicitly assumes that information bits and parity bits are not mixed in a same column.) However, for general array codes, since information and parity bits can be mixed in the same column, it is not as obvious that this property holds as it seems to be. Fortunately, this property can be generalized to general linear array codes, and we will prove it here.

Theorem 3.1 *The dual code of an MDS array code is also MDS.*

Proof: Consider an MDS array code C of size $n \times l$. Suppose its distance is $r+1$ with respect to columns. The parity check matrix of C can then be written as $\mathbf{H} = (h_1\ h_2\ \cdots\ h_l)$, where h_i is a submatrix of size $nr \times n$ that corresponds to the ith block in the vector form of a codeword or to the ith column in the array $(1 \leq i \leq l)$. Since C is MDS, any combination of r submatrices (h_i's) is *linearly independent*, in terms of their columns.

Since $\mathbf{H}$ is the generator matrix of the dual code $C^{\perp}$, let a nonzero codeword $c \in C^{\perp}$ have s nonzero columns, where $s \leq l - r$, thus c has zero-columns in some set of r blocks h_i. Without loss of generality, let these blocks be $(h_1\ h_2\ \cdots\ h_r)$. Since c is by definition a linear combination of the r rows of $\mathbf{H}$ (this still holds for any linear array code), the $nr \times nr$ square submatrix formed by $(h_1\ h_2\ \cdots\ h_r)$ must be *singular*, which contradicts the fact that any combination of r blocks (h_i's) are *linearly independent*. Thus the minimum column weight of any codeword of $C^{\perp}$ must be greater than $l - r$, i.e., the minimum distance of $C^{\perp}$ is greater than $l - r$. By the Singleton bound[31], this shows the dual code $C^{\perp}$ is also MDS. $\Box$

Since 1-dimensional linear block codes are just a special case of array codes, the above theorem certainly holds and the proof above reduces to one of many proofs for 1-dimensional block codes[31].

3.2.3 A New Graph Description of the B-Code

Typically, an array code is described by its *geometrical construction lines* [4][5][7][15], or by its *parity check matrix* [8][37]. Constructions of array codes are difficult to get using these descriptions. In this chapter, we describe the B-Code and its dual using a new graph approach. By relating the graph conditions for constructing the B-Code to a classical graph problem, *perfect one-factorization* of complete graphs, we obtain new constructions.

For any array code, each parity bit is the sum of some information bits; for binary codes, the addition is just the simple XOR (binary *exclusive OR*) operation. If a parity bit P is the sum of an information bit I and other information bits, then we say that the information bit I *appears* in the parity bit P. Now consider the dual B-Code $\hat{B}_l$. Because of its MDS and optimal encoding properties, each information bit must appear *exactly* $l-2$ times in the parity bits. Since the numbers of the total information and parity bits are $2n$ and $nl-2n$ respectively, each parity bit must be the sum of $\frac{2n(l-2)}{nl-2n}$ or *exactly* 2 information bits. (This is reflected in the parity check matrix by the fact that the weight of each row is exactly 3). So if we represent an information bit as a vertex, then a parity bit can be represented by an edge, where the parity bit is the sum of the two information bits whose vertices form the edge. This is the key idea of describing the B-Code and its dual with graphs.

Since the construction of $\hat{B}_{2n}$ can be obtained from $\hat{B}_{2n+1}$ simply by deleting the last parity column, here we focus on the graph description of $\hat{B}_{2n+1}$. $\hat{B}_{2n+1}$ has $2n$ information bits and $n(2n-1)$ parity bits, which can be represented exactly by a *complete* graph of $2n$ vertices, K_{2n}, which also has exactly $\binom{2n}{2} = n(2n-1)$ edges. The mapping is simple: one information bit can be represented by one vertex, and the parity bit that is the sum of 2 information bits can be represented by the edge that links the 2 corresponding vertices. So the only remaining problem is to define on K_{2n} the grouping relation that determines which information and parity bits occupy the same column of the code. This can be thought of as labeling the vertices and edges of the complete graph K_{2n} in such a way that information bit and parity bits in the same column are labeled with the same label. Since $\hat{B}_{2n+1}$ has $2n+1$ columns, we need $2n+1$ labels. Notice that the each of the first $2n$ columns has exactly 1 information bit and $n-1$ parity bits, and that the last column has n parity bits. A formal way of describing the $\hat{B}_{2n+1}$ is as follows:

Description 3.1 Graph Description of $\hat{B}_{2n+1}$

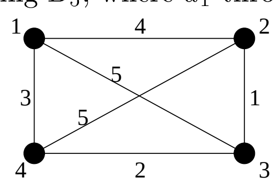
Given a complete graph K_{2n} with 2n vertices, which are labeled with integers from 1 to 2n, find an edge labeling scheme such that

1) each edge is labeled exactly once by an integer from 1 to 2n+1

2) For any pair of vertices (i, j) and any other vertex k, where $i, j, k \in [1, 2n]$, there is always a path to k from either i or j, using only the edges labeled with i or j.

3) For any vertex i and any other vertex k, where $i, k \in [1, 2n]$, there is always a path from i to k, using only the edges labeled with i or $2n + 1$.

With the above description, it is easy to see that the vertex and edges with the label i in the K_{2n} represent the information bit and parity bits in the ith column of $\hat{B}_{2n+1}$. The properties 2) and 3) ensure that any two columns of the code can recover the information bits in all other columns, thus the code is of column distance $2n$. Figure 3.5 shows such a labeling of K_4 and the corresponding $\hat{B}_5$, where a_1 through a_4 are the information bits.



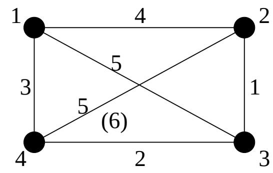
(a)

a_1	a_2	a_3	a_4	$a_1 + a_3$
$a_2 + a_3$	$a_3 + a_4$	$a_4 + a_1$	$a_1 + a_2$	$a_2 + a_4$

(b)

Figure 3.5: (a) graph and (b) array representations of $\hat{B}_5$

Naturally, if the edges of K_{2n} are used to represent information rather than parity bits, and vertices to represent the parity bits, it should be expected that by using such a labeling scheme and reindexing the edges, such a complete graph can represent B_{2n+1}, i.e., the B-Code itself. And in fact this is true. In the graph representation of B_{2n+1}, a parity bit is the sum of all the information bits whose edges are incident with its vertex. B_{2n} can easily be obtained from B_{2n+1} by setting all the information bits in the last column to zero and then deleting them after the parity bits are changed accordingly. B_5 is shown in Figure 3.6, where the edge labeled with (6) represents the information bit a_6 in the 5th column. It is also interesting to point out that B_5 happens to be a *perfect code* too, i.e., it achieves the *Hamming Bound*[31].



(a)

a_1	a_2	a_3	a_4	a_5
$a_3 + a_4 + a_5$	$a_4 + a_6 + a_1$	$a_5 + a_1 + a_2$	$a_6 + a_2 + a_3$	a_6

(b)

Figure 3.6: (a) graph and (b) array representations of B_5

3.3 B-Code and P1F

As already described in Section 3.2, constructing the B-Code amounts to the same problem as designing an edge labeling scheme such as in Description 3.1 for a complete graph K_{2n}. Fortunately this can be related to another graph theory problem, namely the *perfect one-factorization* problem.

3.3.1 Perfect One-Factorization of Complete Graphs

Definition 3.2 [30] Let *G=(V,E)* be a graph. A *factor* or *spanning subgraph* of G is a subgraph with vertex set V. In particular, a *one-factor* is a factor which is a regular graph of degree 1. A *factorization* of G is a set of factors of G which are pairwise *edge disjoint*, and whose union is all of G. A *one-factorization* of G is a factorization of G whose factors are all one-factors. In particular, a one-factorization is *perfect* if the union of any pair of its one-factors is a *Hamilton cycle*, a cycle that passes through every vertex of G.

Figure 3.7 shows a perfect one-factorization of K_4. A perfect one-factorization of K_6 is shown in Figure 3.8(b), where edges with the same label form a one-factor.

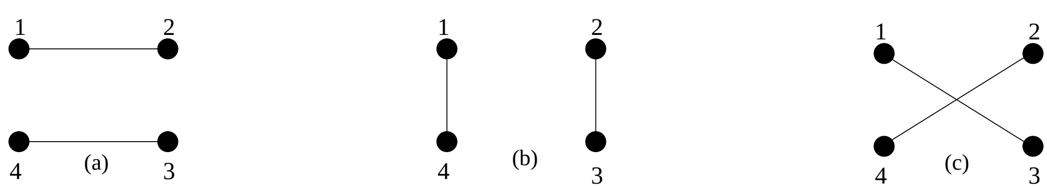


Figure 3.7: (a)(b)(c) are 3 one-factors, that together form a perfect one-factorization of K_4

The perfect one-factorization of complete graphs has been studied for many years since its introduction in [16]. It is now known that[30]:

Theorem 3.2 *If p is an odd prime, then K_{p+1} and K_{2p} have perfect one-factorizations.*

Constructions of P1F for K_{p+1} and K_{2p} can be found in [2] and [29]. Additionally, constructions of P1F for K_{2n}'s whose n's are some other sporadic integers have also been found[29][30]. However it still remains a conjecture [29][30] that:

Conjecture 3.1 *For any positive integer n, K_{2n} has perfect one-factorization(s).*

3.3.2 Equivalence between the B-Code and P1F

Let P_{2n+2} be a P1F for K_{2n+2}. Recall that $\hat{B}_{2n+1}$ has $2n+1$ columns, and P_{2n+2} also has $2n+1$ one-factors. So, if we are able to find a 1–to–1 mapping between the columns and one-factors, then we can get constructions for $\hat{B}_{2n+1}$ from P_{2n+2}, and vice versa. Luckily enough, such a mapping does exist. The following two algorithms give such a 1–to–1 mapping.

Algorithm 3.1 *Constructing $\hat{B}_{2n+1}$ from P_{2n+2}*

Step 1. Label the vertices of K_{2n+2} with $0, 1, \cdots, 2n, \infty$;

Step 2. If a P1F exists for K_{2n+2}, then let F_i denote the one-factor which contains the edge $0i$, where $i = 1, 2, \cdots, 2n, \infty$;

Step 3. In each F_i, delete the two vertices 0 and ∞ and all the edges which are incident with either of them; For $i = 1, 2, \cdots, 2n$, label all the remaining edges in F_i with i, and label all the remaining edges of F_∞ with $2n+1$.

Figure 3.8 shows the construction of $\hat{B}_5$ from P_6, where in (a) ∞ is replaced with 5, and the edges with the same label i form the one-factor F_i.

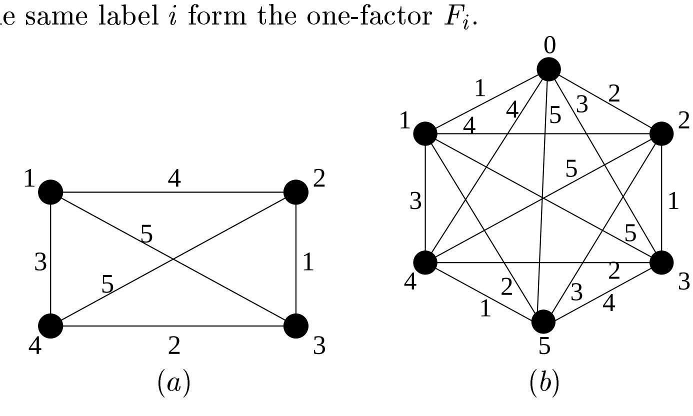


Figure 3.8: Constructing (a) $\hat{B}_5$ from (b) P_6

Theorem 3.3 Algorithm 3.1 *gives a graph as described in* Description 3.1, *i.e., a construction of* $\hat{B}_{2n+1}$.

Proof: First observe that in a P1F of K_{2n+2}, each edge appears exactly once in the whole set of the one-factors, thus step 2 is feasible. Now check the conditions of the graph description of $\hat{B}_{2n+1}$:

1) obviously holds;

2) Since F_i and F_j $(i, j \in [1, 2n])$ are two one-factors of a P_{2n+2}, their union is a Hamilton cycle of $2n+2$ vertices, after deleting the vertices 0 and ∞ and the four edges incident with them, the Hamilton cycle breaks into two paths , covering all the remaining vertices from 1 to $2n$. These paths start from i or j, thus this condition holds.

3) Since F_i and F_∞ $(i \in [1, 2n])$ form a Hamilton cycle of $2n+2$ vertices where 0∞ is an edge, after the deletion of the vertices 0 and ∞ and the *three* edges incident with them, the Hamilton cycle becomes *one* path starting from i, thus this condition holds. □

Since B_{2n+1} and $\hat{B}_{2n+1}$ can be described with the same complete graph K_{2n}, both B_{2n+1} and $\hat{B}_{2n+1}$ can be constructed from P_{2n+2}. Additionally, B_{2n} and $\hat{B}_{2n}$ can be easily obtained from B_{2n+1} and $\hat{B}_{2n+1}$, so the B-Code and its dual (of size $n \times l$) can be constructed from the known P1F constructions of K_{2n+2}. In particular, from *Theorem 3.2*

Theorem 3.4 *For any odd prime p, a B-Code and its dual code of size $n \times l$ can be constructed, where n is either $\frac{p-1}{2}$ or $p-1$.*

When n=$\frac{p-1}{2}$, the corresponding B-Code is the code in [37][8]. The B-Code of $n = p-1$ was not known before. The next natural question is : Can we get P_{2n+2} from B_{2n+1}? The answer is *yes* and the following algorithm can do it.

Algorithm 3.2 *Constructing P_{2n+2} from $\hat{B}_{2n+1}$*

Step 1. If $\hat{B}_{2n+1}$ exists, use Description 3.1 *of $\hat{B}_{2n+1}$, let $\tilde{F}_i$ denote the set of edges with the label i, where $i = 1, 2, \cdots, 2n$, and let $\tilde{F}_\infty$ denote the set of the edges with the label $2n+1$;*

Step 2. Add two vertices 0 and ∞ to K_{2n};

Step 3. For $i = 1, 2, \cdots, 2n$ and ∞, add the edges $i0$ and $k\infty$ to $\tilde{F}_i$, where k is integer from 1 to 2n such that the expanded set, F_i, is a one-factor of the complete graph K_{2n+2} of vertices $0, 1, 2, \cdots, 2n, \infty$.

Theorem 3.5 Algorithm 3.2 *gives a* P_{2n+2}.

Proof: Observe that because of the MDS and optimal encoding properties of $\hat{B}_{2n+1}$, in each of the first $2n$ columns of $\hat{B}_{2n+1}$:

1) each information bit appears at most once;

2) there is exactly one bit which does *not* appear; also *no* pair of the columns miss the same bit, since otherwise that bit can *not* be recovered from just these two columns;

3) in the last column, each bit appears exactly once.

Thus 2) guarantees that in Step 3, there exists a unique k. Further, 1) ensures that for any pair of columns i and j, where $i, j = 1, 2, \cdots, 2n$, the two vertices i and j can only be the endpoints of the two paths in the graph description. Thus Step 3 of the above algorithm makes the union of any pair of F_i and F_j, where $i, j = 1, 2, \cdots, 2n$, into a Hamilton cycle. Step 3 also makes the union of any F_i ($i = 1, 2, \cdots, 2n$) and F_∞ a Hamilton cycle. Thus $\{F_1, F_2, \cdots, F_{2n}, F_\infty\}$ is a P1F of K_{2n+2}, i.e., it is P_{2n+2}. $\square$

Theorem 3.3 and Theorem 3.5 reveal a surprising result:

Theorem 3.6 *Constructing* $\hat{B}_{2n+1}$ (*or equivalently* B_{2n+1}) *is equivalent to constructing* P_{2n+2}, *i.e.,* $P_{2n+2} \Longleftrightarrow B_{2n+1}$.

Note that the equivalence does not include the B-Codes of even length, i.e., B_{2n}. This equivalence, however, already shows that any progress in P1F gives a new B-Code, and vice versa.

3.3.3 Erasure Decoding of the B-Code

Obviously the encoding of the B-Code can be done using Algorithm 3.1. Now consider erasure decoding for the B-Code. Recall that the dual B-Code can recover all information bits from any two columns. Erasure decoding for the dual B-Code is almost obvious from its graph description (*Description 3.1*). The two paths, starting from i and j and leading to all the other vertices in the graph, give the decoding chain used in recovering a $\hat{B}$-Code from its ith and jth columns. Figure 3.9 shows the decoding chain used in recovering $\hat{B}_5$ from its 1st column and its 2nd, 3rd and 5th columns, respectively.

The B-Code can recover any two missing columns. Decoding for the B-Code itself is almost the same as for its dual, except that the roles of edges and vertices are exchanged. Figure 3.10 shows the decoding chains for recovering B_5's 1st column and 2nd, 3rd, and 5th

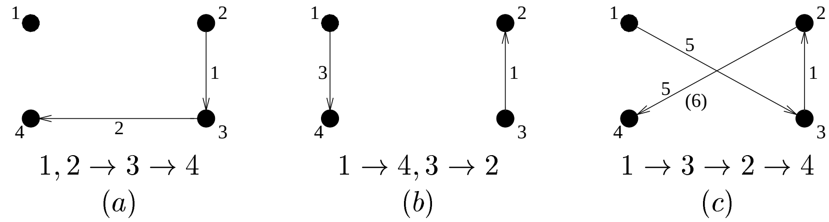


Figure 3.9: Erasure decoding of $\hat{B}_5$: recovering from its 1st and (a) 2nd (b) 3rd and (c) 5th columns. The decoding chains for each case are also listed. 1 through 4 are the information bits in the corresponding columns.

columns respectively. Comparing the decoding sequences here with those of $\hat{B}_5$, it is easy to obverse that the decoding chains for recovering the ith and jth columns of B_l are just the reversed sequences of those for recovering its dual code $\hat{B}_l$ from its ith and jth columns. This also shows that the two codes are dual to each other, since their graph descriptions are dual to each other.

