

Error Correction Codes in P2P

system

IKT 404

Distributed System

June, 2006



HØGSKOLEN I AGDER

Agder University College

Faculty of Engineering and Science

<p>Author(s): Kun Yang; Fei Yao Li Zhang; Wen Hu</p>	<p>Supervisor(s): Ulf Carlsen</p>
<p>Version: 1.0 Status: FINAL</p>	<p>Pages: 51 (including this page) Modified date: 2006-06-02</p>
<p>Keywords: Error-correcting codes Reed-Solomon codes Tornado codes Matlab P2P systems</p>	
<p>Abstract:</p> <p>The issue of efficiently retrieving a file that has been broken into blocks and distributed across the wide-area pervades applications that utilize peer-to-peer, and distributed file systems. Error Correction Codes are able to improve the fault-tolerance and performance of different file systems. This report describes some overviews on applications of ECC in the distributed systems.</p> <p>The problem of implement the application of Error-correcting codes in the peer-to-peer systems has been approached in the way with the use of two different algorithms: Reed-Solomon and Tornado codes. And the results show that both of the algorithms perform well, the error rate approximately equal to zero.</p> <p>Matlab was used for the implementation.</p> <p>In this report, we first introduce some basic theory of Error-correcting codes and some detail information of our project. In the second section, it is showed that the comparison among different P2P systems. In the next section, we describe how we design and implement two different error-correcting codes in detail, and then we evaluate the results. Finally, it is presented that the references we used and give the codes in Appendix A and B.</p>	

Table of content

Abstract.....	1
1 Introduction.....	4
1.1 Project background	4
1.2 Project description.....	4
1.3 Limitation.....	5
2 Related technologies.....	5
2.1 Peer to Peer System.....	5
2.2 Security in peer-to-peer systems	7
2.3 Redundancy.....	9
2.4 Error correction codes	11
3 Use of ECC in different P2P systems.....	12
3.1 Principle	13
3.2 Bit Torrent	14
3.2.1 How BitTorrent works.....	14
3.2.2 Why not using error-correcting codes	15
3.3 Avalanche	15
3.3.1 How Avalanche works.....	15
3.3.2 Using error correction codes	16
3.4 OceanStore	17
4 Two different error-correction codes.....	18
4.1 Reed-Solomon Codes.....	18
4.1.1 Introduction.....	18
4.1.2 Properties of Reed-Solomon codes	19
4.1.3 Architectures for encoding and decoding Reed-Solomon codes.....	22
4.1.4 Implementation of Reed-Solomon encoders and decoders	23
4.2 Tornado Codes	25
4.2.1 Overview of Tornado Codes.....	25
4.2.2 Properties of Tornado code.....	26
4.2.3 How does Tornado Codes work	27
5. Implementation	29
5.1 Implementation with Reed-Solomon Codes	30
5.1.1 Generate the files.....	30
5.1.2 Construct the redundancy.....	30
5.1.3 Transfer the files.....	31
5.1.4 Reconstruct the data	33
5.1.5 Compare the result	34
5.2 Implementation with Tornado Codes	34
5.2.1 Generate the file	34
5.2.2 Construct the redundancy.....	36
5.2.3 Transfer the file	38
5.2.4 Reconstruct the data	39
5.2.5 Compare the result	40

6. Conclusion	41
7. References	42
8 Appendixes.....	43
8.1 Codes using Reed-Solomon	43
8.2 Codes using Tornado.....	46

1 Introduction

1.1 Project background

Distributed systems are becoming more and more popular in the world. One of important characteristic is that they should be able to tolerable to as many errors as possible which maybe occur during the transferring or in the clients because of technical reasons. If their ability of fault tolerance is very low, they can crash down very easily. In that case, they are unreliable and don't worth to be implemented.

In order to improve the ability of fault tolerance in distributed systems, many techniques have been proposed. One important method is to implement Error-correcting codes.

Error-correcting codes have been used around for decades. And recently they have been used for failure recovery in the distributed systems. In principle, decoding is to get some redundancies according to some algorithms. As a result, we can increase the length of data. If we can receive the same number of original data or a bit more, we can rebuild the whole data. And it has nothing to do with which parts you have received, but just depend on the number of data you have received. In this way, we can make our distributed systems more reliable.

1.2 Project description

Peer-to-peer system is one typical distributed system which has been used widely. In order to have a good understanding on role of Error-correcting codes in peer-to-peer systems, we will get some overviews on applications of Error-correcting codes in the distributed systems and compare the peer-to-peer systems who have implemented the technique with the ones who haven't.

After that, we will evaluate the performance of them and get some conclusion.

Later, we will focus on the two typical codes: Reed-Solomon codes and Tornado codes and tell advantages and disadvantages.

Finally, we will implement the two algorithms with MATLAB and prove them according to the result.

1.3 Limitation

Because of the condition, we can't test the systems on our own. Instead of doing that, we checked a lot of resources and make some theoretical conclusions.

2 Related technologies

2.1 Peer to Peer System

Peer-to-peer systems, beginning with Napster, Gnutella, and several other related systems, became immensely popular in the past few years, primarily because they offered a way for people to get music without paying for it.

Peer-to-peer networking is the utilization of the relatively powerful computers (personal computers) that exist at the edge of the Internet for more than just client-based-computing task. The modern personal computer (PC) has a very fast processor, vast memory, and a large hard disk, none of which are being fully utilized when performing common computing tasks such as e-mail and Web browsing. The modern PC can easily act as both a client and server (a peer) for many types of

applications.

The typical computing model for many applications is a client/server model. A server computer typically has vast resources and responds to requests for resources and data from client computers. A good example of the client/server model of computing is Web browsing. Web servers on the Internet are typically high-end dedicated server computers with very fast processors (or multiple processors) and huge hard dist arrays. The Web server stores all of the content associated with a Web site (HTML files, graphics, audio and video files, etc.) and listens for incoming requests to view the information on a particular Web page. When a page is requested, the Web server sends the page and its associated files to the requesting client.

Peer-to-peer applications have become immensely popular in the Internet. Traffic measurement shows that p2p traffic is starting to dominate the bandwidth in certain segments of the Internet. Among p2p applications, file sharing is perhaps the most popular application. Compared to traditional client/server file sharing (such as FTP, WWW), p2p file sharing has one big advantage, namely, scalability. The performance of rational file sharing applications deteriorates rapidly as the number of clients increases, while in a well-designed p2p file sharing system, more peers generally means better performance. There are several issues have to be addressed in order to understand the behavior of the system.

