# 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 $i$th ($i = 1, \cdots, 7$) node just broadcasts the 3 bits in the $i$th 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.