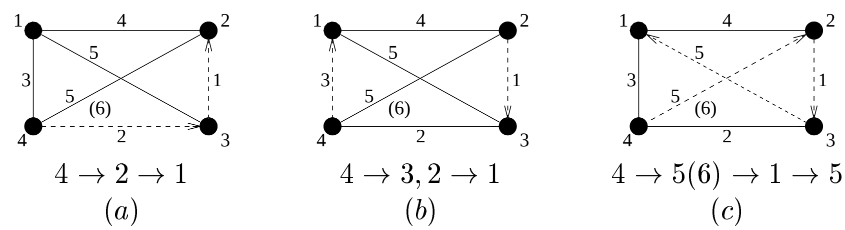


Figure 3.10: Erasure decoding of B_5: recovering its 1st and (a) 2nd (b) 3rd and (c) 5th columns. The decoding chains for each case are also listed. 1 through 6 are the information bits in the corresponding columns, except that 6 is also in the 5th column.

Finally, the above decoding algorithms can be summarized as follows:

Algorithm 3.3 *Erasure decoding of the dual B-Code*

The edges labeled i and j create two paths which span the vertex set. Starting at vertex i and at vertex j, use these paths, by adding a known information bit and a known parity bit to recover a new unknown information bit. Repeating this step along each path recovers the dual B-code from its ith and jth columns.

Algorithm 3.4 *Erasure decoding of the B-Code*

To recover the ith and jth columns of the B-Code, use the same paths with edges labeled with i and j. This time, traverse the paths in the opposite directions of the corresponding paths for the dual B-Code. Along each path, add all known information and parity bits to *get a new unknown information bit. Repeating this step along each path recovers the ith and jth columns of the B-Code from the other $n-2$ columns.*

3.3.4 Error Decoding of the B-Code

Recall that a B-Code of size $n \times l$ is of distance 3, so it can correct one error. To do this, the key is again to locate the error location; the error value can easily be determined once the location is found. One way to find the error location is to make a table that maps syndromes to single-error locations, and then do a table lookup after calculating the syndrome of a received array. The drawbacks are 1) such a table is needed for each B-Code and 2) table lookup is not efficient in both computation time and space (since the total number of 1-error syndromes is 2^n). Another rather straightforward algorithm is to consider the ith column and (i+1)th column to be erasures (where $i = 1, 3, 5, \cdots, 2n-1$ for $l = 2n$; if $l = 2n+1$, the lth column and the 1st column are also included), and then recover those columns. If exactly one of the recovered columns differs from the original ones, then that discrepant column is the error column. This algorithm can correct one error. The algorithm requires *on average* $\frac{n}{2}$ erasure decodings, each of which needs $2n(l-3)$ additions, thus the average total number of additions is $n^2(l-3)$, which is in the order of n times of $n \times l$. Another shortcoming is that the algorithm will give a *false* decoding result if more than one error occurs.

We present here a more efficient decoding algorithm for correcting one error. Observe the relation between a B-Code and its dual from their graph descriptions: if an information bit i of a B-Code appears in a set P of parity-bit positions, then in its dual code, the elements of P will be information bits and i will be then a parity bit; further, all the elements of P appear in the parity bit i. Thus, if there is a single column error in a received array of a B-Code, use the syndrome of this received array as the information vector of the dual code. In the obtained dual codeword, the parity bits in the error column of the B-Code should be zero, while other parity bits are nonzero because of the structural properties of the B-Code observed in the proof of Theorem 3.5. This differentiates the error location from other columns. The decoding algorithm can be described semi-formally as follows:

Algorithm 3.5 *Error Decoding of the B-Code*

1. *Given a received array R of size $n \times l$, calculate its syndrome, denoted as S (which*

is a vector of length $2n$ *);*

2. If S *is a zero vector, then the received array* R *is a codeword of* B_l*; otherwise go to next step;*

3. Use S *as the information vector of the dual B-Code* $\hat{B}_l$*, encode to get a codeword* C *of* $\hat{B}_l$*;*

4. If the weight of the syndrome S *is* even, *and if there is a* unique *all-zero column in* C*, then this is the error column of the original received array* R*. On the other hand, if the weight of syndrome* S *is* odd, *and if there is a* unique *column whose information bit is nonzero and whose parity bits are all zero, then this is the error column of* R*;*

5. If the error column of R *is found in the above step, regard this column as an erasure and recover it; otherwise declare decoding failure: there are at least two error columns in* R*.*

Notice that the above property holds only when the B-Code is defined in $G(2^m)$, i.e., each cell of the B-Code consists of a block of m *binary* bits. However, this decoding algorithm can be modified to work for the general B-Codes defined in $G(q^m)$, where q is not a power of 2. Here we will stick to the case where $q = 2$ and $m = 1$.

Before we prove the correctness of the above algorithm, we show an example of it.

Example 3.1 *Error-correcting for the B-Code*

Consider B_6, whose graph description is shown in Fig. 3.1. (Its array description is easy to get from either its graph description or the array description of $\hat{B}_6$, as in Fig. 3.1.) If two received arrays are as follows:

$$R_1 = \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline\hline 0 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array} \qquad R_2 = \begin{array}{|c|c|c|c|c|c|} \hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline\hline 1 & 0 & 0 & 0 & 0 & 0 \\ \hline \end{array}$$

Then the two syndromes are respectively:

$$S_1 = \begin{array}{|c|c|c|c|c|c|} \hline 0 & 1 & 1 & 1 & 0 & 1 \\ \hline \end{array} \qquad S_2 = \begin{array}{|c|c|c|c|c|c|} \hline 1 & 1 & 1 & 1 & 0 & 1 \\ \hline \end{array}$$

Since neither S_1 or S_2 is a zero vector, both R_1 and R_2 have errors. Now use S_1 and S_2 as information vectors of $\hat{B}_6$, whose graph and array descriptions are shown in Fig. 3.1. We get two codewords of $\hat{B}_6$

$$C_1 = \begin{array}{|c|c|c|c|c|c|}\hline 0 & 1 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 1 & 1 & 1 & 1 \\ \hline 0 & 0 & 0 & 1 & 0 & 1 \\ \hline \end{array} \qquad C_2 = \begin{array}{|c|c|c|c|c|c|}\hline 1 & 1 & 1 & 1 & 0 & 1 \\ \hline 0 & 0 & 1 & 1 & 0 & 0 \\ \hline 0 & 1 & 0 & 0 & 0 & 1 \\ \hline \end{array}$$

Since the weight of S_1 is even, and column 1 is the unique all-zero column in C_1, column 1 is the error column of R_1; on the other hand, since the weight of S_2 is odd, and column 1 is the unique column in C_2 whose information bit is nonzero and whose parity bits are all zeros, column 1 is the error column of R_2 too. Once the error column of R_1 (R_2) is found, the error value is easy to get. The corrected arrays of R_1 and R_2 are both all-zero arrays. $\square$

Now we prove the correctness of the decoding algorithm.

Proof: The following can be observed from the graph description of the B-Code: each information bit appears in exactly *two* parity bits, and this information bit and the two parity bits are in three different columns; in addition, two information bits from the same column can *not* appear in the same parity bit, thus all possible errors in the information bits of a single column contribute *even* weight to its syndrome vector. On the other hand, single parity-bit error adds exactly *one* to the weight of the syndrome vector, i.e., a parity-bit makes the weight of the syndrome *odd*.

Suppose there is only one error column in a received array of a B-Code. Call this column the ith column. Consider the following two cases:

Case 1: *All errors occur in information bits.* Then the syndrome should be of *even* weight. Now we prove that the ith column of the obtained codeword of the dual B-Code is the only all-zero column:

1). *The ith column is an all-zero column.* This is true because of the relation between the B-Code and its dual : in the B-Code, an information bit i appears in two parity bits P_1 and P_2, and in the dual B-Code these two bits $\hat{P}_1$ and $\hat{P}_2$ are information bits, and the bit $\hat{i}$ is a parity bit such that both $\hat{P}_1$ and $\hat{P}_2$ appear in $\hat{i}$. Since $P_1 = P_2 = i$ and $\hat{P}_1 = \hat{P}_2$, $\hat{i} = 0$. Thus all parity bits of the ith column of the dual B-Code are zero. Additionally, since there is no error in the parity bit of the B-Code, the information bit of the ith column of the dual B-Code is also zero. So the entire ith column of the dual B-Code is zero.

2). *All the other columns are nonzero columns.* If there were at least one more all-zero column in the codeword, then the weight of the codeword would be no greater than $l-2$, which contradicts the fact the minimum distance of the dual B-Code is $l-1$. Here l is the length of the codeword.

Case 2: *An error also occurs in the parity bit as well.* In the dual B-Code, among all the columns which have both an information bit and parity bits (if the length of the dual B-Code is odd, there is one column containing no information bit), by the linearity of the dual B-Code, the ith column has a nonzero information bit, and all its parity bits are zero. No other column can have a nonzero information bit and all zero parity bits. The reason is as follows: the weight of the information bits in the dual B-Code is *odd*, and all the information bits appear in the ith column. Since each information bit is missing from exactly one of the columns, the number of nonzero information bits that appear in the jth column ($j \neq i$) is *even*. Thus if the information bit of the jth column is nonzero, then at least one of its parity bits is nonzero, since each parity bit is the sum of two information bits, and the total number of information bits which appear in the parity bits is now *odd*.

When multiple column errors occur, there can be multiple all-zero columns or multiple columns with the first component nonzero and all other components zero.

This concludes the proof for the correctness of the decoding algorithm. $\square$

The complexity of the above decoding algorithm is easy to analyze. For a received array of size $n \times l$, the syndrome calculation requires $2(l-2)n$ additions; the encoding of the dual B-Code requires $(l-2)n$ additions; finally, correcting one erasure requires $(l-3)n$ additions. This adds up to $(4l-9)n$ additions, which is linear in the number of total bits in an array of size $n \times l$. The same trick used here cannot be applied directly to correct multiple errors for the dual B-Code, since multiple errors can weave together and cannot be easily separated. In general, it still remains a challenge to correct multiple errors efficiently (total additions linear in total number of bits in an array) for array codes.

3.4 Further Equivalence Discussion

The equivalence between the B-Code and P1F has been shown in the above section. It is quite clear that B_{2n} can be constructed from B_{2n+1} simply by *shortening*, namely, setting all the information bits in the last column to *zero*. Similarly, $\hat{B}_{2n}$ can be derived from $\hat{B}_{2n+1}$

by *puncturing*, i.e., deleting the last parity row. The relations among P_{2n+2}, B_{2n+1} and B_{2n} can be described as follows, where $\Longrightarrow$ means to *lead to*:

$$P_{2n+2} \Longleftrightarrow B_{2n+1} \Longrightarrow B_{2n}$$

A further question is whether B_{2n+1} (or $\hat{B}_{2n+1}$) can be constructed from a known construction of B_{2n} (or $\hat{B}_{2n}$), i.e., whether the last $\Longrightarrow$ can be replaced with $\Longleftrightarrow$. Our conjecture is *yes*.

Conjecture 3.2 *For any positive integer n, $\hat{B}_{2n+1}$ (or B_{2n+1}) can be constructed from $\hat{B}_{2n}$ (or B_{2n}) using* Algorithm 3.6.

Algorithm 3.6 *Constructing $\hat{B}_{2n+1}$ from $\hat{B}_{2n}$*
Extend a given $\hat{B}_{2n}$ by adding one more column, which contains, as parity bits, all the unused *or* unlabeled *edges in the graph description of $\hat{B}_{2n}$.*

The B-Codes shown in Figure 3.1, Figure 3.5 and Figure 3.6 are all what we call *shift codes*, and it is easy to verify *Conjecture 2* is true for these examples.

Definition 3.3 (*Shift Code*) An array code (of size $n \times l$) is called a *shift* code if any row of its parity check matrix is just a cyclic shift of the first row, i.e., the remaining columns of the code can be constructed by cyclically shifting the first column.

In general, for a shift B-Code, *Conjecture 2* can be proven true, namely,

Theorem 3.7 *For any shift B-Code, $\hat{B}_{2n+1}$ (or B_{2n+1}) can be constructed from $\hat{B}_{2n}$ (or B_{2n}) using* Algorithm 3.6.

Proof: Given a shift dual B-Code, $\hat{B}_{2n}$, notice that the missing edges are the diagonals, $(i, i+n)$, (addition is modulo $2n$). Indeed, if $(i, i+n)$ were present in column j of $\hat{B}_{2n}$, then it would be included in column $n+j$ as well, because of the shift property, making the code non-MDS.

To complete the proof, we need to show that by using an arbitrary column, j, of $\hat{B}_{2n}$ together with the diagonals $(i, i+n)$, $0 \leq i < n$, one can recover all remaining $2n-1$ columns, i.e. we indeed have $\hat{B}_{2n+1}$. Suppose that is not true. There exists a column j, in which a set of edges, combined with the diagonals, form a loop:

$$a_1 + n + a_2 + n + ... + a_q + n = 0 \text{ mod 2n}$$

where q is the number of edges involved in the loop, a_i's are their lengths and n is the length of the diagonals. For example let $n = 6$, $q = 3$, $a_1 = 1$, $a_2 = 2$ and $a_3 = 3$:

$$1 + 6 + 2 + 6 + 3 + 6 = 24 = 0 \text{ mod } 12$$

We will show that this cannot happen. We will show that column $j+n$ and column j form a loop and therefore the original code is not $\hat{B}_{2n}$. Using the above equation:

$$\sum_{i=1}^{q} a_i = qn$$

Because column $j+n$ is a cyclic shift of column j, it contains a set of edges of lengths $b_i = a_i$ such that A_1 connects to B_2, which in turn connects to A_3, etc.

$$\sum_{i=1}^{q} a_i + \sum_{i=1}^{q} b_i = 2qn = 0 \text{ mod 2n}$$

There is a loop. □

If *Conjecture 2* can be proven true for any arbitrary B-Code, then we can get a *strong* equivalence between the B-Codes and P1F, i.e, the B-Code construction is completely equivalent to the P1F construction.

3.5 Summary

In this chapter we have presented the B-Code and its dual, a new class of optimal MDS array codes of size $n \times l$ (where $l = 2n$ or $2n+1$) with distance 3 (or $l-1$ for the dual code). We proved an equivalence between the B-Code and perfect one-factorizations using a new graph description of the B-Code. We also described encoding and decoding algorithms for the B-Code and its dual based on their graph descriptions. There are a number of open problems: (i) are the B-Code constructions strongly equivalent to perfect one-factorizations? (ii) can the graph description of the B-Codes be extended to design optimal array codes of arbitrary distance? (iii) how does one efficiently correct multiple errors for the dual B-Code (or other array codes)? and the ultimate question, (iv) can coding theory techniques be used to solve the P1F conjecture?

Chapter 4 Efficient Deterministic Voting in Distributed Systems

4.1 Introduction

In distributed storage systems, particularly for distributed file systems, voting can be used to keep replicated data consistent. Distributed voting is itself an important problem in the creation of fault-tolerant computing systems, e.g., it can be used to keep distributed data consistent and to provide mutual exclusion in distributed systems. In an N Modular Redundant (*NMR*) system, when the N computational modules execute identical tasks, they need to be synchronized periodically by voting on the current computation state (or result, and they will be used interchangeably hereafter), and then all modules set their current computation state to the majority one. If there is no majority result, then other computations are needed, e.g., all modules recompute from the previous result. This technique is also an essential tool for task-duplication-based checkpointing[38].

Many aspects of voting algorithms have been studied, e.g., data approximation, reconfigurable voting, dynamic modification of voting weights, and metadata-based dynamic voting[9][18][24]. In this chapter, we focus on the *communication complexity* of the voting problem. Several voting algorithms have been proposed to reduce the *communication complexity*[10][20]. These algorithms are nondeterministic because they perform voting on signatures (hash functions) of local computation results. Recently, a majority voting scheme based on error-control codes[21] was proposed: each module first encodes its local result into a codeword of a designed error-detecting code and sends part of the codeword. By using the error-detecting code, discrepancies of the local results can be detected with some probability, and then by retransmitting the full local results, a majority voting decision can be made. Though the scheme drastically reduces the *average case* communication complexity, it can still fail to detect some discrepancies of the local results and might reach a *false* voting result, i.e., the algorithm is still a probabilistic one. In addition, this scheme uses only the error-detecting capabilities of codes. As this chapter will show, in general, using only *error-detecting codes* (*EDC*) does not help to reduce the communication complexity of a deterministic voting algorithm. Though they have been used in many applications such as reliable distributed data replication[1], *error-correcting codes* (*ECC*) have not been applied to the voting problem.

For many applications[38], *deterministic* voting schemes are needed to provide more accurate voting results. In this chapter, we propose a novel *deterministic* voting scheme that uses error-correcting codes. The voting scheme generalizes all known simple deterministic voting algorithms. Our main contributions related to the voting scheme include: (i) using the correcting capability in addition to the detecting capability of codes (only the detection was used in known schemes) to drastically reduce the chances of retransmission of the whole local result of each node, thus reducing the communication complexity of the voting, (ii) a proof that our scheme reaches the same voting result as the naive voting algorithm in which every module broadcasts its local result to all the other modules, and (iii) a method of tuning the scheme for optimal average case communication complexity by choosing the parameters of the error-correcting code, thus making the voting scheme adaptive to various application environments with different error rates.

The chapter is organized as follows: in Section 4.2, we describe the majority voting problem in NMR systems. Our voting algorithm together with its correctness proof are described in Section 4.3. Section 4.4 analyzes both the worst case and the average case communication complexity of the algorithm. Section 4.5 presents experimental results of the performance of the proposed voting algorithm, as well as the performance of two other simple voting algorithms for comparison. Section 4.6 concludes the chapter.

4.2 The Problem Definition

In this section, we define the model of the NMR system and its communication complexity. Then we address the voting problem in terms of the communication complexity.