- Peer Evolution: In p2p file sharing, the number of peers in the system is an important factor in determining network performance. Therefore, it is useful to study how the number of peers evolves as a function of the request arrival rate, the peer departure rate, the unloading/downloading bandwidth of each peer, etc.
- Scalability: To realize the advantages of p2p file sharing, it is important for the network performance to not deteriorate, and preferably to actually improve, as the size of the network increases. Network performance can be measured by the average file downloading time and the size of the network can be characterized

by the number of peers, the arrival rate of peers, etc.

- **File Sharing Efficiency:** It is common for peers in a p2p network to have different unloading/downloading bandwidths. Further, in Bit Torrent-like systems, a file may be broken into smaller pieces and the pieces may be distributed at random among the peers in the network. To efficiently download the file, it is important to design the file-sharing protocol such that each peer is matched with others who have the pieces of the file that it needs and further, to ensure that the downloading bandwidth of each peer is fully utilized.
- **Incentives to prevent free-riding:** Free-riding is a major cause for concern in p2p networks. Free-riders are peers who try to download from others while not contributing to the network, i.e, by not unloading to others. Thus, most p2p networks try to build in some incentives to deter peers from free-riding. Once the incentive mechanism is introduced into the network, each peer may try to maximize its own net benefit within the constraints of the incentive mechanism. Thus, it is important to study the effect of such behavior on the network performance.

2.2 Security in peer-to-peer systems

Making p2p systems “secure” is a significant challenge. In general, any system not designed to withstand an adversary is going to be broken easily by one, and system are not exception. If p2p systems are to be widely deployed on the Internet, they must be robust against a conspiracy of some nodes, acting in concert, to attack the remainder of the nodes. A malicious node might give erroneous responses to a request, both at the application level (returning false data to a query, perhaps in an attempt to censor the data) or at the network level (returning false routers, perhaps in an attempt to partition the network). Attackers might have a number of other goals, including traffic analysis against systems that try to provide anonymous communication, and censorship against systems that try to provide high availability.

In addition to such “hard” attacks, some users may simply wish to gain more from the network than they give back to it. Such disparities could be expressed in terms of disk space (where an attacker wants to store more data on p2p nodes than is allowed on the attacker’s home node), or in terms of bandwidth (where an attacker refuses to use its limited network bandwidth to transmit a file, forcing the requester to use some other replications). While many p2p applications are explicitly designed to spread load across nodes, “hot-spots” can still occur, particularly if one node is responsible for a particularly popular document.

Furthermore, a number of “trust” issues occur in p2p networks. As new p2p applications are designed, the code for them must be deployed. In current p2p systems, the code to implement the p2p system must be trusted to operate correctly; p2p servers typically execute with full privileges to access the network and hard disk. If arbitrary users are to create code to run on p2p systems, an architecture to safely execute untreated code must be deployed. Likewise, the data being shared, itself, might not be trustworthy. Popularity based ranking systems will be necessary to help users discover documents that they desire.

There are some security requirements to p2p systems:

- Censorship resistance—mechanisms to allow/prevent download of specific content;
User: How do I ensure possibility to download “any” content without being censored?
Law enforcement bodies: How do we detect/block download of specific content?
- Anonymity—How do I hide myself? Onion routing(gnunet/TOR);
- Content integrity—How ensure that rubbish (spam) is not distributed, also spy-ware and viruses, mechanisms;
- Availability—replication/popularity of given file, but also technical solutions to add redundancy: information dispersal, error correction codes;

- Robustness—relates to availability and also to sensor ship resistance.

In our paper, we focus on one of these requirements, namely, availability. We will analyze some technical solutions to add redundancy.

2.3 Redundancy

Many approaches have been used or proposed for providing security for information dissemination over networks, including encryption, authentication, and digital signatures. These mechanisms do not, however, necessarily help ensure that a message is delivered at all. Attacks that try to destroy or intercept security messages require other mechanisms.

Encryption can provide secrecy, authentication can provide assurance of the source, digital signatures can provide integrity verification, firewalls can filter out dangerous transmissions, and so on. But these and other traditional mechanisms offer little assistance with interruption threats. No matter how elaborate the encryption or authentication, if the information is dropped on the floor, destroyed or transformed into a piece of garbage, blocked because of the overloading of an intermediate link, or disrupted by other malicious acts, information availability is damaged. In many cases, attackers can achieve their ends merely by ensuring that important information does not reach its destination, even if they cannot decrypt it, forge it, or alter it.

The traditional solution is to require acknowledgement of important messages. Since attackers might try to forge acknowledgements, they are typically signed (and possibly encrypted, if they contain sensitive information). If an acknowledgement is not received soon enough, the message is resent. This method works well if a relatively small number of message require acknowledgement. If a very large number of messages must be acknowledged, then hierarchical or other load distribution

methods must spread out the responsibility for checking the acknowledgements. In the general case, all nodes performing the checks must be trusted.

A further problem is that an attacker can repeatedly intercept or destroy the retransmitted message. Without other mechanisms, an attacker who has compromised a single link or router node may permanently prevent the delivery of the message, since each retransmission will probably still follow the same path through the compromised resource.

The problem is that there is only a single path for information transmission. If any point of this single path is corrupted, transmission security is corrupted. This problem can be reduced by adding redundancy to information transmission structures. Such redundancy can improve transmission resiliency and greatly improve the availability and other aspects of security. Typically such redundancy can be provided by using more than one path through the network to reach the destination.

If the redundant paths are completely disjoint, the attackers must compromise multiple resources in the network to prevent message delivery. The greater the degree of redundancy, the more resources they must compromise. Assuming that there is cost and risk in compromising each resource, increasing the degree of redundancy can thus increase the difficulty of preventing successful delivery.

Redundancy uses more resources than single-path transmission. Thus, there is a tradeoff between the degree of security achieved and the cost of providing it.

Similar arguments have demonstrated the value of redundancy for many hardware fault tolerance problems. In the networking realm, however, actually providing true redundancy may be difficult. While two distinct disks can be used for storing the same data, or two distinct processors can be loaded with the same instructions, it is not always true that two or more disjoint paths can be easily found for reaching a specific

destination through a network. Such paths might not exist. Even if they do, existing network routing protocols and the desire to hide network complexities from higher levels make discovering and using the disjoint paths difficult. And it is even more difficult to know where those physical lines that a message follows are.

Nonetheless, redundancy can have some value. Even if the paths are not fully disjoint, any non-shared portions of the path limit an attacker's choice of attack points. The attacker must either find and compromise shared links or routers on the path, or must compromise the right set of non-shared elements. The volatility and obscurity that makes finding disjoint paths difficult also makes attacking them hard. While some choke points cannot be avoided, link-by-link (or segment-by-segment) redundancy may still prove very useful.