4.2.1 NMR System Model

An NMR system consists of N computational modules which are connected via a communication medium. For a given computational task, each module executes a same set of instructions with independent computational error probability p. The communication medium could be a bus, a shared memory, a point-to-point network or a broadcast network.

Here we consider the communication medium as a *reliable broadcast network*, i.e., each module can send its computational result to all the other modules with only one error-free communication operation. The system evolution is considered to be synchronous, i.e., the voting process is round-based.

4.2.2 Communication Complexity

The *communication complexity* of a task in an NMR system is defined as the *total* number of bits that are sent through the communication medium during the whole execution of the task. In a broadcast network, let m_{ij} be the number of the bits that the ith module sends at the jth round of the execution of a task, then the communication complexity of the task is $\sum_{i=1}^{N}\sum_{j=1}^{K} m_{ij}$, where N is the number of the modules in the system, and K is the number of rounds needed to complete the task.

4.2.3 The Voting Problem

Now consider the voting function in an NMR system. In order to get a final result for a given task in an NMR system, each module completes its own computation separately, then it must be synchronized with the other modules by voting on the result. Let X_i denote the local computational result of the ith module. The *majority function* is defined as follows:

$$Majority(X_1, \cdots, X_N) := \begin{cases} X & if \ |\{1 \leq i \leq N\} : X_i = X| \geq \frac{N+1}{2} \\ \phi & otherwise \end{cases}$$

where in general, N is an odd natural number, and ϕ is any predefined value different from all possible computing results.

Example 4.1 *Majority voting*

If $X_1 = 0000$, $X_2 = 0001$, $X_3 = 0100$, $X_4 = 0000$, $X_5 = 0000$, then

$$Majority(X_1, X_2, X_3, X_4, X_5) = 0000;$$

If X_5 changes to 0010, and the other $X_i's$ remain unchanged, then

$$Majority(X_1, X_2, X_3, X_4, X_5) = \phi.$$

□

The result of voting in an NMR system is that each module gets $Majority(X_1, \cdots, X_N)$ as its final result , where X_i $(i = 1, \cdots, N)$ is the local computation result of the ith module.

One obvious algorithm for the voting problem is that after each module computes the task, it broadcasts its own result to all the other modules. When a module receives all the other modules' results, it simply performs the majority voting locally to get the result. The algorithm can be described as follows:

Algorithm 4.1 *Send-All Voting*

For Module P_i, $i \in [1 : N]$:

Broadcast(X_i);

Wait Until Receive All X_j, $j \in [1 : N] \setminus i$;

$X := Majority(X_1, \cdots, X_N)$;

Return(X);

□

This algorithm is simple: each module only needs one communication (i.e., broadcast) operation, but apparently its communication complexity is too high. If the result for the task has m bits, then the communication complexity of the algorithm is Nm bits. In most cases, the probability of a module having a computational error is rather low, namely at most times all modules have the same result, thus each module only needs to broadcast part of its result. If all the results are identical, then each module agrees with that result. If not, then the modules can use *Algorithm 4.1*. In other words, we can improve the voting algorithm as follows:

Algorithm 4.2 *Simple Send-Part Voting*

For Module P_i, $i \in [1 : N]$:

Partition the local result X_i into N symbols: $X_i[1 : N]$;

Broadcast($X_i[i]$);

Wait Until Receive All $X_j[j]$, $j \in [1 : N] \setminus i$;

$X := X_1[1] * \cdots * X_N[N]$;

$F_i := (X = X_i)$;

Broadcast(F_i);

If $Majority(F_1, \cdots, F_N) = TRUE(1)$

Return(X);

Else

Broadcast$(X_i[j])$, $j \in [1:N] \setminus i$;

Wait Until Receive All X_j, $j \in [1:N] \setminus i$;

Return$(Majority(X_1, \cdots, X_N))$;

□

In the above algorithm, * is a concatenation operation of strings, e.g., 000*100 = 000100; and = is an equality evaluation:

$$(X = Y) := \begin{cases} TRUE(1) & \text{if X equals Y} \\ FALSE(0) & \text{otherwise} \end{cases}$$

Some padding may be needed if the length of the local result is not an exact multiple of N. The following example demonstrates a rough comparison of the two algorithms.

Example 4.2 *Comparison of the voting algorithms*

Let $X_1 = X_2 = X_3 = X_4 = 00000$ and $X_5 = 10000$, *Algorithm 4.1* requires 1 round of communication, and transmits a total of 25 bits. On the other hand, with *Algorithm 4.2*, all P_i's ($i = 1, \cdots, 5$) broadcast 0, and $X = 00000$. Thus $(F_1, \cdots, F_5) = 11110$, so $Majority(F_1, \cdots, F_5) = 1$, and X is the majority voting result. In this case, 2 rounds of communication are used, and 10 bits (5 bits for X and 5 bits for F) are transmitted.

If $X_5 = 00001$, and all the other X_i's remain same, then with *Algorithm 4.2*, $X = 00001$, which results in $(F_1, \cdots, F_5) = 00001$. Thus $Majority(F_1, \cdots, F_5) = 0$, and the *Else* part of the algorithm is executed. Finally, the majority voting result is obtained by voting on all the X_i's, i.e., $X = Majority(X_1, \cdots, X_5) = 00000$. Now 3 rounds of communication are needed, and in total, 30 bits (25 bits for the X_i's and 5 bits for F) are transmitted. □

From the above example, it can be observed that

1. *Algorithm 4.1* always requires only 1 round of communication, and *Algorithm 4.2* requires 2 or 3 rounds of communication;

2. The *Else* part of *Algorithm 4.2* is actually *Algorithm 4.1*;

3. The communication complexity of *Algorithm 4.1* is always Nm, but the communication complexity of *Algorithm 4.2* may be $m + N$ or $Nm + N$, depending on the X_i's;

4. In *Algorithm 4.2*, by broadcasting and voting on the *voting flags*, i.e., F_i's, the chance for getting a *false* voting result is eliminated.

If the *Else* part of *Algorithm 4.2*, i.e., *Algorithm 4.1*, is not executed too often, then the communication complexity can be reduced from Nm to $m+N$, and in most cases, $m \gg N$, thus $m + N \approx m$. So the key idea to reduce the communication complexity is to reduce the probability of executing *Algorithm 4.1*. In most computing environments, each module has a low computational error probability p, thus with high probability either *(1)* all modules get the same result or *(2)* only few of them get different results from the rest. In case *(1)*, *Algorithm 4.2* has low communication complexity, but in case *(2)*, *Algorithm 4.1* is actually used and the communication complexity is high (i.e., $Nm + N$). If we can detect and correct these few inconsistent results, then the *Else* part of the *Algorithm 4.2* does not need to be executed, and so the communication complexity can still be low. This detecting and correcting capability can be achieved by using error-correcting codes.

4.3 The Solution Based on Error-Correcting Codes

Error-correcting codes (*ECC*) can be used in the voting problem to reduce the communication complexity. The basic idea is that instead of broadcasting its own computation result X_i directly, the ith module, P_i, first encodes its result X_i into a codeword Y_i of some code, and then broadcasts one *symbol* of the codeword to all the other modules. After receiving all the other *symbols* of the codeword, it reassembles them into a vector. If all the modules have the same result, i.e., all the X_i's are equal, then the received vector is the codeword of the result, thus it can be decoded to retrieve the result. If a majority result exists, i.e., $Majority(X_1, \cdots, X_N) \neq \phi$, and there are t ($t \leq \lfloor \frac{N}{2} \rfloor$) modules whose results are different from the *majority result* X, then the *symbols* from all these modules can be regarded as error symbols with respect to the *majority result*. As long as the code is designed to correct up to t errors, these error symbols can be corrected to get the codeword corresponding to the majority result, thus *Algorithm 4.1* does not need to be executed. When the code length is less than Nm, the communication complexity is reduced compared to *Algorithm 4.1*. On the other hand, if only *error-detecting codes* are used, once errors are detected, *Algorithm*

4.1 still needs to be executed, and thus increases the whole communication complexity of the voting. Thus error-correcting codes are preferable to error-detecting codes for voting. By properly choosing the error-correcting codes, the communication complexity can *always* be lowered below that of *Algorithm 4.1*.

But it is possible that no *majority result* exists, i.e., $Majority(X_1, \cdots, X_N) = \phi$, yet the vector that each module gets can still be decoded. As in the example above, introducing voting flags can avoid this *false* result.

4.3.1 A Voting Algorithm with ECC

With a properly designed error-correcting code, which can detect up to d *error symbols* and correct up to t *error symbols* ($0 \leq t \leq d$), a complete voting algorithm using this code is as follows:

Algorithm 4.3 *ECC Voting*

For Module P_i, $i \in [1:N]$:

    $Y_i :=$ Encode(X_i), partition Y_i into N symbols: $Y_i[1:N]$;

    Broadcast$(Y_i[i])$;

    Wait Until Receive All $Y_j[j], j \in [1:N] \setminus i$;

    $Y := Y_1[1] * \cdots * Y_N[N]$;

    If Y is undecodable

        Execute *Algorithm 4.1* ;

    Else

        $X :=$ Decode(Y);

        $F_i := (X = X_i)$;

        Broadcast(F_i);

        If $Majority(F_1, \cdots, F_N) = TRUE(1)$

            Return(X);

        Else

            Execute *Algorithm 4.1* ;

□

Notice that to execute *Algorithm 4.1*, each module P_i does not need to send its whole result X_i. It only needs to send additional $N-(d+t)-1$ *symbols* of its codeword Y_i. Since the code is designed to detect up to d *and* correct up to t *symbols*, it can correct up to $d+t$ *erasures*, thus the unsent $d+t$ symbols of Y_i can be regarded as *erasures* and recovered; hence the original X_i can be decoded from Y_i.

The flow chart of the algorithm given in Figure 4.1 shows the algorithm more clearly. and the following example shows how the algorithm works.

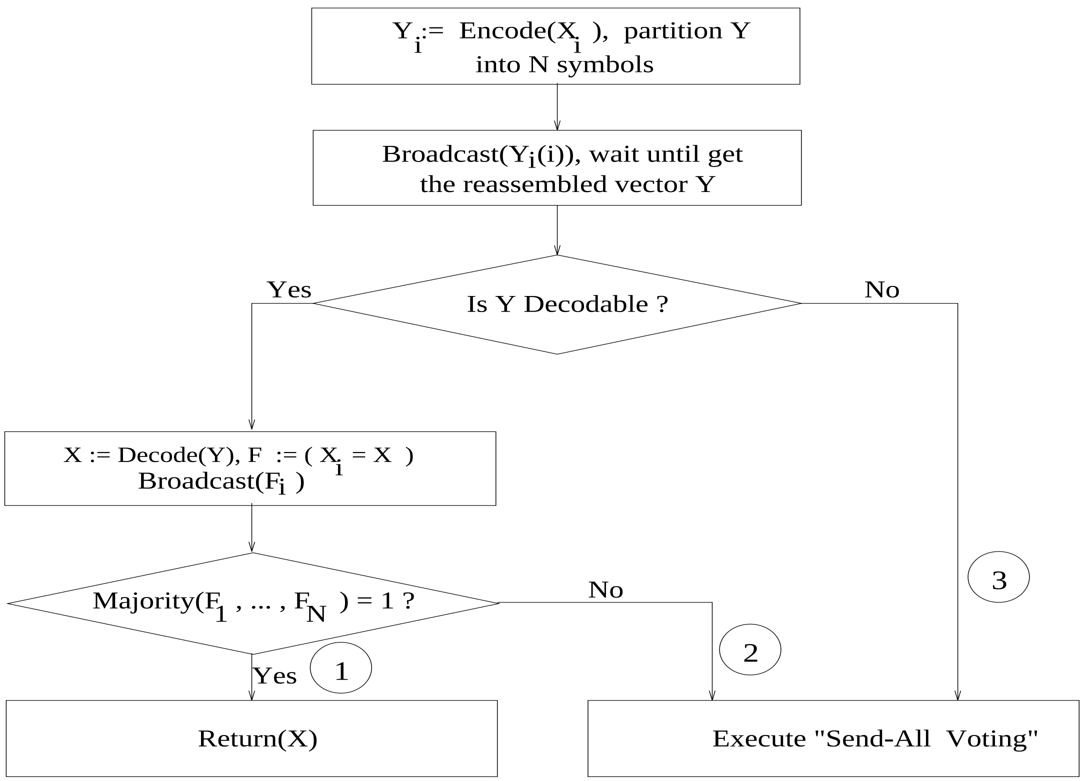


Figure 4.1: Flow chart of *Algorithm 4.3*

Example 4.3 *Voting Algorithm 4.3*

There are 5 modules in an NMR system, and the task result has 6 bits, i.e., $N = 5$ and $m = 6$. Here the EVENODD code[4] is used which divides 6-bit information into 3 *symbols* and encode information *symbols* into a 5-*symbol* codeword. This code can correct 1 error *symbol*, i.e., $d = t = 1$.

Now if $X_i = 000000$, $i = 1, 2, 3, 4$, and $X_5 = 000011$, then with the EVENODD code, $Y_i = 0000000000$, $i = 1, 2, 3, 4$, and $Y_5 = 0000111101$. After each module broadcasts 1 *symbol* (i.e., 2 bits) of its codeword, the reassembled vector is Y=0000000001. Since Y has only 1 error symbol, it can be decoded into X=000000. From the flow chart of the algorithm, we can see that $F_i = 1$, $i = 1, 2, 3, 4$, and $F_5 = 0$, thus $Majority(F_1, \cdots, F_5) = 1$, so X=000000 is the majority voting result.

In this case, there are 2 rounds of communication, and the communication complexity is 15 bits. As a comparison, *Algorithm 4.1* needs 1 round of communication, and its communication complexity is 30 bits; *Algorithm 4.2* needs 3 rounds of communication, and its communication complexity is 35 bits. In this example, the EVENODD code is used, but the specific code itself does not affect the communication complexity as long as it has same properties as the EVENODD code, namely, it is an *MDS* code with $d = t = 1$. □

As can be seen in the flow chart of the algorithm, introducing voting on the F_i's ensures that it is impossible to reach a *false* voting result, and going to the *Send-All Voting* in the worst case guarantees that the modules reach the majority result if it exists. Thus the algorithm gives a correct majority voting result. A rigorous correctness proof of the algorithm follows.

4.3.2 Correctness of the Algorithm

Theorem 4.1 Algorithm 4.3 *gives* $Majority(X_1, \cdots, X_N)$ *for a given set of local computational results* X_i*'s* $(i = 1, \cdots, N)$.

Proof: From the flow chart of the algorithm, it is easy to see that the algorithm terminates in the following two cases:

1. Executing the *Send-All Voting algorithm*: the correct majority voting result is certainly reached;

2. Returning an X: in this case, since $Majority(F_1, \cdots, F_N) = TRUE(1)$, i.e., majority of the $X's$ are equal to X, the correct majority voting result is also reached: X is the majority result. □

To see how the algorithm works for various instances of the local results X_i's $(i = 1, \cdots, N)$, we give two stronger observations about the algorithm, which also help to analyze the communication complexity of the algorithm.

Observation 4.1 *If* $Majority(X_1, \cdots, X_N) = \phi$*, then* Algorithm 4.3 *outputs* ϕ*, i.e.,* Algorithm 4.3 *never gives a* false *voting result.*

Proof: It is easy to see from the flow chart that after the first round of communication, each module gets the same *voting vector* Y. According to the decodability of Y, there are two cases:

1. If Y is undecodable, then the *Send-All Voting* algorithm is executed, and the output will be ϕ;

2. If Y is decodable, the decoded result X now can be used as a reference result. But since there does not exist a majority voting result, a majority of the X_i's are not equal to the X, i.e., $Majority(F_1, \cdots, F_N) = FALSE$, so the *Send-All Voting* algorithm is executed, and the output again will be ϕ. □

Observation 4.2 *If* $Majority(X_1, \cdots, X_N) = X(\neq \phi)$, *then* Algorithm 4.3*'s output is exactly X, i.e.,* Algorithm 3 *will not* miss *the majority voting result.*

Proof: Suppose there are e modules whose local results are different from the majority result X, then $e \leq \lfloor \frac{N-1}{2} \rfloor$.

1. If $e \leq t$, then there are e *error symbols* in the *voting vector* Y with respect to the corresponding codeword of the majority result X, so Y can be correctly decoded into X. A majority of the X_i's are equal to X, i.e., a majority of the F_i's are 1, hence the correct majority result X is outputted.

2. If $e > t$, then Y is either undecodable or incorrectly decoded into another X', where $X' \neq$ X. In either case, the *Send-All voting* algorithm is executed and the correct majority result X is reached. □

4.3.3 Proper Code Design

In order to reduce the communication complexity, we need an error-correcting code that can be used in practice for *Algorithm 4.3.* Consider a block code of length *M*. Because of the symmetry among the *N* modules, *M* needs to be a multiple of *N*, i.e., each codeword consists of *N symbols*, and each *symbol* has *k* bits, thus $M = Nk$. If the *minimum distance* of the code is d_{min}, then $d_{min} \geq (d+t)k+1$, where $0 \leq t \leq d \leq \lfloor \frac{N}{2} \rfloor$, since the code should be able to *detect* up to d error symbols and *correct* up to t error symbols[19]. Recall that the final voting result has m bits, the code to design is a $(Nk, m, (d+t)k+1)$ block code.

To get the smallest value for k, by the *Singleton Bound* in coding theory[19],

$$Nk - m + 1 \geq (d+t)k + 1 \tag{4.1}$$

we get

$$k \geq \frac{m}{N-(d+t)} \tag{4.2}$$

Equality holds for all *MDS Codes*[19].

So, given a specified (d, t), the smallest value for k is $\lceil \frac{m}{N-(d+t)} \rceil$. If $\frac{m}{N-(d+t)}$ is an integer, all *MDS Codes* can achieve this lower bound of k. One class of commonly used MDS codes for arbitrary distances is the Reed-Solomon code[19]. If $\frac{m}{N-(d+t)}$ is not an integer, then *any* $(Nk, m, (d+t)k+1)$ block code can be used, where $k = \lceil \frac{m}{N-(d+t)} \rceil$. One example of this is the BCH code, which can also have arbitrary distances[19]. The exact parameters (k, d, t) can be achieved by *shortening* (i.e., setting some information symbols to zeros) or *puncturing* (deleting some parity symbols) proper codes[19].

Notice that $0 \le t \le d \le \frac{N-1}{2}$, thus $\frac{m}{N} \le k \le m$. In most applications, $N \ll m$, thus the N bits of F_i's can be neglected, and k is approximately the number of the bits that each module needs to send to get final voting result, so the communication complexity of *Algorithm 4.3* is always lower than that of *Algorithm 4.1*.

In this chapter, only the communication complexity of voting is considered, since in many systems, computations for encoding and decoding on individual nodes are much faster than reliable communications among these nodes; the latter requires rather complicated data management in different communication stacks and retransmission of packets between distributed nodes when packet loss occurs. However, in real applications, design of proper codes should also make the encoding and decoding of the codes as computationally efficient as possible. When the distances of codes are relatively small, as in most applications when the error probability p is relatively low, more computationally-efficient MDS codes should be used, such as the X-Code or the B-Code discussed in the previous chapters.

4.4 Communication Complexity Analysis

4.4.1 Main Results

From the flow chart of *Algorithm 4.3*, we can see if the algorithm terminates in *Branch 1*, i.e., the algorithm gets a majority result, then the communication complexity is $N(k+1)$; if it terminates in *Branch 2*, then the communication complexity is $N(m+1)$; finally if the algorithm terminates in *Branch 3*, the communication complexity is Nm. Thus the *worst case* communication complexity C_w is $N(m+1)$. When $m \gg 1$, $C_w \approx Nm$.

Let C_a denote the *average case* communication complexity of *Algorithm 4.3*, and define the *average reduction factor* α as the ratio of C_a over the communication complexity of the

Send-All Voting algorithm, i.e. $\alpha = \frac{C_a}{Nm}$. The following theorem gives the relation between α and the parameters of both an NMR system and the corresponding code:

Theorem 4.2 *For an NMR system with N modules, each of which executes an identical task whose result is m-bits long. Assume each module has computational error probability p, independent of other modules' activities. If* Algorithm 4.3 *uses an ECC which can detect up to d and correct up to t error symbols, and $m \gg N > 1$, then the following relation holds for the average reduction factor of* Algorithm 4.3*:*

$$\alpha < \frac{P_1}{N-(d+t)} + (1-P_1) + \frac{1}{m} \tag{4.3}$$

where

$$P_1 = \sum_{i=0}^{t} \binom{N}{i} p^i (1-p)^{N-i} \tag{4.4}$$

Proof: To get the average case communication complexity C_a of *Algorithm 4.3*, we need to analyze the probability P_i of the algorithm terminating in the *Branch* i, $i = 1, 2, 3$. First assume that if a module has an erroneous result X_i, then it contributes an error symbol to the voting vector Y. From the proof of *Observation 2*, if the algorithm terminates in *Branch 1*, then at most t modules have computational errors; the probability of this event is exactly P_1. The event that the algorithm reaches *Branch 2* corresponds to the *decoder error* event of a code with *minimum distance* of $d+t+1$, thus[19]

$$P_2 = \sum_{i=d+t+1}^{N} A_i \sum_{k=0}^{\lfloor \frac{d+t}{2} \rfloor} P_{ik} \tag{4.5}$$

where $\{A_i\}$ is the *weight distribution* of the code being used, and P_{ik} is the probability that a received vector Y is exactly Hamming distance k away from a *weight-i* (binary) codeword of the code. More precisely,

$$P_{ik} = \sum_{j=0}^{k} \binom{i}{k-j} \binom{N-i}{j} p^{i-k+2j} (1-p)^{N-i+k-2j} \tag{4.6}$$

If the weight distribution of the code is unknown, P_2 can be approximately bounded by

$$P_2 \leq 1 - \sum_{i=0}^{\lfloor \frac{d+t}{2} \rfloor} \binom{N}{i} p^i (1-p)^{N-i} \tag{4.7}$$

since the second term in the right side of the inequality above is just the probability of event that *correctable* errors occur. Finally P_3 is the probability of the *decoder failure* event,

$$P_3 = 1 - P_1 - P_2 \tag{4.8}$$

Now notice that a module with an erroneous result can also contribute a correct symbol to the voting vector, so the average case communication complexity is

$$C_a \leq P_1 N(k+1) + P_2 N(m+1) + P_3 N m \tag{4.9}$$

and the average reduction factor is

$$\alpha \leq \frac{k}{m} P_1 + (1 - P_1) + \frac{P_1 + P_2}{m} \tag{4.10}$$

By noticing that $k = \lceil \frac{m}{N-(d+t)} \rceil$, and $P_1 + P_2 < 1$, we get the result of Eq. (4.3). □

Remarks on the theorem : From Eq. (4.3), we can see the relation between the average reduction factor α and each branch of *Algorithm 4.3.* The first term relates to the first branch, whose reduction factor is $\frac{k}{m}$, or $\frac{1}{N-(d+t)}$ when m is large enough relative to N that the round-off error can be neglected; P_1 is the probability of that branch. One would expect this term to be the dominant one for α, since with a properly designed code tuned to the system, the algorithm is supposed to terminate at *Branch 1* in most cases. The second term is simply the probability that the algorithm terminates at either *Branch 2* or *Branch 3*; in this case the reduction factor is 1 (i.e., there is no communication reduction since all the local results are transmitted), without considering the 1 bit for F_i's in *Branch 2.* The last term is due to the 1 bit for voting on F_i's. When the local result size is large enough, i.e., $m \gg 1$, this 1 bit can be neglected in our model. Thus in most applications, the assumption that $m \gg 1$ is quite reasonable, and the result of the theorem can be simplified to

$$\alpha \approx \frac{P_1}{N-(d+t)} + (1-P_1) \tag{4.11}$$

From the above theorem and its proof, it can be seen that for a given *NMR* system (i.e., N and p), P_1 is only a function of t. Once t is chosen, it is easy to see from Eq. (4.3) or Eq. (4.11) that α decreases monotonically as d decreases. Recall that $0 \le t \le d$, thus for a fixed t, setting $d = t$ minimizes α when $m \gg 1$. Even though it is not straightforward to get a closed form solution for the value of t which minimizes α, it is almost trivial to get such an optimal t by numerical calculation.

Figure 4.2 shows relationship between α and (t, p, N). Figure 4.2(a) and Figure 4.2(b) show how α (using Eq. (4.11)) changes with t for some fixed (N, p) and $d = t$. It is easy to see that for small p and reasonable N, a small t (e.g., $t \le 2$, for $N \le 51$ with $p = 0.01$) can achieve minimal α. These results show that for a highly reliable *NMR* system (e.g., $p \le 0.01$), adding a small amount of redundancy to the local results and applying error-correcting codes to them can drastically reduce the communication complexity of the majority voting. Since the majority result is m bits long, and each module gets an identical result after voting, the communication complexity of the voting problem is at least m bits, thus $\alpha \ge \frac{1}{N}$, i.e., $\frac{1}{N}$ is a lower bound of α. Figure 4.2c shows how close *Algorithm 4.3* comes to achieving the theoretical lower bound of α.

(a) α vs. t for different p with fixed N = 31 (b) α vs. t for different N with fixed p = 0.01 (c) Minimum α vs. N

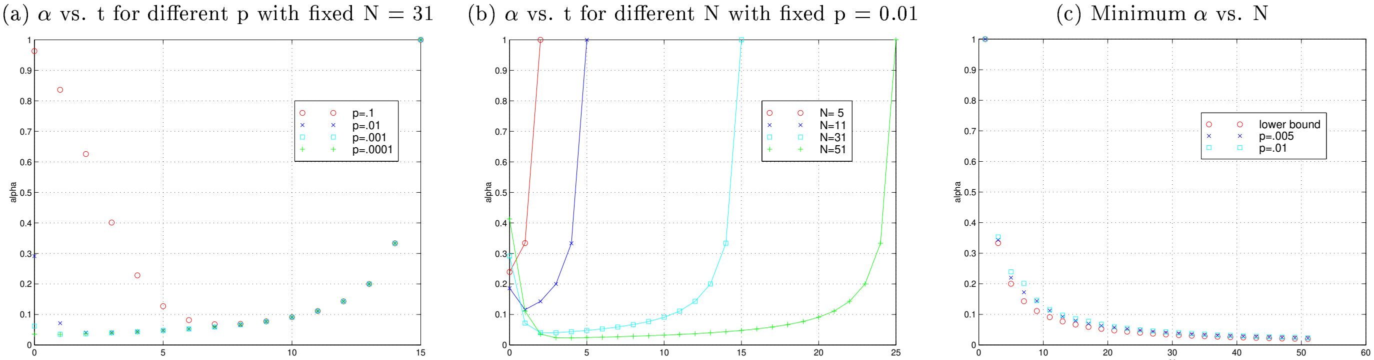


Figure 4.2: Relations between α and (t, p, N)

4.4.2 More Observations

From the above results, we can see that the communication complexity of *Algorithm 4.3* is determined by the code design parameters (d, t). In an *NMR* system with N modules, we only need to consider the case where at most $\lfloor \frac{N}{2} \rfloor$ modules have local results different from the majority result, thus the only constraint of (d,t) is $0 \le t \le d \le \lfloor \frac{N}{2} \rfloor$. For some specific values of (d,t), the algorithm reduces to the following cases:

1. When $d = t = \frac{N-1}{2}$, i.e., a *repetition* code is used, the algorithm becomes *Algorithm 4.1.* Since a repetition code is always the worst code in terms of redundancy, it should always be avoided for reducing the communication complexity of voting. On the other hand, when d=t=0: the algorithm becomes *Algorithm 4.2*, and from Figure 4.2, we can see that for small enough p and reasonable N, e.g., $p = 10^{-5}$ and $N = 31$, *Algorithm 4.2* actually is the best solution of the majority voting problem in terms of the communication complexity. In addition, *Algorithm 4.2* has low computational complexity since it does not need any complex encoding or decoding operations. Thus the *ECC* voting algorithm is a generalized voting algorithm, and its communication complexity is determined by the code chosen.

2. $t = 0$, then the code only has detecting capability, but if $m \gg N$, then the analysis above shows increasing d actually makes α increase. Thus, when $m \gg N$, it is not good to add redundancy to the local results just for error detecting, i.e., using only *EDCs* (*error detecting codes*) does not help to reduce the communication complexity of voting. The scheme proposed in [21] ($d = \lfloor \frac{N}{2} \rfloor$) falls under this category.

3. $d = \lfloor \frac{N}{2} \rfloor$: as above, it is not good in general to have $d > t$ in terms of α, since increasing d will increase α for fixed t. But in this case, *Algorithm 4.3* has a special property: *Branch 2* of the algorithm can declare there is *no* majority result directly, *without* executing the *Send-All Voting* algorithm, because the code now can detect up to $\lfloor \frac{N}{2} \rfloor$ errors. So if there was a majority result, then Y (refer to Figure 4.1) can have at most $\lfloor \frac{N}{2} \rfloor$ erroneous modules, and since Y is decodable, the majority of the local results should agree with the decoded result X, i.e., $Majority(F_1, \cdots, F_N) = TRUE(1)$. This differs from the actual $Majority(F_1, \cdots, F_N)$, so there is *no* majority result. By setting d to $\lfloor \frac{N}{2} \rfloor$, *Algorithm 4.3* always has *2 rounds* of communication and the *worst case* communication complexity is thus Nm instead of $N(m+1)$. This achieves the lower bound of the *worst case* communication

complexity of the distributed majority voting problem[21].

4.5 Experimental Results

In this section, we show some experimental results of the three voting algorithms discussed above. The experiments are performed on a cluster of Intel Pentium/Linux-2.0.7 nodes connected via a 100 Mbps Ethernet. Reliable communication is implemented by a simple improved UDP scheme: whenever there is a packet loss, the voting operation is considered a failure and is redone from the beginning. By choosing suitable packet size, there is virtually no packet loss using UDP.

To examine the real performance of the above three voting algorithms, N nodes vote on a result of length m using all three voting algorithms. For the *ECC Voting* algorithm, an *EVENODD Code* is used, which corrects 1 *error symbol*, i.e., $d = t = 1$. Random errors are added to local computing results with a preassigned error probability p, independent of results at other nodes in the NMR system. The performance is evaluated by two parameters for each algorithm: the total time to complete the voting operation T and the communication time for the voting operation C. The maximum T and C of the whole NMR system are chosen from among all the local T's and C's,, since if the voting operation is considered a collective operation, the system's performance is determined by the *worst* local performance in the system. For each set of NMR system parameters (N nodes and error probability p), each voting operation is done 200 times, and random computation errors in each run are independent of those in other runs. The arithmetic average of C's and T's are regarded as the performance parameters for the tested NMR system.

Experimental results are shown in Figures 4.3 through 4.5. Figure 4.3 compares the experimental *average reduction factors* of the voting algorithms with the theoretical results in the previous section, for an NMR system of 5 nodes. Figure 4.4 shows the performance (T and C) of the voting algorithms. Detailed communication patterns of the voting algorithms are shown in Figure 4.5 to provide some deeper insight into the voting algorithms.

Figure 4.3(a) and Figure 4.3(b) show the experimental *average reduction factors* of the voting communication time (C) for the *Simple Send-Part Voting* algorithm and the *ECC Voting* algorithm. Figure 4.3(a) and Figure 4.3(b) also show the theoretical average reduction factors of the Algorithm 4.2 and 4.3 as computed from Eq. (4.11). Notice that the average communication time reduction factors α of both *Algorithm 4.2* and *Algorithm 4.3* are below 1 for all the result sizes greater than 1 Kbyte, Also notice that as the computational result size increases, the reduction factor approaches the theoretical bound.

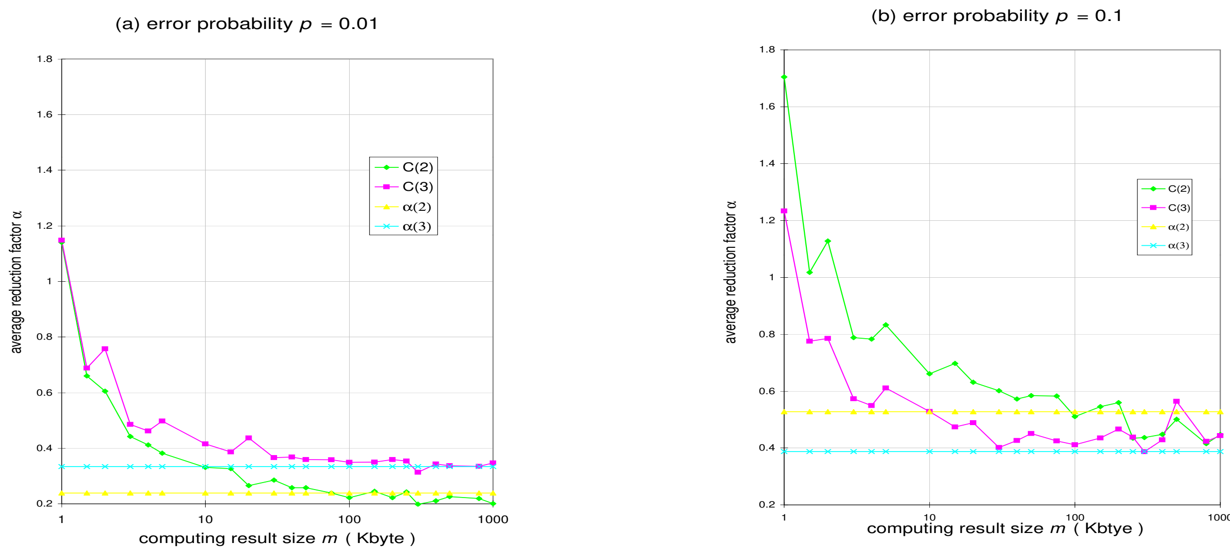


Figure 4.3: Average reduction factors

(C(i) is the experimental average reduction factor of communication time for voting using *algorithm 4.i*, and α(i) is the theoretical bound of the average communication reduction factor using *algorithm 4.i*, $i = 2, 3$)

Figure 4.4 shows the performance of each voting algorithm applied to an NMR system of 5 nodes. Figure 4.4(a)(b) show the *total* voting time T and Figure 4.4(c)(d) show the *communication* time C for voting. The only different parameter of the NMR systems related to Figures (a) and (b) (symmetrically (c) and (d)) is the error probability p: $p = 0.1$ in Figures (a) and (c), while $p = 0.01$ in Figures (b) and (d). It is easy to see from the figures that for *Algorithm 4.1 (Send-All Voting)*, T and C are the same, since there is no local computation other than communication. Figures 4.4(a)(b) show that Algorithms 4.2 (*Simple Send-Part Voting*) and 4.3 (*ECC Voting*) perform better than the Algorithm 4.1 (*Send-All Voting*) in terms of the total voting time T. On the other hand, Figures 4.4(c)(d) show that, in terms of the communication complexity C, the *ECC Voting* algorithm is better than the *Simple Send-Part Voting* algorithm when the error probability is relatively large (Figure 4.4(c)) and worse than the *Simple Send-Part Voting* algorithm when the error probability is relatively small (Figure 4.4(d)), which is consistent with the analysis results in the previous section.