2.4 Error correction codes

Error correction has the feature that enables localization of the errors and correcting them. Given the goal of error correction, the idea of error detection may seem to be insufficient. However, error correction schemes may be computationally intensive, or require excessive redundant data which may be inhibitive for a certain application. Error correction in some applications, such as a sender-receiver system, can be achieved with only a detection system in tandem with an automatic repeat request scheme to notify the sender that a portion of the data sent was received incorrectly and will need to be retransmitted, however where deficiency is important, it is possible to detect and correct errors with far less redundant data.

In information theory and coding, an error correcting code or ECC is a code in which each data signal conforms to specific rules of construction so that departures from this construction in the received signal can generally be automatically detected and corrected.

Error correction coding is aim to detect and locate errors in transmission. Once located, the correction is trivial: the bit is inverted. Error correction coding requires lower rate codes than error detection. It is therefore uncommon in terrestrial communication, where better performance is usually obtained with error detection and retransmission. However, in satellite communications, the propagation delay often means task of data handling very complex. Real-time transmission often precludes retransmission. It is necessary to get it right first time. In these special circumstances, the additional bandwidth required for the redundant check-bits is an acceptable price.

Error correction coding is used through the internet, use in computer data storage, for example in dynamic RAM, and in data transmission, such as deep space telecommunication, satellite broadcasting and so on. Examples include Hamming code, Reed-Solomon code, Reed- Muller code, Binary Golay code, convolution code, turbo code and others. The simplest error correcting codes can correct single bit error (single error correction) and detect double-bit errors (double error detection). Other codes can detect or correct multi-bit errors.

Shannon's theorem is an important theory in error correction which describes the maximum attainable efficiency of an error correcting scheme versus the levels of noise interference expected.

3 Use of ECC in different P2P systems

The problem of efficiently retrieving a file that has been broken into blocks and distributed across the wide-area pervades applications that utilize peer-to-peer, and distributed file systems. In this section, we use Error Correction Codes to improve the fault-tolerance and performance of different file systems. However, despite the advantage of ECC, not all P2P file sharing systems implement ECC, for example, Bit

Torrent, Gnutnet, i2p, etc.

In this section, we describe the basic principle of using ECC in some P2P systems, and then compare Bit Torrent with Avalanche. We explain the reason why Bit Torrent doesn't use ECC, while Avalanche use and analyze their advantage and disadvantage, respectively. Finally we present OceanStore, a P2P file storage system, who implements ECC to retrieve original files.

3.1 Principle

P2P networks provide scalability for the file sharing applications they underlie. A P2P system uses a peering architecture that offers the support for various P2P services such as file sharing between the various peers. Examples of such system are Napster, Gnutella, eDonkey, emule, Bit Torrent, or Kazaa, etc. Using these systems, each node is able to determine its peers and to localize data over the peer network. P2P file sharing systems account for a high percentage of the traffic volume in the fixed Internet, having exceeded http (WWW) or email traffic.

P2P files sharing systems have specific characteristics compared to classical distributed storage systems. A major concern is the potentially large number of peers which are equal (e.g., up to 1.5 million users online at any time for Kazaa System in May 2002). Users install a P2P file sharing software which enables the users' computer to connect other computer who installs the same software. In this way, both the users are able to share their recording or audiovisual works. When a transaction is complete, the user has an identical copy of the file on his or her computer and may also then disseminate the file to other users connected to the network [1].

ECC can also improve the reliability of P2P storage systems by encoding the data, splitting up the encoded data and distributing the fragments over various servers [9].

The basic idea is that a redundant fragment located on a server can compensate for the loss of any other fragment due to the failure of another server. From a fault-tolerance point of view, the use of EEC refers to “mean time of failures by many orders of magnitude compared to replication systems with similar storage and bandwidth requirements” [10]. A user receives encoded parts of the data from several servers and is able to reconstruct the data as soon as it obtains a sufficient number of different packets. However, this solution can only be applied in particular contexts (such as software distribution) and presents several drawbacks such as, for example, the ending of the connections or the management of the network congestions [10].

3.2 Bit Torrent

3.2.1 How BitTorrent works

Bit Torrent is both the name of a peer-to-peer (P2P) file distribution client application and also the name of the file sharing protocol itself, both of which were created by programmer Bram Cohen. Bit Torrent is designed to widely distribute large amounts of data without incurring the corresponding consumption in costly server and bandwidth resources [1]. Bit Torrent is one of the few that has managed to attract millions of users.

According to the Bit Torrent protocol, each Bit Torrent clients are capable of preparing, requesting and transmitting files over a network. Files can contain any type of digital information, including text, audio or encrypted content. Clients first create a “torrent” which is called “seed”. The torrent files contain tracker and file information. The former specifies the URL of the tracker and the latter contains a suggested name for the file, fragment size, a key length, file length and a pass. Users firstly get the torrent files and open it in a Bit Torrent client. After opening the torrent, the Bit Torrent client connects to the tracker. A group of peers on a Bit Torrent or P2P are connected with each other to share a particular torrent. In this process, none server is involved. The

advantage of Bit Torrent is the more users download, the more seed is, so the faster the rate of download.

3.2.2 Why not using error-correcting codes

Despite the above advantage of Bit Torrent, one of drawback is that the last seed goes offline while there are still several peers online who don't complete download file. Attempts by the peers to reconstruct the original file are rarely successful, since the chances of every single block being present on one of the seeds are generally very low. It's likely that at least one block is missing.

In principle, applying error correction codes to Bit Torrent could resolve the above problem. If there's no seed left error correction codes increase the chances that the entire file will be recoverable. Take erasure codes as an example, in practice, when a file becomes unrecoverable it's because there was only one seed and several downloaders started from scratch, then the seed disappeared after uploading less than the total length of the file. Erasure codes obviously would not help out in that case. On the other hand, using error correction codes will affect stability of the Internet and degrade performance of the system.

3.3 Avalanche

3.3.1 How Avalanche works

Avalanche, a Bit Torrent-like P2P distribution project Microsoft is currently working on [3]. Based on Bit Torrent, Avalanche is also working to tackle above problems with Bit Torrent.

The researchers claim download times are between 20-30 per cent faster, using their

network coding approach, than on systems that only code at server, and between 200 and 300 per cent faster than distributing un-encoded information. The principle behind network coding is to allow intermediate nodes to encode packets by using local information. “The only code at server” refers to FEC (Forward Error Correction).