In the analysis in the previous section, the size m of local computing result does not show up as a variable in the average reduction factor function α, since the communication complexity is considered to be only proportional to the size of the messages that need to be broadcast. But practically, communication time is *not* proportional to message size, since startup time of communication also needs to be included. More specifically, in the Ethernet environment, since the maximum packet size of each physical send (broadcast) operation is limited by the physical ethernet, communication completion time becomes a more complicated function of the message size. Thus from the experimental results, it can be seen that for a computing result of small size, e.g., 1 Kbyte, the *Send-All Voting* algorithm actually performs best in terms of both C and T, since the startup time dominates the performance of communication. Also, the communication time for broadcasting the 1-bit *voting flags* cannot be neglected as analyzed in the previous section, particularly for a computational result of small size. This can also be seen from the detailed voting communication time pattern in Figures 4.5(a)(b): *round 2* of the communication is for the 1-bit *voting flag*; even though it finishes in much more shorter time than *round 1*, it is still not negligibly small. This explains the fact that for small size computing results, the average communication time reduction factors of *Algorithm 4.2* and *Algorithm 4.3* are quite different from their theoretical bound.

Further examination of the detailed communication time pattern of voting provides a deeper insight into *Algorithm 4.3*. From Figures 4.5(c)(d), it is easy to see that in the first round of communication, *Algorithm 4.2* needs *less* time than *Algorithm 4.3*, since the size of the message to be broadcast is smaller for *Algorithm 4.2*. Also, the first round of communication time does not vary with the error probability p for the either algorithm. The real difference between the two algorithms lies in the third round of communication. From Figure 4.5(c), this time is small for the both algorithms since the error probability p is small (0.01). But as the error probability p increases to 0.1, as shown in Figure 4.5(d), for *Algorithm 4.2*, the third round time increases to more than the first round time, since it has no *error-correcting* capability. Once a full message needs to be broadcast, the message size is much bigger than that in the first round. On the other hand, though the communication time for the third round also increases for *Algorithm 4.3*, it is still much smaller than in the first round; this comes from the error-correcting codes that *Algorithm 4.3* uses, which can correct the most frequently occurring error pattern: errors at a single computing node. Thus, even though the computation error probability is high, in most cases, the most expensive third round of communication can be avoided. Thus *Algorithm 4.3* performs better (in terms of communication complexity or time) than *Algorithm 4.2* in high error probability systems, just as the analysis in the previous section predicted.

4.6 Summary

We have proposed a deterministic distributed voting algorithm using error-correcting codes to reduce the communication complexity of the voting problem in *NMR* systems. We have also given a detailed theoretical analysis of the algorithm. By choosing the design parameters (d, t) of the error-correcting code, the algorithm can achieve a low communication complexity, which is quite close to its theoretical lower bound. We have also implemented the voting algorithm over a network of workstations, and the experimental performance results match the theoretical analysis well. The algorithm proposed here needs 2 or 3 rounds of communication. It is left as an open problem whether there is an algorithm for the distributed majority voting problem with *average case* communication complexity less than Nm using *only* 1 round of communication.

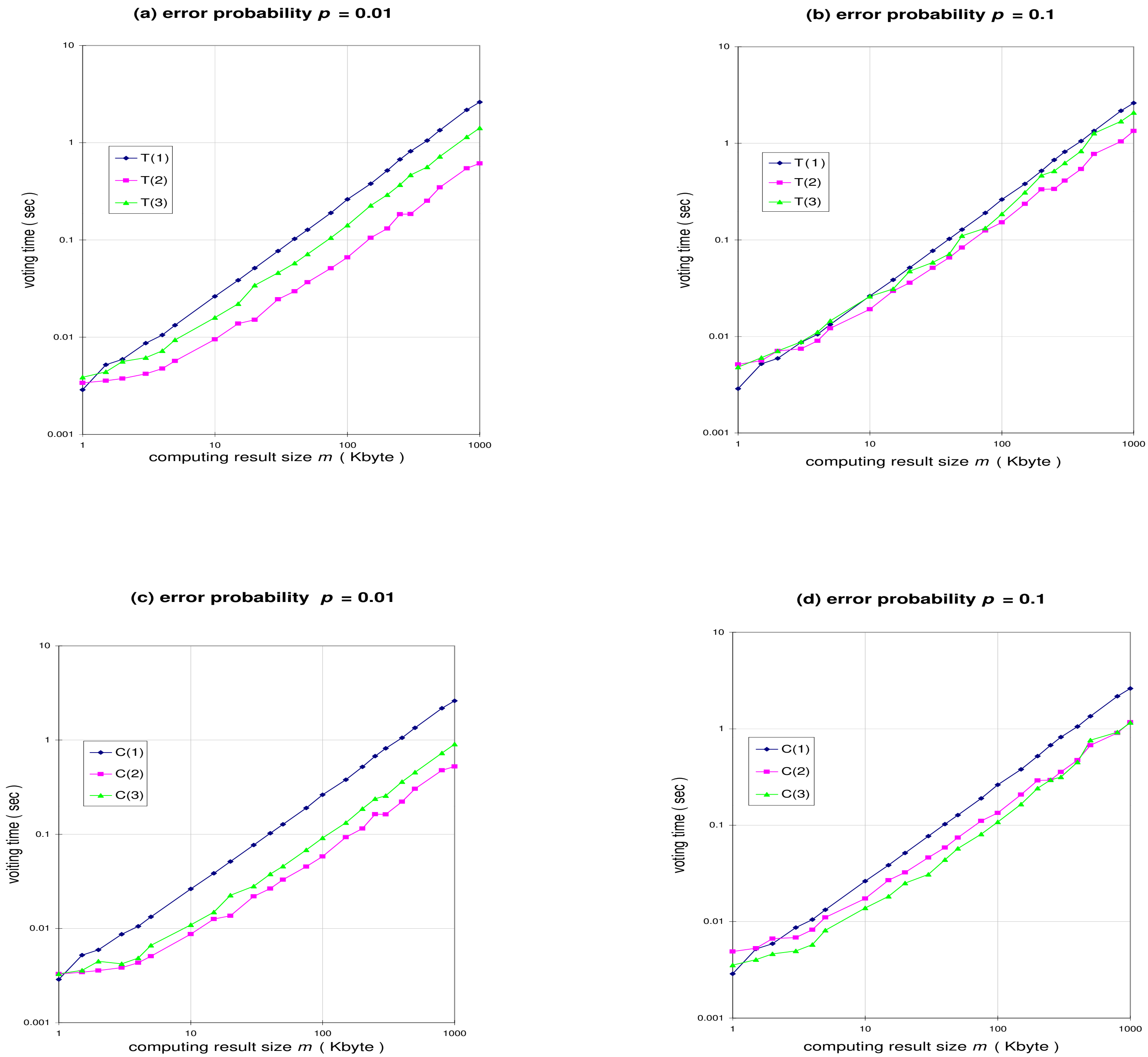


Figure 4.4: Experimental voting performance of 5-node NMR system
(T(i) and C(i) are the total and communication time for voting using *algorithm 4.i* respectively, $i = 1, 2, 3$)

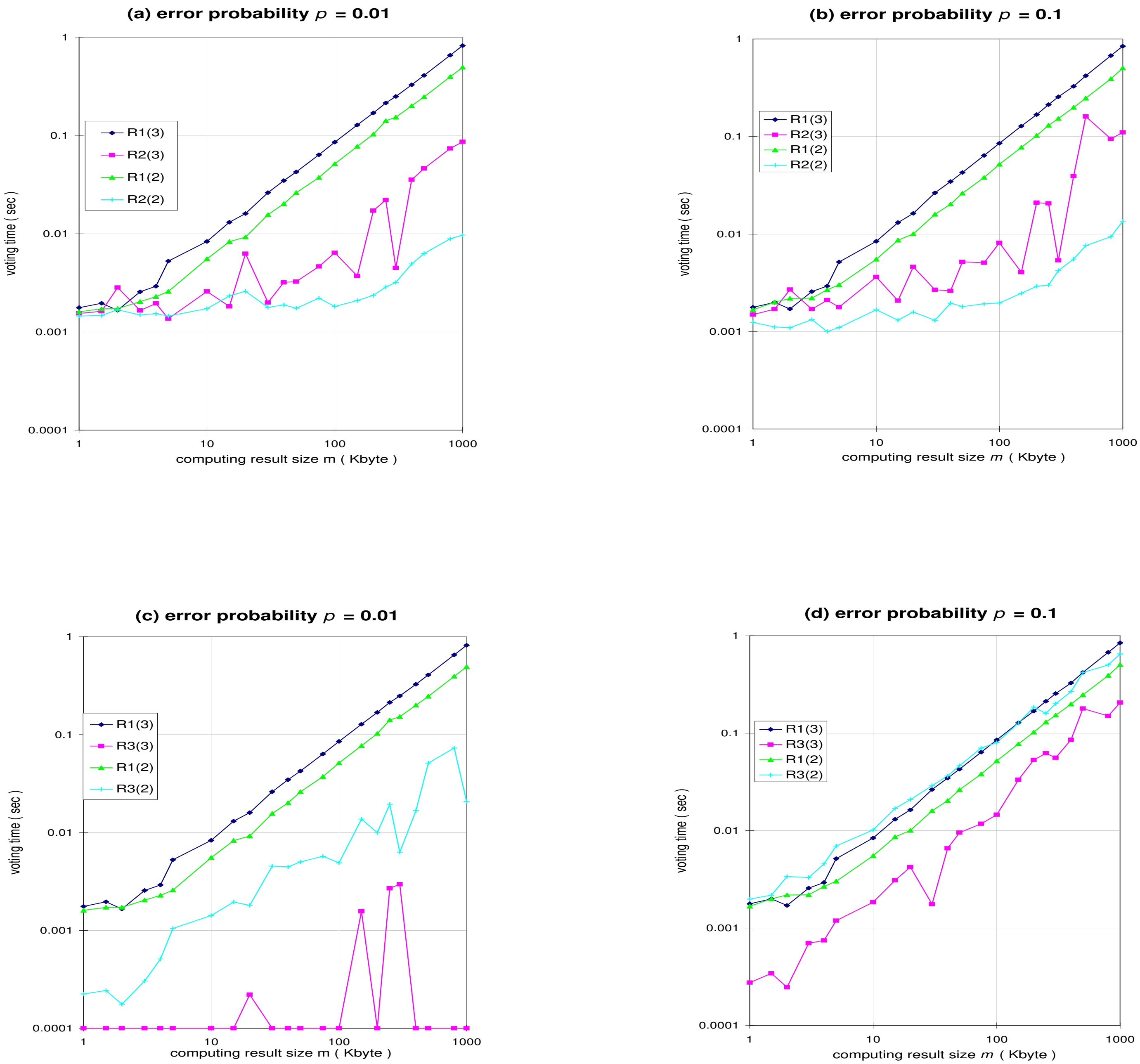


Figure 4.5: Detailed communication time pattern of voting

(R$i(k)$ is the communication time in *round* i using the voting *algorithm 4.k*, $i = 1, 2, 3$, and $k = 2, 3$)

Chapter 5 Improving the Performance of Data Servers

5.1 Introduction

It has become common to use a cluster of distributed computing nodes in server systems, such as image servers, video servers, multimedia servers, and web servers, all of which can be regarded as specific kinds of the more general concept of data servers. Distributed server systems can improve both data *reliability* (or availability) and *performance* (or efficiency, i.e., data throughput) of the system. Much research, such as the well known k-out-of-n systems[3], has been done to improve reliability by introducing *data redundancy* or *information dispersity*[27] into systems.

In a real system, although the redundant data enables the system to provide continuous service when certain failures (communication link failures, server node failures) occur, most of the time the system works in a normal mode, i.e., there is no failure in the system; in this case data redundancy can be used to improve the performance of the system. It was first shown in [11] that redundant data can make packet routing more efficient by reducing the mean and variance of the routing delay. Recently, more scalable and efficient reliable multicast schemes have also been proposed based on data redundancy in the messages to be multicast[14].

In this chapter, we propose a method based on error-correcting codes for improving the performance of services in general data server systems. Our system setup is shown in Figure 5.1: a cluster of servers is connected via some reliable communication network. In addition, broadcast is supported over the network, so that a client can broadcast its request for certain data to some or all of the n servers in the system. The data is distributed over the servers in such a way that a client can recover the complete requested data after it gets data from at least k of the n servers and this is true for *any* k servers. Such a distributed data server system is called an (n, k) server system in this chapter. Again, such (n, k) systems can be implemented by using error-correcting codes, particularly MDS array codes.

For a data server system, since the I/O speed of the local disks is much slower than

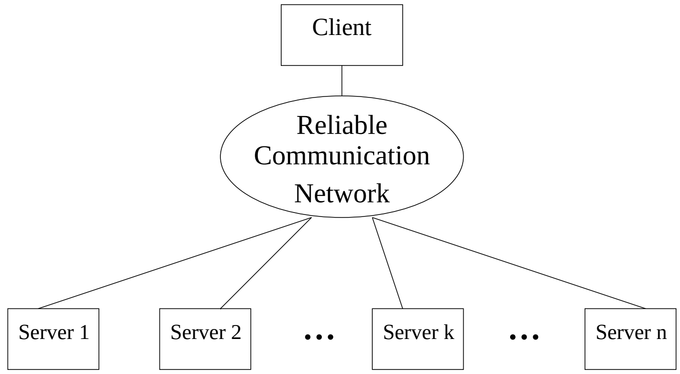


Figure 5.1: An (n, k) server system

the CPU speed of the servers, the whole performance of the system is dominated by the bottleneck of the disks. Distributing data over multiple servers can help overcome this bottleneck and improve the data throughput of the whole system, since the data can be accessed in parallel from multiple servers at the same time. For a server system with n servers, the system performance can be further improved by introducing proper data redundancy into the system, i.e., an (n, k) system should be properly chosen instead of naively distributing the raw data over all the n servers. This was shown in Example 1.5 and Example 1.6 in Chapter 1. What is the *proper* redundancy when the total number of the servers is given? Or how should k be determined when n is given, in order to achieve the best system performance? This is the so-called *data distribution* problem at the server side, which will be discussed in detail later in this chapter. Also, as already discussed in Example 1.7 in Chapter 1, there is another problem called the *data acquisition* at the client side: that once data redundancy is properly distributed among the servers, how should matching read approaches be chosen to optimize the mean service time? This problem will be also explored in this chapter. The main contribution of this chapter is to propose data distribution and acquisition schemes for a given server system that improve the system performance.

This chapter is organized as follows. Section 5.2 first describes a probability model of distributed server systems, then gives analytical results about the service time of a general distributed data server system. In Section 5.3, experimental results are used to create a probability model for service time in a practical disk-based data server system. The data distribution and data acquisition schemes are discussed in more detail in Section 5.4. Section 5.5 concludes the chapter and proposes a future research direction.

5.2 Preliminary Analysis

Before we consider other problems, we first define a server system model we will be using. Then we give some basic analytical results that can be used further to solve the data distribution and data acquisition problems.

5.2.1 System Model

In this chapter, a distributed server system consists of n servers. A client can broadcast its request for certain data to all the servers. All communications among the client and servers are reliable, i.e., there is no packet loss, order change or content corruption. Each server stores a portion of the requested data in such a way that the client can recover its requested data after it receives data from at least k ($k \leq n$) out of the n servers. Define the service time T_i of the server i ($1 \leq i \leq n$) to be the elapsed time from when the client sends its request to the server i to when it receives data from the server i. Notice that T_i does *not* include the time needed at the client side to do any necessary *computations* to recover the final data, since here we assume that the computations are rather simple and thus take much less time than does the data delivery through communication media. We model T_i as a continuous random variable with *probability density function* (pdf) $f_i(t)$[23]. For simplicity of analysis, we assume that all T_is are i.i.d (*independent, identically distributed*) random variables, i.e., $f_i(t) = f(t)$, $1 \leq i \leq n$.

5.2.2 Analysis Results

Let $F_i(t)$ be the *cumulative distribution function* (cdf) of T_i, i.e.[23],

$$F_i(t) = Probability(T_i \leq t) = \int_0^t f_i(x)dx$$

Now let $T(n,k)$ be the elapsed time from when the client broadcasts its data request to the servers to when it receives data from at least k out of the n servers. Then $T(n,k)$ is another random variable and is a simple function of all the T_is:

$$T(n,k) \geq T_i, \quad where \ \|\{i\}\| \geq k$$

In the above equation, $\|S\|$ is the number of the elements in the set S.

Let $f_{(n,k)}(t)$ and $F_{(n,k)}(t)$ be the pdf and cdf of $T(n,k)$ respectively, then it is easy to relate $F_{(n,k)}(t)$ and $f_{(n,k)}(t)$ to $F(t)$ and $f(t)$ [11]:

$$F_{(n,k)}(t) = \sum_{i=k}^{n} \binom{n}{i} F(t)^i [1 - F(t)]^{n-i} \tag{5.1}$$

or [11][32]:

$$f_{(n,k)}(t) = \frac{dF_{(n,k)}(t)}{dt} = k \binom{n}{k} F(t)^{k-1} [1 - F(t)]^{n-k} f(t) \tag{5.2}$$

The *mean* of $T(n,k)$, $E[T(n,k)]$, is a good measurement of the server system's performance. It can be calculated once the $f_{(n,k)}(t)$ is known:

$$E[T(n,k)] = \int_0^{\infty} t f_{(n,k)}(t) dt \tag{5.3}$$

5.2.3 Properties of Mean Service Time

Though it is usually hard to get a clean closed form of $E[T(n,k)]$ for a general pdf $f(t)$, it is still possible to get some of its properties with respect to n and k. Intuitively, for a fixed pdf $f(t)$, a bigger n and/or a smaller k leads to a smaller $E[T(n,k)]$ and this can be proven mathematically.

Before we discuss properties of the $E[T(n,k)]$, we give a lemma which can be used to prove the properties.

Lemma 5.1 *Let two continuous random variables X and Y be defined on $[a,b]$ with cdf's $F_X(t)$ and $F_Y(t)$ respectively. If $F_X(t) \geq F_Y(t)$, for all t, $a \leq t \leq b$, and $F_X \not\equiv F_Y$, i.e., the pdf of X is* left *of that of Y, then $E[X] < E[Y]$.* □

Proof: Notice that $F_X(b) = F_Y(b) = 1$ and $F_X(a) = F_Y(a) = 0$, then

$E[X] - E[Y] = \int_a^b t dF_X(t) - \int_a^b t dF_Y(t) = [tF_X(t)|_a^b - \int_a^b F_X(t) dt] - [tF_Y(t)|_a^b - \int_a^b F_Y(t) dt]$
$= t[F_X(t) - F_Y(t)]|_a^b - \int_a^b [F_X(t) - F_Y(t)] dt = -\int_a^b (F_X(t) - F_Y(t)) dt < 0.$ □

It has been shown in [27] that

Lemma 5.2 *For a random variable T with a fixed pdf $f(t)$, the following inequalities hold for $1 \leq k \leq n$ and for $0 < F(t) < 1$:*

1. *$F_{(n,k)}(t) < F_{(n+m,k)}(t)$, for $m \geq 1$;*

2. $F_{(n,k)}(t) > F_{(n,k+m)}(t)$, *for* $m \geq 1$*;*

3. $F_{(n,k)}(t) > F_{(n+m,k+m)}(t)$, *for* $m \geq 1$*;*

4. $F_{(i,j)}(t) \leq F_{(n,k)}(t)$, *if* $n \geq i$ *and* $k \leq j$, *equality holds only when* $n = i$ *and* $k = j$*;*

5. $F_{(i,j)}(t) > F_{(n,k)}(t)$, *if* $n \geq i$, $k > j$ *and* $n - k \leq i - j$.

□

Using the two lemmas, it is straight forward to get the following properties of the mean of the service time:

Theorem 5.1 *For a random variable* T *with a fixed pdf* $f(t)$*, the following inequalities hold for* $1 \leq k \leq n$*:*

1. $E[T(n,k)] > E[T(n+m,k)]$, *for* $m \geq 1$*;*

2. $E[T(n,k)] < E[T(n,k+m)]$, *for* $m \geq 1$*;*

3. $E[T(n,k)] < E[T(n+m,k+m)]$, *for* $m \geq 1$*;*

4. $E[T(i,j)] \geq E[T(n,k)]$, *if* $n \geq i$ *and* $k \leq j$, *equality holds only when* $n = i$ *and* $k = j$*;*

5. $E[T(i,j)] < E[T(n,k)]$, *if* $n \geq i$, $k > j$ *and* $n - k \leq i - j$.

□

We will use these properties as guidelines in Section 5.4 for the data distribution. One would hope that the *variances* of random variables also had the similar properties. Unfortunately, however, the above properties do *not* hold for the variances. We will show one example about the variances and one more property of the $E[T(n,k)]$.

Lemma 5.3 *Let two continuous random variables* X *and* Y *be defined on* $[a,b]$ *with pdf's* $f(t)$ *and* $g(t)$*, respectively. If* $f(t) = g(a+b-t)$*, for all t,* $a \leq t \leq b$*, i.e.,* $f(t)$ *is the* reflection *of* $g(t)$ *about the line* $t = \frac{a+b}{2}$*, then* $E(X) = a + b - E(Y)$*, and* $Var(X) = Var(Y)$*, where* $Var(X)$ *and* $Var(Y)$ *are the* variances *of* X *and* Y*.* □

Proof: Straight forward, omitted.

Lemma 5.4 *If the pdf $f(t)$ of a random variable T defined on $[a, b]$ is* symmetric *or* self-reflective *about the line $t = \frac{a+b}{2}$, i.e., $f(t) = f(a+b-t)$, then $f_{(n,k)}(t) = f_{(n,n+1-k)}(a+b-t)$.* □

Proof: First, it is easy to show that if $f(t) = f(a + b - t)$, then

$$F(t) = 1 - F(a + b - t) \tag{5.4}$$

Using Eq.(5.2), Eq.(5.4) and the identity $k\binom{n}{k} = (n + 1 - k)\binom{n}{n+1-k}$, we can get

$$f_{(n,n+1-k)}(a + b - t) = (n + 1 - k)\binom{n}{n+1-k} F(a + b - t)^{n-k}[1 - F(a + b - t)]^{k-1} f(a + b - t)$$

i.e.,

$$f_{(n,n+1-k)}(a + b - t) = k\binom{n}{k} [1 - F(t)]^{n-k} F(t)^{k-1} f(t) = f_{(n,k)}(t)$$

□

This lemma shows that if T's pdf is symmetric, then there is also symmetry between $T(n, k)$ and $T(n, n+1-k)$: their pdfs are reflections of each other. The above two lemmas lead to following theorem:

Theorem 5.2 *If the pdf $f(t)$ of a random variable T, defined on $[a, b]$ is* symmetric *about the line $t = \frac{a+b}{2}$, i.e., $f(t) = f(a + b - t)$, then $E[T(n, n + 1 - k)] = a + b - E[T(n, k)]$, and $Var[T(n, k)] = Var[T(n, n + 1 - k)]$.* □

Here we see an example where the monotonicity of $E[T(n, k)]$ with respect to n or k does *not* hold for $Var[T(n, k)]$.

5.3 Server Performance Model

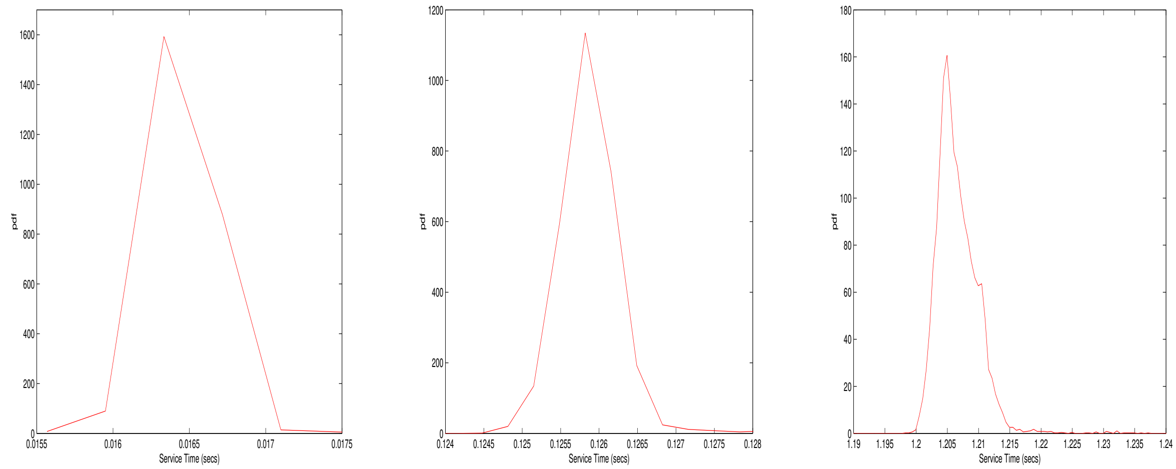
From Eq.(5.2) and Eq.(5.3), $E[T(n, k)]$ is a function of the pdf $f(t)$ of an individual server's data service time. The goal of the data distribution and data acquisition problem is to reduce $E[T(n, k)]$ under various conditions. Before we analyze the data distribution and data acquisition problem, it is necessary to establish some model of $f(t)$.

5.3.1 Abstraction from Experiments

The data service time T depends on many factors in a practical server system, such as computing power (i.e., CPU speed) of the servers and the client, local disk I/O speed of the servers and bandwidth and latency of the communication medium (usually including a reliable communication software layer) connecting the servers and the client. A model considering all the factors will be fairly complex. In this chapter, we will try to model the data service time as a simple probability distribution, that can be analyzed rather easily, and yet can approximate the real data service time closely. Such a model will be abstracted from experimental results of a real data server system.

Our experimental server system consists of several servers, which are PCs running Linux. Each server has data stored on its local hard disk. Data is accessed via the Linux file system. The client is also a PC running the same Linux. The nodes are connected via Myrinet switches. A *sliding window* protocol is used to ensure reliable communication. Experiments are conducted in such a real system to measure the service time for data of different sizes. The procedure of the experiment is as follows: (1) the client sends a request for a certain amount of data to a server; (2) the server reads the data from its local disk and sends it to the client through the reliable communication layer; (3) the data is delivered to the client through the reliable communication layer. The data service time is measured from the instant that the client finishes sending its request to the instant that the client gets the data. We run the above procedure a few thousand times for data of a given size, and get the service time pdf according to the observed frequencies of different ranges of service time. Figure 5.2 shows empirical service time pdfs for data sizes (a) 32 Kbytes, (b) 320 Kbytes and (c) 3200 Kbytes.

The effective data bandwidths in this experiments are quite low, since they are the concatenation of the local disk bandwidth and the reliable communication layer bandwidth. But the shapes of the bandwidth pdfs are more interesting. The experiment results show that the shapes of empirical pdfs of different data size can be approximated by the same distribution. A closer look shows that the width of the distribution base is approximately *proportional to* the data size. More complex distributions, such as the Gamma distribution or the Beta distribution, might give more accuracy. But to simplify the analysis, that follows, we will regard the data service time T as a random variable defined on $[a, b]$ (a



(a) *data size* $= 32Kbytes$ (b) *data size* $= 320Kbytes$ (c) *data size* $= 3200Kbytes$

Figure 5.2: Empirical pdfs of service time for data of different sizes

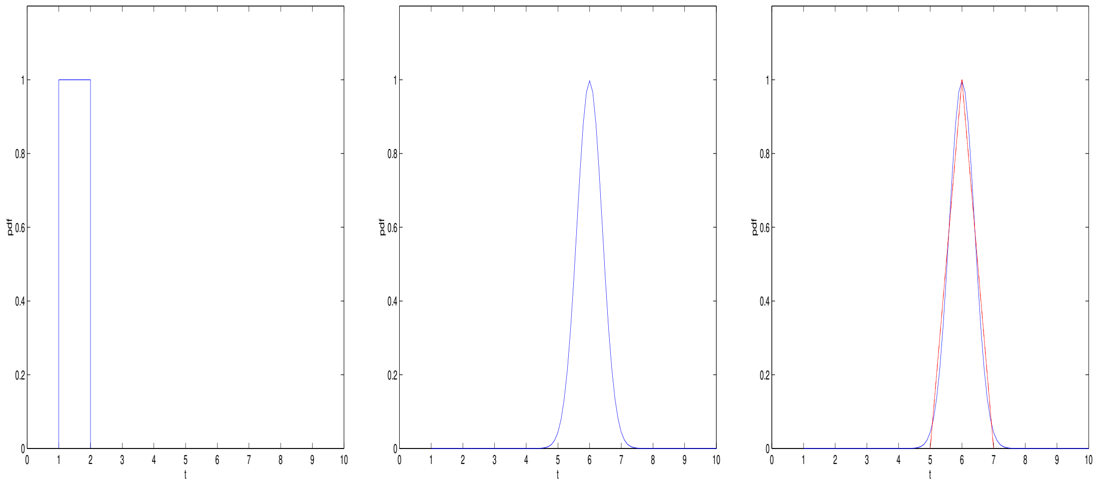
and b are two parameters of a real system), which follows a triangular distribution, denoted $Tr[a, b]$:

$$f(t) = \begin{cases} \frac{4(t-a)}{(b-a)^2} & a \leq t \leq \frac{a+b}{2} \\ \frac{4(b-t)}{(b-a)^2} & \frac{a+b}{2} < t \leq b \end{cases} \tag{5.5}$$

Its cdf (*cumulative distribution function*) is

$$F(t) = \begin{cases} \frac{2(t-a)^2}{(b-a)^2} & a \leq t \leq \frac{a+b}{2} \\ 1 - \frac{2(b-t)^2}{(b-a)^2} & \frac{a+b}{2} < t \leq b \end{cases} \tag{5.6}$$

One explanation for this model is as follows: in a real system, data is delivered in packets of some small size. The delivery time of the ith packet is a random variable t_i, whose probability distribution can be characterized by a uniform distribution over some time span; the t_i's are assumed to be i.i.d. random variables. Then the service time T of the whole data is: $T = s + \sum_i t_i$, where s is another uniform random variable describing the setup (or overhead) time for sending a certain amount of data. Thus the pdf of T is a *Gaussian-like* function, whose base width is approximately proportional to the number of the packets in the data, which in turn is proportional to the data size. For simplicity, we approximate the Gaussian-like function by a suitable triangular function. The distributions are shown in Figure 5.3.



(a) (b) (c)

Figure 5.3: Probability distributions of data service time of (a) single packet, (b) the whole data, (c) the approximation with $Tr[a, b]$

5.3.2 Verification with T(n,1)

Intuitively, having more servers should provide better performance when the amount of data stored on each server is fixed, i.e., $E[T(n,k)]$ decreases as n increases and/or k decreases. We can get pdfs of the $T(n, k)$ for a data server system by evaluating Eq.(5.2) for the service time distribution in Eq.(5.5) and Eq.(5.6). Figure 5.4(a) shows the pdfs of $T(n, 1)$, where $1 \leq n \leq 3$ and T is of the triangular distribution $Tr[1, 2]$. Here we can see the pdf of $T(n, k)$ shifts left as n increases.

To further verify the properties of $E[T(n, k)]$, simple experiments to measure $T(n, 1)$ were done on the experimental server system described in previous subsection. The system consists of three servers. In order to remove other factors that also affect data service time, such as contention in the communication medium (including the reliable communication layer, which is a bottleneck if we use a single client which communicates with the three servers), we use three clients, each of which is served by a separate server. Conceptually the three clients are regarded as a single client, thus the whole data service time is the minimum of the three individual service time of the server-client pairs. Figure 5.4(b) shows the service times (T_1, T_2, and T_3) of the three individual server-client pairs for 3200 Kbytes data each. Since the variance among the three pairs is bigger than the variance within each pair, the whole service time (T_{min}), which is the minimum of the three, is determined by

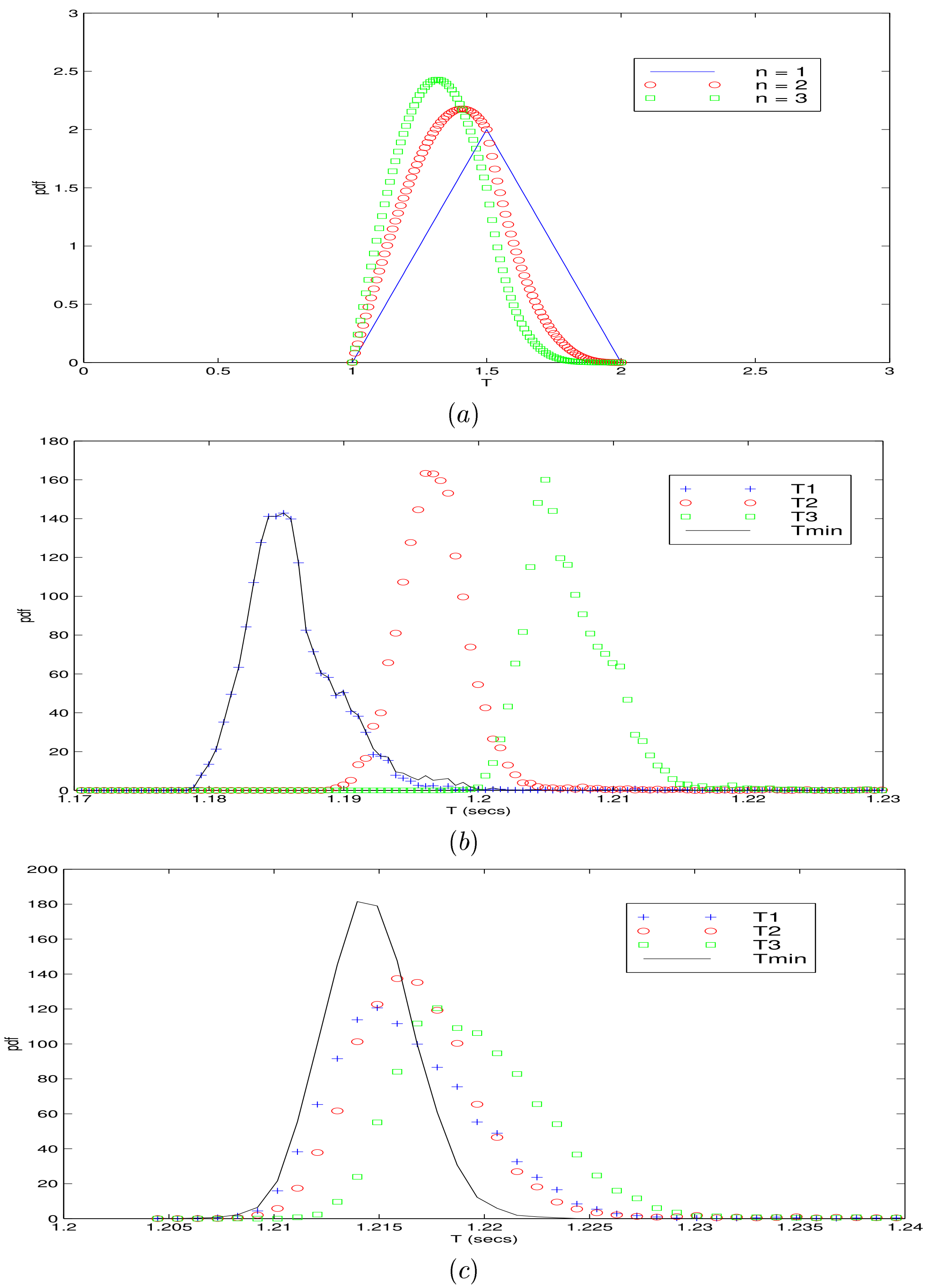


Figure 5.4: pdfs of $T(n, 1)$: (a) analytical result, where the pdf of T is $Tr[1, 2]$, and experimental service time for data of size 3200 Kbytes, where (b) no other loads on the servers, and (c) other random loads on the servers

the service time of the best client-server pair as can be seen in the experimental results. In this case, the pdf of T_{min} is very close to that of T_1. To make the experimental results more interesting, some random loads are added to each server, so that the variance among the three client-server pairs is less than the variance within each pair, i.e., each pair behaves more similarly. The service times of three individual pairs (T_1, T_2, and T_3) and the whole service time (T_{min}) are shown in Figure 5.4(c). Of those four pdfs (T_{min}, T_1, T_2 *and* T_3), that of T_{min} is the leftmost, which supports the analytical properties of $T(n, k)$ and the pdf model of T.