With network coding, each node of the distribution network is able to generate and transmit encoded blocks of information. One important feature of network coding is that all nodes are able to finish their download even if there is no seed. For instance, it is assumed that there are 500 nodes in a p2p system. The server leaves the system after serving only 5% extra blocks. Only 40% of the nodes finished downloading by using Bit Torrent, however, all 500 nodes can complete the download when using Avalanche.

3.3.2 Using error correction codes

According to Microsoft, Avalanche can re-create missing blocks of data that can be used in place of missing chunks. Once one peer has downloaded a few of these, it can generate new combinations from the ones it has, and send those out to other peers. Collect enough of these pieces and the peer will have enough information to reconstruct the entire file. Even if one of peers doesn't have all the original pieces distributed by the peer who held the original version of the file, it can still get the whole file [2]. Peers can make use of any new piece, instead of having to wait for specific chunks that are missing. This means that no one peer can become a bottle neck, since no piece is more important than any other. It also means overall network traffic is lower, since the same information doesn't have to travel back and forth multiple times.

However, Bram Cohen, the father of Bit Torrent, is unimpressed with Avalanche [5]. He claimed that Avalanche is vaporware on his blog. Cohen doesn't believe that

results from a simulation could possibly match up to results of a real test due to real world Internet behavior. He thought Microsoft research didn't simulate varying transfer rate abilities, transfer rate abilities varying over time, or endgame mode. Potential issues are on the wire overhead, CPU usage, memory usage, and disk access time. What is more, the central problem is disk access, he believed. If the size of the file being transferred is greater than the size of memory, their entire system could easily get bogged down doing disk seeks and reads, since it needs to do constant recombination of the entire file to build the pieces to be sent over the wire [6].

On his blog, he mentioned another big problem with error correction. Peers can't verify data with a secure hash before they pass it on to other peers. As a result, it's quite straightforward for a malicious peer to poison an entire swarm just by uploading a little bit of data.

3.4 OceanStore

The coordination of widely distributed file servers is a complex problem that challenges wide-area, peer-to-peer and Grid file systems. Systems like OceanStore and Bit Torrent aggregate wide-area collections of storage servers to store files on the wide-area. OceanStore is an Internet-scale, persistent data store designed for incremental scalability, secure sharing, and long-term durability.

OceanStore, an Internet P2P file storage system, places a high premium on identifying the validity of distributed data. There are two states of data in OceanStore: active and archival. The active data means current state of data and can be modified. On the other hand, the archival data store the past state and can be read, but not modified.

In particular, the archival data utilizes erasure codes to fragment and reconstruct data.

Data that has been fragmented and distributed among multiple servers in this scheme can be recovered from as few as one quarter of the fragments, said Kubiawicz. In this case, the chance of losing data will decrease sharply. However, rather than relying on redundancy in the data fragments to detect and correct a corrupted fragment, which may take factorial time in the event of error, OceanStore introduces a system for incorporating verification information of the complete data, fragment and fragment name into each block [8].

OceanStore uses a cryptographic hash of each fragment to create a unique name. The hashes of all the fragments in a block are repeatedly concatenated and re-hashed, until a single hash is created. The results are tagged with global unique identifiers (GUIDs), split in pieces and dispersed among servers in different geographical areas. This dispersed network of servers will be regarded as 'untrusted', meaning they will be able to read the GUID tags, but not the underlying data. Once users search for data, OceanStore will look for the fragment in given vicinity and reconstruct the original data. If the search fails, a wider hierarchical search will begin.

4 Two different error-correction codes

Error Correction Codes mechanisms provide transport protocols with reliable delivery of content. In our project, we simulate two of EEC codes, Reed-Solomon Codes and Tornado Codes.

4.1 Reed-Solomon Codes

4.1.1 Introduction

Reed-Solomon codes are block-based error correcting codes with a wide range of

applications in digital communications and storage. Reed-Solomon codes are used to correct errors in many systems including:

- Storage devices (including tape, Compact Disk, DVD, barcodes, etc)
 - Wireless or mobile communications (including cellular telephones, microwave links, etc)
 - Satellite communications
 - Digital television / DVB
 - High-speed modems such as ADSL, xDSL, etc.
- A typical system is shown here:



The Reed-Solomon encoder takes a block of digital data and adds extra "redundant" bits. Errors occur during transmission or storage for a number of reasons (for example noise or interference, scratches on a CD, etc). The Reed-Solomon decoder processes each block and attempts to correct errors and recover the original data. The number and type of errors that can be corrected depends on the characteristics of the Reed-Solomon code.

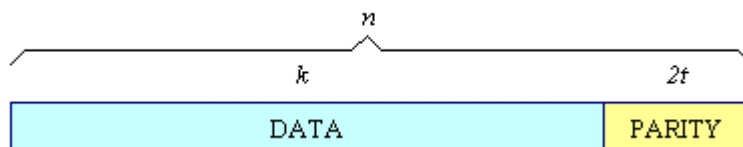
4.1.2 Properties of Reed-Solomon codes

Reed Solomon codes are a subset of BCH codes and are linear block codes. A Reed-Solomon code is specified as $RS(n,k)$ with s -bit symbols.

This means that the encoder takes k data symbols of s bits each and adds parity symbols to make an n symbol codeword. There are $n-k$ parity symbols of s bits each.

A Reed-Solomon decoder can correct up to t symbols that contain errors in a codeword, where $2t = n - k$.

The following diagram shows a typical Reed-Solomon codeword (this is known as a Systematic code because the data is left unchanged and the parity symbols are appended):



Example: A popular Reed-Solomon code is RS(255,223) with 8-bit symbols. Each codeword contains 255 code word bytes, of which 223 bytes are data and 32 bytes are parity. For this code:

$$n = 255, k = 223, s = 8$$

$$2t = 32, t = 16$$

The decoder can correct any 16 symbol errors in the code word: i.e. errors in up to 16 bytes anywhere in the codeword can be automatically corrected.

Given a symbol size s , the maximum codeword length (n) for a Reed-Solomon code is

$$n = 2s - 1$$

For example, the maximum length of a code with 8-bit symbols ($s=8$) is 255 bytes.

Symbol Errors

One symbol error occurs when 1 bit in a symbol is wrong or when all the bits in a symbol are wrong.

Example: RS(255,223) can correct 16 symbol errors. In the worst case, 16 bit errors may occur, each in a separate symbol (byte) so that the decoder corrects 16 bit errors. In the best case, 16 complete byte errors occur so that the decoder corrects 16×8 bit

errors.

Reed-Solomon codes are particularly well suited to correcting burst errors (where a series of bits in the codeword are received in error).

Decoding

Reed-Solomon algebraic decoding procedures can correct errors and erasures. An erasure occurs when the position of an erred symbol is known. A decoder can correct up to t errors or up to $2t$ erasures. Erasure information can often be supplied by the demodulator in a digital communication system, i.e. the demodulator "flags" received symbols that are likely to contain errors.

When a codeword is decoded, there are three possible outcomes:

(1). If $2s + r < 2t$ (s errors, r erasures) then the original transmitted code word will always be recovered,

Otherwise:

(2). The decoder will detect that it cannot recover the original code word and indicate this fact.

Or:

(3). The decoder will mis-decode and recover an incorrect code word without any indication.

The probability of each of the three possibilities depends on the particular Reed-Solomon code and on the number and distribution of errors.

Coding Gain

The advantage of using Reed-Solomon codes is that the probability of an error remaining in the decoded data is (usually) much lower than the probability of an error if Reed-Solomon is not used. This is often described as coding gain.

Example: A digital communication system is designed to operate at a Bit Error Ratio (BER) of 10^{-9} , i.e. no more than 1 in 10^9 bits are received in error. This can be

achieved by boosting the power of the transmitter or by adding Reed-Solomon (or another type of Forward Error Correction). Reed-Solomon allows the system to achieve this target BER with a lower transmitter output power. The power "saving" given by Reed-Solomon (in decibels) is the coding gain.

4.1.3 Architectures for encoding and decoding Reed-Solomon codes

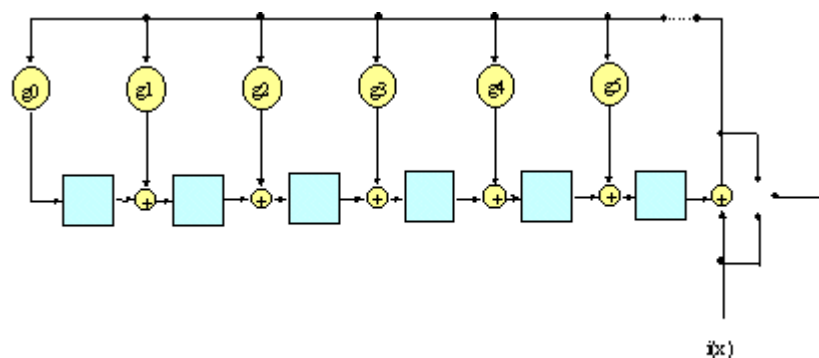
Reed-Solomon encoding and decoding can be carried out in software or in special-purpose hardware.

(1) Encoder architecture

The $2t$ parity symbols in a systematic Reed-Solomon codeword are given by:

$$p(x) = i(x) \cdot x^{n-k} \bmod g(x)$$

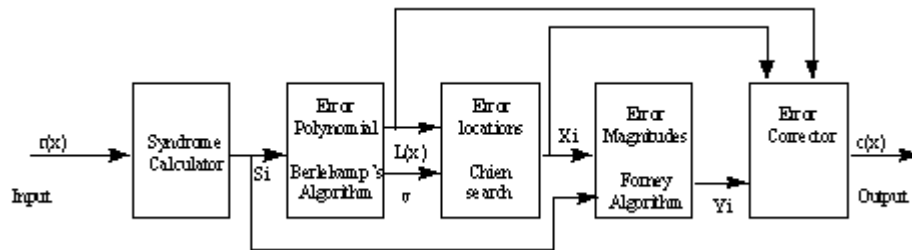
The following diagram shows an architecture for a systematic RS(255,249) encoder:



Each of the 6 registers holds a symbol (8 bits). The arithmetic operators carry out finite field addition or multiplication on a complete symbol.

(2) Decoder architecture

A general architecture for decoding Reed-Solomon codes is shown in the following diagram.



Key

$r(x)$ Received codeword

S_i Syndromes

$L(x)$ Error locator polynomial

X_i Error locations

Y_i Error magnitudes

$c(x)$ Recovered code word

v Number of errors

The received codeword $r(x)$ is the original (transmitted) codeword $c(x)$ plus errors: $r(x) = c(x) + e(x)$

A Reed-Solomon decoder attempts to identify the position and magnitude of up to t errors (or $2t$ erasures) and to correct the errors or erasures.

4.1.4 Implementation of Reed-Solomon encoders and decoders

(1) Hardware Implementation

A number of commercial hardware implementations exist. Many existing systems use "off-the-shelf" integrated circuits that encode and decode Reed-Solomon codes. These ICs tend to support a certain amount of programmability (for example, RS(255,k) where $t = 1$ to 16 symbols). A recent trend is towards VHDL or Verilog designs (logic cores or intellectual property cores). These have a number of

advantages over standard ICs. A logic core can be integrated with other VHDL or Verilog components and synthesized to an FPGA (Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit) – this enables so-called "System on Chip" designs where multiple modules can be combined in a single IC. Depending on production volumes, logic cores can often give significantly lower system costs than "standard" ICs. By using logic cores, a designer avoids the potential need to do a "lifetime buy" of a Reed-Solomon IC.

(2) Software Implementation

Until recently, software implementations in "real-time" required too much computational power for all but the simplest of Reed-Solomon codes. The major difficulty in implementing Reed-Solomon codes in software is that general purpose processors do not support Galois field arithmetic operations. For example, to implement a Galois field multiply in software requires a test for 0, two log table look-ups, modulo add and anti-log table look-up. However, careful design together with increases in processor performance means that software implementations can operate at relatively high data rates. The following table gives some example benchmark figures on a 166MHz Pentium PC:

Code	Data rate
RS(255,251)	12 Mbps
RS(255,239)	2.7 Mbps
RS(255,223)	1.1 Mbps

These data rates are for worst case decoding (correcting the maximum number of errors in "every" code word): encoding is considerably faster since it requires less computation. [11]

4.2 Tornado Codes

4.2.1 Overview of Tornado Codes

Tornado codes are a family of data encodings which are efficient to encode and decode and allow receivers to reconstruct source data when any $(1+\epsilon)k$ packets are received (where there are k original source packets).