5.4 Design An Efficient System

The performance of a server system with redundancy can be improved in two dimensions: (1) given the total number of the servers in the system, data can be distributed wisely among the servers so that the overall service time servers is minimized. This is the data distribution problem, which consists of determining k and choosing proper MDS array codes accordingly; (2) once the data distribution is set, data should be read from servers wisely so that the whole service time is minimized. This is the data acquisition problem.

The data distribution problem is at the servers' side, while the data acquisition problem is at the client's side. Because of the properties of $E[T(n, k)]$, the two problems are uncorrelated, thus they can be dealt with separately.

5.4.1 Data Distribution Scheme

In a server system, with a given total number of servers, n, we need to determine the number k of the servers which store the *raw* data in order to maximize the performance of the whole system (i.e., to minimize the mean service time of client's data request); given k, the rest of the servers can store the *redundant* data. When n and the pdf $f(t)$ are fixed, $E[T(n, k)]$ decreases monotonically as k decreases. This means that in order to make $E[T(n, k)]$ small, k should be as small as possible. On the other hand, however, the smaller k is, the more data needs to be stored on each server, since the total amount of the data a client needs is always fixed; this means higher service time from each server. Our goal is to find such a k that when both sides of the problem are considered, $E[T(n, k)]$ is minimized.

After the parameter k is determined, in order to achieve optimal performance in terms of $E[T(n,k)]$, we can use MDS array codes to distribute the redundant data so that data from *any* k servers can be assembled to form the whole of the requested data, as was shown in the introduction. Now the remaining problem is to determine k to minimize $E[T(n,k)]$. Applying the pdf model of each server's service time, T, and using MDS codes for distributing the redundant data, we get that if the pdf of T is $Tr[a,b]$ when k=1, then for general k, the corresponding pdf is $Tr[\frac{a}{k}, \frac{b}{k}]$, since the base width of the pdf is proportional to the data size. Theoretically, the optimal k can be calculated as follows:

$$k_{min} = argmin_k \int_0^{\infty} k \binom{n}{k} F(t)^{k-1}[1-F(t)]^{n-k} t f(t) dt \tag{5.7}$$

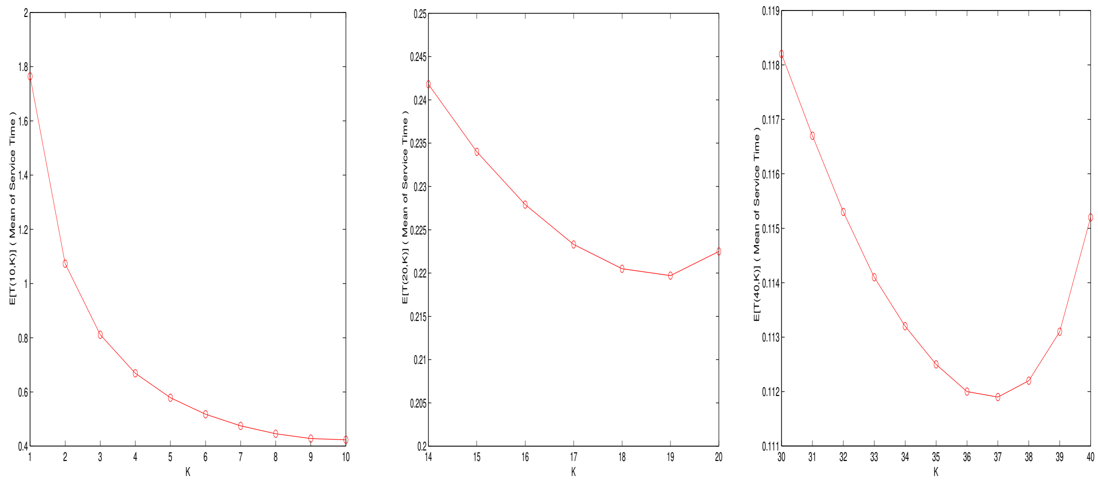
where $f(t)$ and $F(t)$ are as in Eq.(5.5) and Eq.(5.6), except that a and b should be replaced by $\frac{a}{k}$ and $\frac{b}{k}$ respectively. Notice that k_{min} is a function of the entire pdf $f(t)$, not only the mean $E(T)$ and the variance $Var[T]$.

Even for a simple pdf such as $Tr[a,b]$, the above equation can not be solved in closed form. But in practice, the system parameters a and b can be determined by experiments, then the above equation can be solved numerically. Figure 5.5 gives several examples of solving the above equation. In the examples, $a = 1$ and $b = 5$. For $n = 10$, 20, and 40, $E[T(n,k)]$ is calculated for $1 \leq k \leq n$. The results are shown in Figure5.5(a)(b)(c), where (b) and (c) only show the last few values for k, since for small k, $E[T(n,k)]$ decreases monotonically as k increases. From the results, we can see (a) $k_{min} = 10$, when $n = 10$, (b) $k_{min} = 19$, when $n = 20$, and (c) $k_{min} = 37$, when n = 40.

Even though the above examples use specific pdfs, the same method also apply with other pdfs by plugging suitable $f(t)$ into Eq.(5.7). Thus, for a given server system, such a k_{min} can always be found. Proper MDS array codes can then be used based on the (n,k) pair. Thus we get an optimal data distribution scheme for a given server system.

5.4.2 Data Acquisition Scheme

Once the data distribution scheme is set, i.e., k is determined and the proper MDS array code is chosen, the client needs to decide how to request (or read) data. In general, a client should send its request to as many servers as possible and also make the amount of data it needs from *each* server as small as possible, since the properties of $E[T(n,k)]$ show that more redundancy brings better performance. For a specific distribution scheme, the client



$(a)\ n = 10$ $(b)\ n = 20$ $(c)\ n = 40$

Figure 5.5: $E[T(n,k)]$ vs. k for different n, where $a = 1$ and $b = 5$

needs to calculate the pdfs of all possible data read schemes, and then choose an optimal read scheme. Since the read schemes are closely related to the MDS array code being used, here we will give an example using a specific code to show the guidelines for choosing an optimal read scheme.

In this example, the server system has $2n$ servers, and the data that the client requests can be assembled from any $2n-2$ servers, i.e., this is a $(2n, 2n-2)$ system. The B-Code in Chapter 3 can be used to implement this system. The data distribution using the B-Code is as follows: (1) the whole raw (*information*) data is partitioned into $2n(n-1)$ blocks of equal size (some paddings are added if necessary); (2) each of the $2n$ servers stores $n-1$ blocks of the data; (3) $2n$ blocks of redundant (or *parity*) data are calculated according to the encoding rules of the B-Code, i.e., each parity block is an XOR of suitable $2n-2$ raw data blocks, and then each server stores 1 parity block. The structure of the B-Code is shown in Figure 3.2 in Chapter 3.

The MDS property of the B-Code gives 3 schemes for reconstructing the whole raw data from the data stored on $2n$ servers, each of which has $n-1$ blocks of raw data and 1 block of parity data: (1) read from all of the $2n$ servers, each of which sends its $n-1$ blocks of raw data; (2) read from any $2n-2$ servers, each of which sends all of its n blocks of data (including raw and parity data); (3) read from all of the $2n$ servers, each of which sends all of its n blocks of data. The 3 schemes are shown in Figure 5.6, where the shaded parts are the data to read.

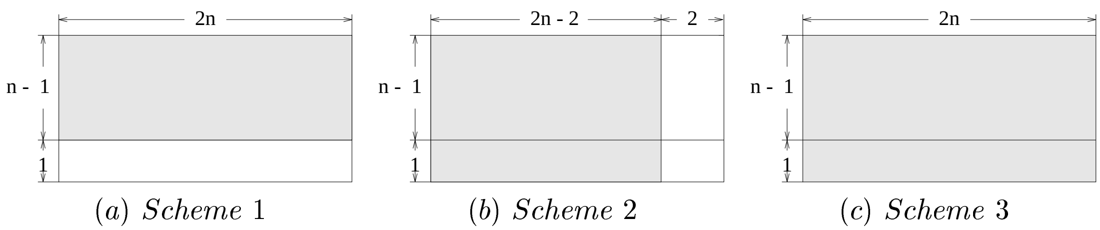


Figure 5.6: Three read schemes using the B-Code

Notice that there is no redundant data in scheme (1) or scheme (2), so the client must wait until it receives all the data from all the servers. But in scheme (3), there is redundant data, then the client only needs to receive data from any $2n-2$ of the $2n$ requested servers. Let $E[T(2n,2n)]_{n-1}$, $E[T(2n-2,2n-2)]_n$ and $E[T(2n,2n-2)]_n$ denote the mean data service time of the three schemes respectively. From Property 1 of Theorem 5.1, $E[T(2n-2,2n-2)]_n > E[T(2n,2n-2)]_n$. But the relation between $E[T(2n,2n)]_{n-1}$ and either $E[T(2n-2,2n-2)]_n$ or $E[T(2n,2n-2)]_n$ is not so obvious, since in scheme (1) the client needs to wait for more servers, but needs less data (thus less service time) from each server. So to determine which scheme is best scheme for a given system, we need to calculate the pdf of the whole service time for all possible the schemes, which are scheme (1) and scheme (3) in this case.

Assume that the pdf of the time T for each server to send n blocks of data to the client is $Tr[a,b]$; then the pdf of T in scheme (1) is $Tr[\frac{n-1}{n}a, \frac{n-1}{n}b]$, since each server only needs to send $n-1$ blocks of data, and the pdf of T in scheme (2) or (3) is $Tr[a,b]$. Now the pdfs of the whole service time in the different schemes can be calculated according to Eq.(5.2), Eq.(5.5) and Eq.(5.6). Figure 5.7 shows the pdfs for different values of n, where $a=1$ and $b=10$.

Using Eq.(5.3), the mean of the whole service time of different schemes can be calculated. These means are listed in Table 5.1, for $a=1$ and $b=10$.

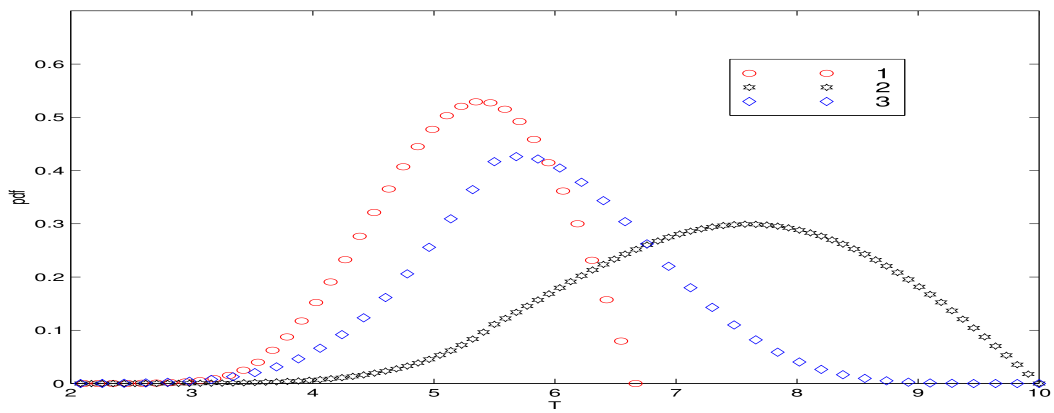
The above calculations show that the performance of the three schemes depends on the system parameter n (when a and b are fixed). In a small server system, scheme (1) is the best. As n increases, scheme (3) becomes better. For a system of 6 servers ($n=3$), scheme (1) is the best, but for systems of 14 servers ($n=7$) and 20 servers ($n=10$), scheme (3) is

n	3	7	10
$E[T(2n, 2n)]_{n-1}$	**5.2195**	7.3128	7.8857
$E[T(2n-2, 2n-2)]_n$	7.4089	8.4207	8.6976
$E[T(2n, 2n-2)]_n$	5.8910	**7.2466**	**7.6786**

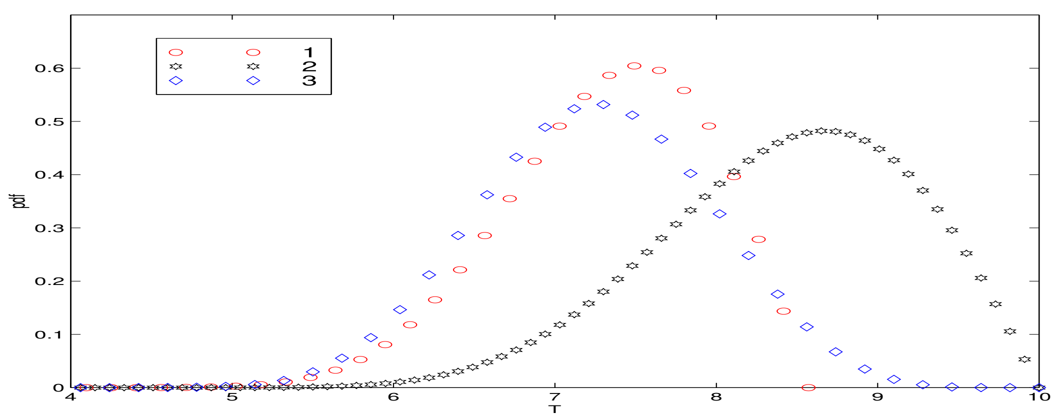
Table 5.1: Mean service time of different data read schemes, where $a = 1$, and $b = 10$

the best.

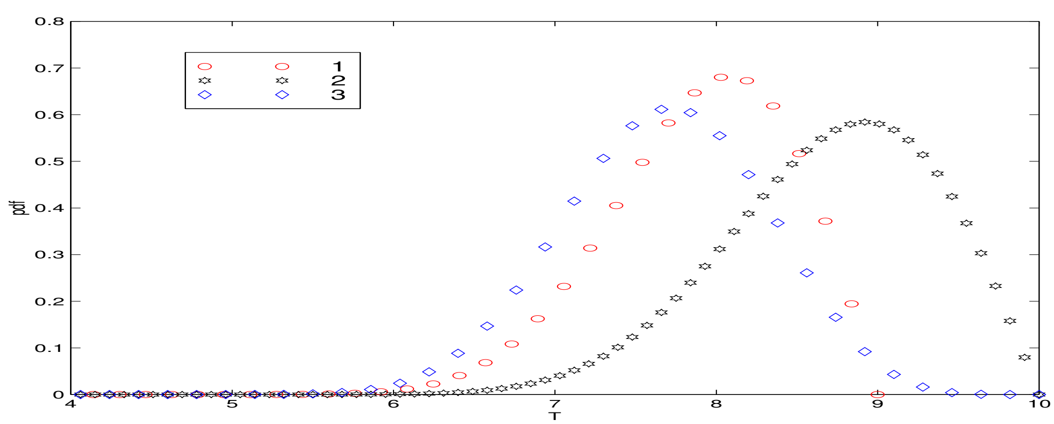
Though quite simple, the above example shows that after the data distribution is set at the server side, the client has different ways of reading data from the servers. For a given system (i.e., a certain the pdf of T, a fixed (n, k) pair and a particular code), there always exists an optimal read scheme for the client. Finding this scheme requires careful calculation. Since the read schemes are highly related to the codes used, exploring codes that offer more read choices is an interesting research problem.



$(a)\ n = 3$



$(b)\ n = 7$



$(c)\ n = 10$

Figure 5.7: PDFs of different data read scheme, where $a = 1$, $b = 10$; 1, 2 and 3 represent scheme (1), (2) and (3) respectively.

5.5 Summary

We have explored options for improving the performance of data server systems by introducing data redundancy based on array codes. We have proposed methods of achieving better performance at both the server side and the client side of systems, namely finding an optimal data distribution scheme and an optimal data acquisition scheme. We have also given a simple probability model for a real data server system, based on experimental results. Our additional experimental results also verify qualitatively our analysis results.

In this chapter, the system model is rather simple: data requests from one client or multiple clients are processed sequentially, i.e., at one time, and each server only handles one data request. But in a more practical system, such as a web server system, multiple data requests can come according to some stochastic process, and they can be processed by the servers in a parallel fashion such that the number of requests processed per unit time is maximized. Optimizing such systems is an interesting and useful research problem that might require that the servers use some sophisticated scheduling schemes, based on the results in this chapter.

Chapter 6 Conclusions and Future Directions

6.1 Conclusions

This thesis deals with two issues in highly available distributed storage systems: reliability and efficiency. To achieve reliability, two new families of MDS array codes are presented, whose optimal encoding operations are optimal. To improve efficiency (performance) of general distributed data systems, including storage systems, two problems and their solutions are proposed, namely the data distribution problem and data acquisition problem. A new efficient deterministic voting algorithm for distributed data is also presented.

The two new classes of MDS array codes of distance 3, the X-code and the B-Code, are the first two classes of array codes that place parity and information bits in the same column. The two codes are still *systematic* in the sense that all information bits can be directly obtained from their codewords without any computation. The encoding operations of both codes are optimal, i.e., their update complexity achieves the lower bound. In addition to encoding algorithms to these array codes, efficient decoding algorithms, both for erasure-correcting and for error-correcting, are also proposed.

The X-Code has a very simple geometrical structure, namely the parity bits are constructed along two groups of parallel *parity lines* of slopes 1 and -1 respectively, where its name comes from. This simple geometrical structure enables simple erasure-decoding and error-decoding algorithms, using only XORs and vector cyclic-shift operations.

The B-Code is related to a 3-decade old graph theory problem. It is proven in this thesis that constructing a B-Code of odd length is exactly equivalent to constructing a perfect one-factorization (or P1F) of a complete graph. Thus if the P1F conjecture is solved, then B-Codes of arbitrary length can be constructed. On the other hand, B-Codes of new lengths can lead to constructions of P1F of new complete graphs. An efficient error-correcting algorithm for the B-Code is also presented, based on the relations between the B-Code and its dual; this algorithm might give a hint of developing efficient decoding algorithms for other codes.