The advantages of Tornado codes are that they provide an efficient approximation to an ideal digital fountain where any k packets from an infinite number of distinct encoding packets can be used to retrieve the original k packets of source data. Specifically, Tornado Z encoding is compared to Solomon-Reed encoding which takes quadratic time in the length of the source data to encode or decode. While Tornado encoding introduces some decoding inefficiency ($(1+\epsilon)k$ packets are needed to reconstruct source data), the run-time of encoding and decoding of Tornado make it much more feasible for large file distribution schemes. When coupled with multicast, a digital fountain provides a channel to which clients can tune-in and with only slightly more packets than the uuencoded source would have required can download the source file, regardless of which specific packets are retrieved.

This model introduces packet distinctiveness problems in real-world implementations (where there aren't an infinite number of distinct encoding packets and useless duplicates may be received). Tornado is shown to outperform alternate digital fountain approximations in these settings also either when holding decoding inefficiency constant across both system and measuring decoding time and when holding decoding time constant and measuring decoding inefficiency.

Tornado is also shown applied to a many-to-many file distribution system (parallel downloading from mirrors). In such a system mirrors all provide a multicast channel broadcasting Tornado encoded packets. Peers receive these packets from several mirrors. While not scaling linearly (optimally) with the number of parallel mirrors (due to packet distinctiveness problems and Tornado decoding inefficiency), the speedup is still very significant.

Tornado proposes a tradeoff between de/encoding cost and decoding efficiency (number of packets required to retrieve the original k source packets). If such a trade-off is an inherent problem it seems that some model could theoretically show the trade-off. Since the contribution of Tornado is an encoding which makes a good compromise in this trade-off it's critical that such a trade-off exist.

4.2.2 Properties of Tornado code

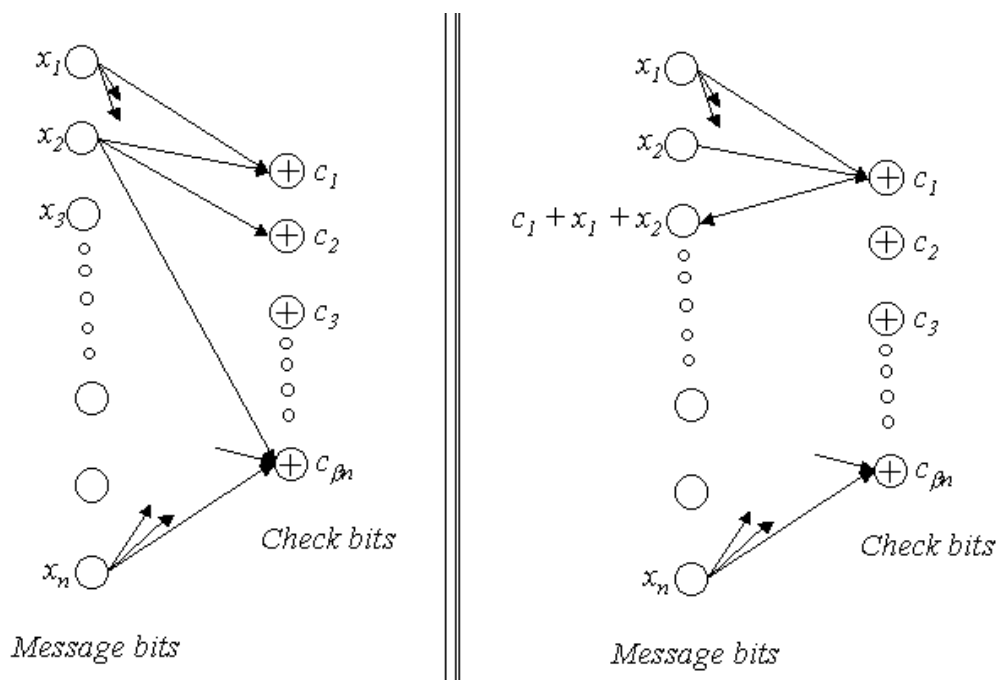
- A FEC code (*erasure code*) that works by constructing successive sets of bipartite graphs, where the RHS of each graph consists of the selective XOR of certain elements on the LHS of that graph. (The LHS of the leftmost bipartite graph is the source data.)
- Stretch factor $c=n/k$ measures number of total packets n needed to redundantly encode k source packets. This paper chooses $l=k$ redundancy pkts, hence $n=k+l=2k$.
- Encoding/decoding time is $O((l+k)P)$ for a P -sized packet. Compare to traditional erasure codes (such as Reed-Solomon) which are typically $O(lkP)$ and rely on more complex operations than XOR; in those codes, only practical for small k and l , whereas Tornado codes are fine with k, l on the order of 10^4 .
- One approach is interleaving combined with traditional codes, with interleaving factor K . Problem: want small K to keep decoding fast, but smaller K means receiver may have to receive a lot more packets to reconstruct data, since

contribution of each arriving packet to an interleaving "bin" has a non-uniform distribution. Also, in general, #packets required to reconstruct interleaved data grows super linearly with orig data size; for Tornado codes, grows only linearly.

- Reception overhead is e if $(1+e)k$ packets must be received to reconstruct orig data. For $n=2k$, $e=1$. [12]

4.2.3 How does Tornado Codes work

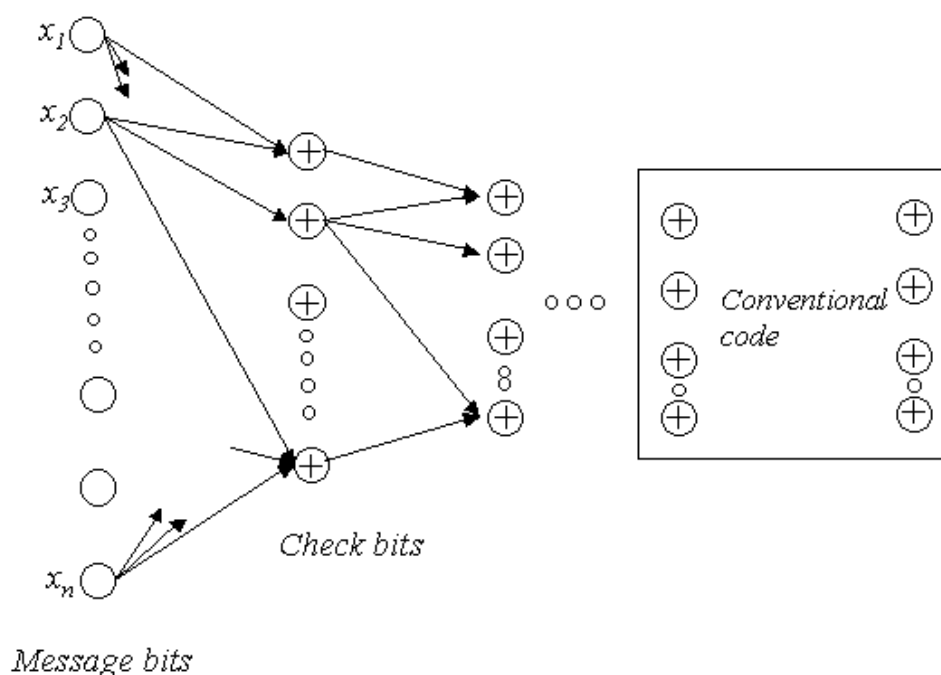
A code $C(B)$ is defined as a set of n message bits and b n check bits that are associated with a bipartite graph B . The graph B has n left nodes and b n right nodes, corresponding to the message bits and the check bits respectively. The encoding consists of computing each check bit as the sum of the bits of its neighbors in B . The encoding time is proportional to the number of edges in B . If the value of a check bit is known and all but one of the message bits associated with that check bit is known, the missing message bit can be found by computing the sum of the check bit and its known message bits.



Graph representation of one stage of the Tornado Code.

Encoding and decoding are simple XOR operations on bits or packets. The total decoding time is proportional to the number of edges in the graph. From the above diagram it is clear that at most one message bit that contributes to a particular check bit or the check bit itself, can be lost if the code is to be recovered. To produce codes that are more loss resilient, the basic structure is cascaded to form the cascaded code C(B). The first structure produces $b \cdot n$ check bits for the original n message bits and at the next stage a similar structure is used to produce $b^2 \cdot n$ check bits for the $b \cdot n$ check bits. At the final stage a conventional loss resilient code such as Reed- Solomon or BCH code is used.

The final code would resemble the following diagram:



Graph representation of the complete code.

The number of stages and the basic structure of the bipartite graph involve a very complicated design procedure based on differential equations that are beyond the scope of this project. An implementation of the Tornado code developed by Jeffrey

Considine was used to explore the possibility of the use of these codes in data synchronization. [13]

5. Implementation

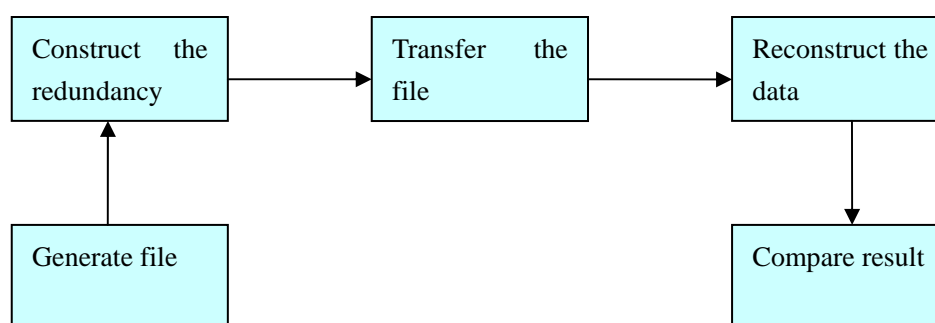
In order to prove the algorithms we mention before, we have implemented the two algorithms, Reed-Solomon codes and Tornado codes respectively.

We decide to use MATLAB to implement the coding techniques .Because the error-correction codes concern about the digital communications.

We will in this chapter explain the problems associated with each of the code techniques during the development, how we solved it and how the final implementation was done. We will look at the different algorithms in separate sections.

The two algorithms use the same flow chart but different designs in detail.

Flow chart as follow:



5.1 Implementation with Reed-Solomon Codes

5.1.1 Generate the files

In our program, we use 4 original file pieces. There are 1000 words with length 4 in each piece. The source codes are as follows:

```
D1=round(rand(n,h*w));
```

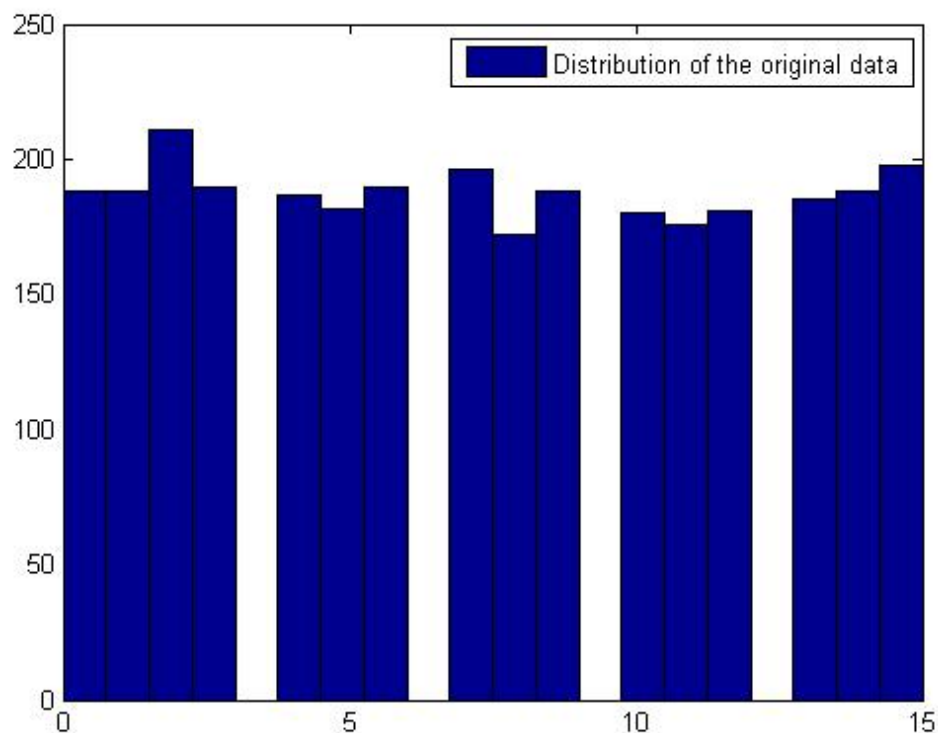


Figure 1

The word length is 4, so the data are in $[0,15]$, and the distribution of the data is showed in the above graph.