To show that in distributed systems, data redundancy should be actively introduced to improve system performance, a novel deterministic voting scheme that uses error-correcting codes is proposed, which generalizes all known simple deterministic voting algorithms. The new voting scheme greatly reduces communication complexity while still providing the correct deterministic voting result. The scheme can be tuned for optimal average case communication complexity by choosing the parameters of the error-correcting code, thus it is very adaptive to various application environments with different error rates.

It is also shown that in general distributed storage systems, proper data redundancy can improve the performance of the systems. Two problems are identified to improve the performance of general data server systems, namely the data distribution problem and the data acquisition problem. Solutions are proposed, as are general analytical results on the performance of (n, k) systems. A simple service time model of a practical disk-based distributed server system is given based on experimental results, which is used as a starting point for data distribution and data acquisition schemes. These results can be used in more sophisticated scheduling schemes that optimize or improve the performance of data server systems that serve multiple clients at the same time.

6.2 Future Directions

Much research still needs to be done to improve the availability of distributed storage systems. More error-correcting codes with low computational overhead that provide flexible reliability need to be designed. Array codes are a good class of codes that should get more researchers' attention. Distributed storage systems can become popular only with matching distributed file systems. Many issues in distributed file systems, such as efficiently reaching consistency of distributed data, maintaining data integrity in the presence of transient faults, and achieving graceful performance degradation while faulty parts of systems are replaced, should be solved. Also distributed storage systems should be easy to scale, easy to manage and easy to maintain. In short, there are many research problems to solve in order to build highly available distributed storage systems.

Many open problems directly related to the topics in the thesis have been already raised in the previous chapters. We summarize some important ones here:

- More MDS array codes: here *more* has many meanings. We need to find codes with distances greater than 3. X-like codes may be a solution. We also need to find codes with more lengths, ideally, arbitrary lengths. Codes that have parity bits mixed with information bits in the same column are typically difficult to shorten, thus one solution to this problem is to find more codes with more lengths. While they have optimal encoding operations and optimal erasure-decoding in terms of the total number of operations, the X-Code and the B-Code are *not* optimal in decoding in number of decoding steps if *parallel* decoding is used. Since in distributed storage systems, parallel decoding means reading data in parallel, more codes with fewer parallel decoding steps can improve the data-read performance of many applications. The *lower bound* of parallel erasure-decoding steps for a given code is *not* even known yet.

- The B-Code: it still needs to be proven that the B-Code of length $2n+1$ can always be constructed from the B-Code of length $2n$, thus the B-Code is totally equivalent to the P1F. Also how to construct a new B-Code from the known B-Codes is very useful, since this gives new P1F construction methods, in addition to more codes that can be obtained.

- Voting: the efficient voting algorithm in Chapter 4 uses 2 or 3 rounds. It is an open problem whether there is a voting scheme that always uses *only* 1 round, but can still reduce average communication complexity. Also the voting schemes discussed use a broadcast model, another interesting problem is how to reduce average communication complexity if a point-to-point communication model is used.

- General data servers: Chapter 5 deals with the case of only one client at a time. But in practical data server systems, it is common to have multiple clients at the same time or a group of requests that are already queued. How to schedule the processing of these multiple data requests is an interesting and difficult research problem.

We conclude this thesis by two more problems related to codes that can be used in distributed storage systems to improve availability.

6.2.1 Reed-Solomon Codes as Array Codes

Reed-Solomon codes are more flexible in lengths and distances. Their only shortcoming is their relatively complex encoding and decoding operations which use finite field operations. Since array codes are 2-dimensional, they are generalizations of 1-dimensional codes, and this generalization applies to any arbitrary 1-dimensional code. Thus Reed-Solomon codes can also be described as array codes. The following example shows an array code representation of a (7,2,6) Reed-Solomon code.

Example 6.1 *Array representation of (7,2,6) Reed-Solomon code*

Let α be a root of the primitive binary polynomial $x^3 + x + 1$ that generates the Galois field GF(8). Using 1, α and α^2 as a basis, the 8 elements of GF(8) can be represented as vectors:

0	α	α^2	α^3	α^4	α^5	α^6	1
000	010	001	110	011	111	101	100

Now a (7,2,6) Reed-Solomon code can be constructed using the following *generator polynomial* [19]:

$$g(x) = (x + \alpha)(x + \alpha^2)(x + \alpha^3)(x + \alpha^4)(x + \alpha^5)$$

i.e.,

$$g(x) = x^5 + \alpha^2 x^4 + (1 + \alpha)x^3 + (1 + \alpha^2)x^2 + (\alpha + \alpha^2)x + \alpha$$

Any information to be encoded can be represented by an information polynomial of degree 1, i.e.,

$$m(x) = (a_1 + a_2\alpha + a_3\alpha^2)x + (b_1 + b_2\alpha + b_3\alpha^2)$$

where the information bits, the a_i's and the b_i's, are in GF(2), i.e., they are binary. The above information polynomial can then be described as an array of size 3×2 over GF(2):

a_1	b_1
a_2	b_2
a_3	b_3

The code polynomial is then of degree 6:

$$c(x) = m(x)g(x)$$

After calculating and simplifying $c(x)$, a codeword of the (7,2,6) Reed-Solomon code can then be represented by an array of size 3×7, in a way similar to the representation of the information polynomial:

a_1	a_2+b_1	$a_1+a_3+b_2$	$a_1+a_2+b_1+b_3$	$a_2+a_3+b_1+b_2$	$a_3+b_2+b_3$	b_3
a_2	$a_2+a_3+b_2$	$a_1+a_2+a_3+b_2+b_3$	$a_3+b_1+b_2+b_3$	$a_1+a_2+b_3$	$a_1+a_3+b_1+b_2$	b_1+b_3
a_3	$a_1+a_3+b_3$	$a_2+a_3+b_1+b_3$	$a_1+b_2+b_3$	$a_1+a_2+a_3+b_1$	$a_2+b_1+b_2+b_3$	b_2

Table 6.1: An array representation of a (7,2,6) Reed-Solomon code. Total number of additions: 39.

□

Representing Reed-Solomon codes using arrays alone does *not* simplify encoding operations. Further simplifications need to be done. For the (7,2,6) Reed-Solomon code in Table 6.1, the last column certainly can be simplified to (b_3, b_1, b_2) instead of (b_3, b_1+b_3, b_2), without changing the code's MDS property, since the two vectors span the same space. We can simplify all other columns in a similar way, so that the density of each column is reduced to its minimum while the space spanned by the column remains unchanged. The following array is a simplified form of the (7,2,6) Reed-Solomon code, derived from its array form in the above example:

a_1	a_2+b_1	$a_1+a_3+b_2$	$a_2+a_3+b_3$	$a_2+a_3+b_1+b_2$	$a_3+b_2+b_3$	b_3
a_2	$a_2+a_3+b_2$	a_2+b_3	$a_1+a_3+b_1$	$a_1+a_2+b_3$	$a_1+b_1+b_3$	b_1
a_3	$a_1+a_3+b_3$	a_3+b_1	$a_1+b_2+b_3$	a_1+b_2	$a_1+a_2+b_2$	b_2

Table 6.2: A simplified array representation of a (7,2,6) Reed-Solomon code. Total number of additions: 27.

Though the above array form has been simplified a lot, i.e., its encoding operation is simpler than the original Reed-Solomon code, it can be further simplified as in Table 6.3 without changing the update complexity, i.e., some intermediate parity bits (s_i's) are calculated once and then reused in calculating other parity bits.

So this gives a way to design more MDS array codes with more choices of length and distance, based on Reed-Solomon codes. Of course, this method should apply with *any* other linear code that is not MDS, i.e., any linear code can be described by a simplified array code with simple encoding operations.

Even though the array in Table 6.3 has been simplified a lot, it is still not clear whether it is the *optimal* form in terms of encoding operations, since we can simplify multiple columns

a_1	s_2	$a_1 + s_3$	$a_3 + s_5$	$s_2 + s_3$	$b_3 + s_3$	b_3
a_2	$a_2 + s_3$	s_5	$a_1 + s_4$	$a_2 + s_1$	$b_1 + s_1$	b_1
a_3	$a_3 + s_1$	s_4	$b_2 + s_1$	s_6	$a_2 + s_6$	b_2

Table 6.3: A further simplified array representation of a (7,2,6) Reed-Solomon code, where $s_1 = a_1 + b_3$, $s_2 = a_2 + b_1$, $s_3 = a_3 + b_2$, $s_4 = a_3 + b_1$, $s_5 = a_2 + b_3$, and $s_6 = a_1 + b_2$. Total number of additions: 17.

at the same time as long as these columns span the same space. There are many research problems to solve, related to representing arbitrary codes as array codes. To name a few: 1) given a linear code, what is its optimal array code representation in terms of encoding complexity? or, how does one determine whether an array description is optimal in terms of its component density, i.e., total number of information bits appearing? 2) how does one design an efficient erasure-correcting algorithm once an array code is derived from a Reed-Solomon code or from another code? 3) how does one design efficient *multiple* error-correcting algorithms for an array code?

6.2.2 Strong MDS Codes

For an MDS (l,k) array code of size $n \times l$, its MDS property means that the nk original information bits can be recovered from any k columns with each containing n bits. As shown in Chapter 5, if an (l, k) MDS array code is used in a data distribution scheme, a matching data acquisition scheme may read nk bits from m ($m \geq k$) servers, with the ith server sending a_i bits such that $\sum_{i=1}^{m} a_i = nk$, where of course $0 \leq a_i \leq n$ and $1 \leq i \leq m$. Mapping to array codes, this leads to a new MDS property, which is called the *strong MDS property*, as defined as follows:

Definition 6.1 *(strong MDS property) An array code of size $n \times l$ with nk raw information bits has the* strong MDS property*, if, for any given m columns and any given series of positive integers a_i, where $k \leq m \leq l$, $0 \leq a_i \leq n$, $1 \leq i \leq m$ and $\sum_{i=1}^{m} a_i = nk$, there* always *exist a_i bits from the ith column such that the nk raw information bits can be reconstructed from these nk bits from the m columns.*

The interested reader can verify that the Reed-Solomon code in Table 6.2 has the strong MDS property. For example, if $m = 3$, and we are given the first 3 columns, let $a_1 = a_2 =$

$a_3 = 2$, then we can choose either of the following 6 bits from the the 3 columns with 2 bits from each column:

a_1	$a_2 + b_1$	$a_2 + b_3$
a_2	$a_2 + a_3 + b_2$	$a_3 + b_1$

a_1	$a_2 + b_1$	$a_2 + b_3$
a_3	$a_2 + a_3 + b_2$	$a_3 + b_1$

Codes with the strong MDS property can provide more choices of which data to read from multiple servers in an optimal data acquisition scheme. They can also provide more flexibility between redundancy and efficiency when distributing data. So codes with the strong MDS property will have useful roles in many applications in distributed storage systems. From the definition, any code with the strong MDS property is of course MDS.

A final question is how to construct codes with the strong MDS property? Since any code can be represented in array form, we answer this question by the following conjecture and conclude this thesis:

Conjecture 6.1 *(Strong MDS Conjecture)*

All MDS codes have the strong MDS property.

Bibliography

[1] K.A.S. Abdel-Ghaffar and A. El Abbadi, "An Optimal Strategy for Computing File Copies," IEEE Trans. on Parallel and Distributed Systems, 5(1), Jan. 1994.

[2] B. A. Anderson, "Symmetry Groups of Some Perfect 1-Factorizations of Complete Graphs," *Discrete Mathematics*, 18, 227-234, 1977.

[3] A. Behr and L. Camarinopoulos, "Two Formulas for Computing the Reliability of Incomplete k-out-of-n:G Systems," *IEEE Trans. on Reliability*, 46(3), 421-429, Sep. 1997.

[4] M. Blaum, J. Brady, J. Bruck and J. Menon, "EVENODD: An Efficient Scheme for Tolerating Double Disk Failures in RAID Architectures," *IEEE Trans. on Computers*, 44(2), 192-202, Feb. 1995.

[5] M. Blaum, J. Bruck, A. Vardy, "MDS Array Codes with Independent Parity Symbols," *IEEE Trans. on Information Theory*, 42(2), 529-542, March 1996.

[6] M. Blaum, P. G. Farrell and H. C. A. van Tilborg, "Chapter on Array Codes", Handbook of Coding Theory, edited by V. S. Pless and W. C. Huffman, to appear.

[7] M. Blaum, R. M. Roth, "New Array Codes for Multiple Phased Burst Correction," *IEEE Trans. on Information Theory*, 39(1), 66-77, Jan. 1993.

[8] M. Blaum, R. M. Roth, "On Lowest-Density MDS Codes," *IEEE Trans. on Information Theory*, 45(1), 46-59, Jan. 1999.

[9] D. M. Blough and G. F. Sullivan, "Voting Using Predispositions," *IEEE Trans. on Reliability*, 43(4), 604-616, 1994.

[10] K. Echtle, "Fault-Masking with Reduced Redundant Communication," *16th Annual International Symp. on Fault-Tolerant Computing Systems*, vol.16, 178-183, 1986.

[11] M. N. Frank, "Dispersity Routing in Store-and-Forward Networks," Ph.D. thesis, University of Pennsylvania, 1975.

[12] P. G. Farrell, "A Survey of Array Error Control Codes," *ETT* , 3(5), 441-454, 1992.

[13] R. G. Gallager, "Low-Density Parity-Check Codes," MIT Press, Cambridge, Massachusetts, 1963.

[14] J. Gemmell, "Scalable Reliable Multicast Using Erasuring-Correcting Re-Sends," Technical Report MSR-TR-97-20, Microsoft Research, June, 1997.

[15] R. M. Goodman, R. J. McEliece and M. Sayano, "Phased Burst Error Correcting Arrays Codes," *IEEE Trans. on Information Theory*, 39, 684-693,1993.

[16] A. Kotzig, "Hamilton Graphs and Hamilton Circuits," *Theory of Graphs and Its Applications* (Proc. Sympos. Smolenice), 63-82, 1963.

[17] T. Krol, "(N,K) Concept Fault Tolerance," *IEEE Trans. on Computers*, 35(4), 339-349, April 1986.

[18] Darrell D.E. Long and Jehan-François Pâris, "Voting Without Version Numbers," *Proceedings of the International Conference on Performance, Computing, and Communications*, 139-145, Feb. 1997.

[19] F. J. MacWilliams and N. J. A. Sloane, *The Theory of Error Correcting Codes*, Amsterdam: North-Holland, 1977.

[20] J. F. Nebus, "Parallel Data Compression for Fault Tolerance," *Computer Design*, 127-134, Apr. 1983.

[21] G. Noubir and H. J. Nussbaumer, "Using Error Control Codes to Reduce the communication Complexity of Voting in NMR Systems," Technical Report, Dept. of Computer Science, Swiss Federal Institute of Technology in Lausanne(EPFL), 1995.

[22] Norman K. Ouchi, "System for Recovering Data Stored in Failed Memory Unit," US Patent 4092732, May 30, 1978.

[23] A. Papoulis, *Probability, Random Variables, and Stochastic Processes*, 2nd Edition, McGraw-Hill, Inc., 1984.

[24] B. Parhami, "Voting Algorithms," *IEEE Trans. on Reliability*, 43(4):617-629, 1994.

[25] D. A. Patterson, G. A. Gibson and R. H. Katz, "A Case for Redundant Arrays of Inexpensive Disks," *Proc. SIGMOD Int. Conf. Data Management*, 109-116, Chicago, IL, 1988.

[26] C. A. Polyzois, A. Bhide and D. M. Dias, "Disk Mirroring with Alternating Deferred Updates", Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993.

[27] H. M. Sun and S. P. Shieh, "Optimal Information-Dispersal for Increasing the Reliability of a Distributed Service," *IEEE Trans. on Reliability*, 46(4), 462-466, Dec. 1997.

[28] R. M. Tanner, "A Recursive Approach to Low Complexity Codes," *IEEE Trans. on Information Theory*, 27(5), 533-547, Sep. 1981.

[29] D. G. Wagner, "On the Perfect One-Factorization Conjecture," *Discrete Mathematics*, 104, 211-215, 1992.

[30] W. D. Wallis, *One-Factorizations*, Kluwer Academic Publisher, 1997.

[31] Stephen B. Wicker, *Error Control Systems for Digital Communication and Storage*, Prentice-Hall Inc., 1995.

[32] Samuel S. Wilks, *Mathematical Statistics*, John Wiley & Sons, Inc., 1963.

[33] L. Xu and J. Bruck, "X-Code: MDS Array Codes with Optimal Encoding," *IEEE Trans. on Information Theory*, 45(1), 272-276, Jan., 1999.

[34] L. Xu, V. Bohossian, J. Bruck and D. Wagner, "Low Density MDS Codes and Factors of Complete Graphs," Proceedings of 1998 IEEE Symposium on Information Theory, Aug., 1998; Revised version to appear in *IEEE Trans. on Information Theory.*

[35] L. Xu and J. Bruck, "Deterministic Voting in Distributed Systems Using Error-Correcting Codes," *IEEE Trans. on Parallel and Distributed Systems*, 9(8), 813-824, Aug., 1998.

[36] L. Xu and J. Bruck, "Improving the Performance of Data Servers Using Array Codes," submitted to the 8th International Symposium on High Performance Distributed Computing, Redondo Beach, CA, USA, August 3-6, 1999.

[37] G. V. Zaitsev, V. A. Zinov'ev, and N. V. Semakov, "Minimum-Check-Density Codes for Correcting Bytes of Errors, Erasures, Or Defects," *Problems of Information Transmission*, 19(3), 197-204, 1983.

[38] A. Ziv and J. Bruck, "Checkpointing in Parallel and Distributed Systems," *Parallel and Distributed Computing Handbook*, 274-302, McGraw-Hill, 1996.