5.1.2 Construct the redundancy

We use 4 original file pieces to construct the 3 redundancy pieces.

$$\mathbf{FD} = \mathbf{C}$$

We define \mathbf{F} to be the 3 x 4 Vander monde matrix: $f_{i,j} = j^{i-1}$

$$\begin{bmatrix} 1 & 1 & 1 & \dots & 1 \\ 1 & 2 & 3 & \dots & n \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & 2^{m-1} & 3^{m-1} & \dots & n^{m-1} \end{bmatrix} \begin{bmatrix} d_1 \\ d_2 \\ \vdots \\ d_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_m \end{bmatrix}$$

The source codes are as follows:

```
F=zeros(m,n);
```

```
for ii=1:m
```

```
    for jj=1:n
```

```
        A(ii,jj)=jj.^(ii-1);
```

```
    end
```

```
end
```

We define the matrix **A** and the vector **E** as follows:

$$A = \begin{bmatrix} I \\ F \end{bmatrix} \quad E = \begin{bmatrix} D \\ C \end{bmatrix}$$

$$\mathbf{AD} = \mathbf{E}$$

And then we put the each element of in E different nodes.

5.1.3 Transfer the files

In this Peer to peer system, each node transfers its data. But in the real channel, some errors will occur randomly. Tolerance is no more than 3, so there are at least 2 errors occurring in the original data, and the rest occur in the redundancy part.

The main source codes are as follows:

```
error_num=round(rand*(m-2)+2);
```

```
error_array=floor(rand(error_num,h)*16);
```

```
error_index=zeros(1,error_num);
```

```
for ii=1:error_num-2
```

```
    error_index(ii)=round(rand*m+n);
```

end

Data with errors are showed in the following figure

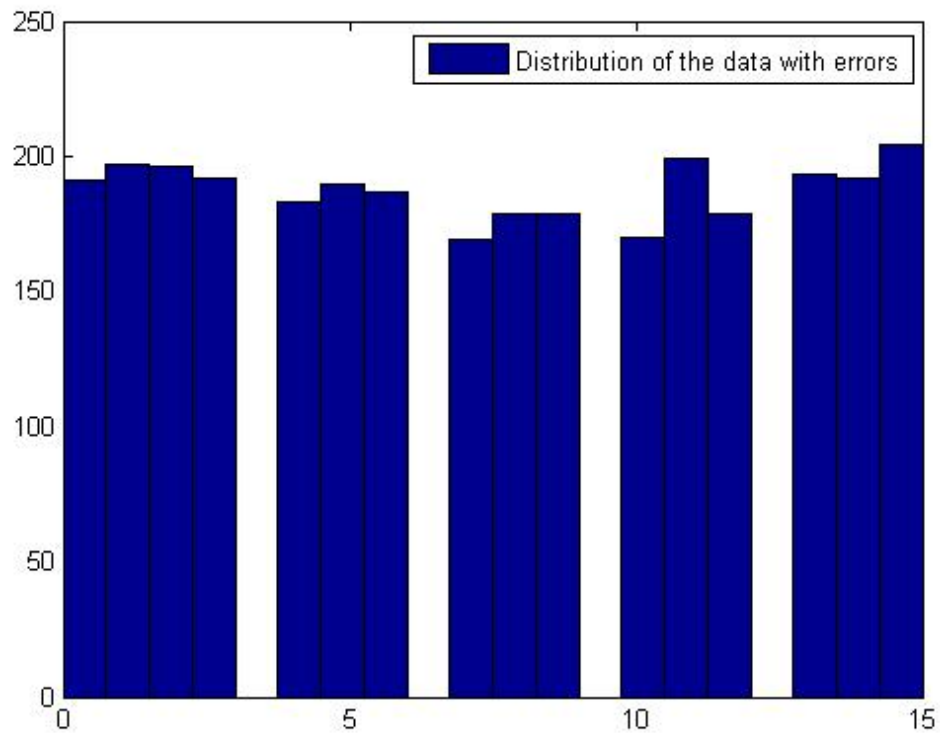


Figure 2

In order to show the difference between the original data and the data with errors, we use another graph as follows which describe the different density distributions:

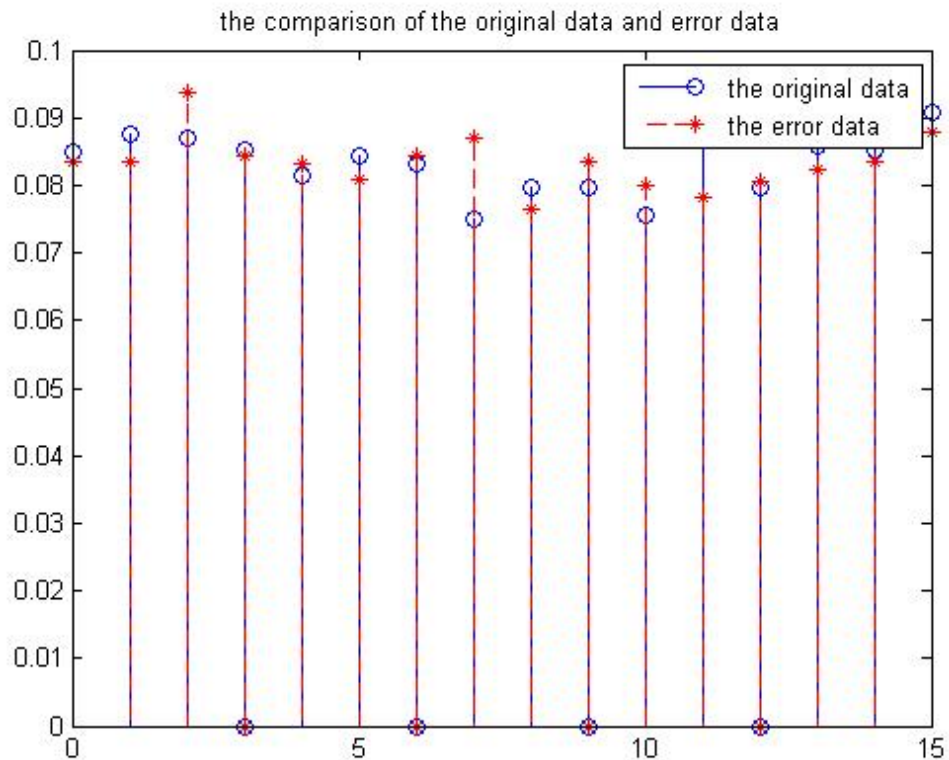


Figure 3

5.1.4 Reconstruct the data

In order to rebuild the original data, we first check which file piece is correct and which one is wrong. If any number of devices up to 3 fails, then they can be restored in the following manner. And then we use the 4 right pieces to rebuild the whole original data according to the formula below:

$$D = (A')^{-1} E'$$

The main source codes are as follows:

$D_restore = \text{inv}(A1) * E1;$

$E_restore = A * D_restore;$

5.1.5 Compare the result

At last, we compare the density of restored data with original data.

The main codes are as follows:

```
[num,error_rate]=symerr(D,D_restore);
```

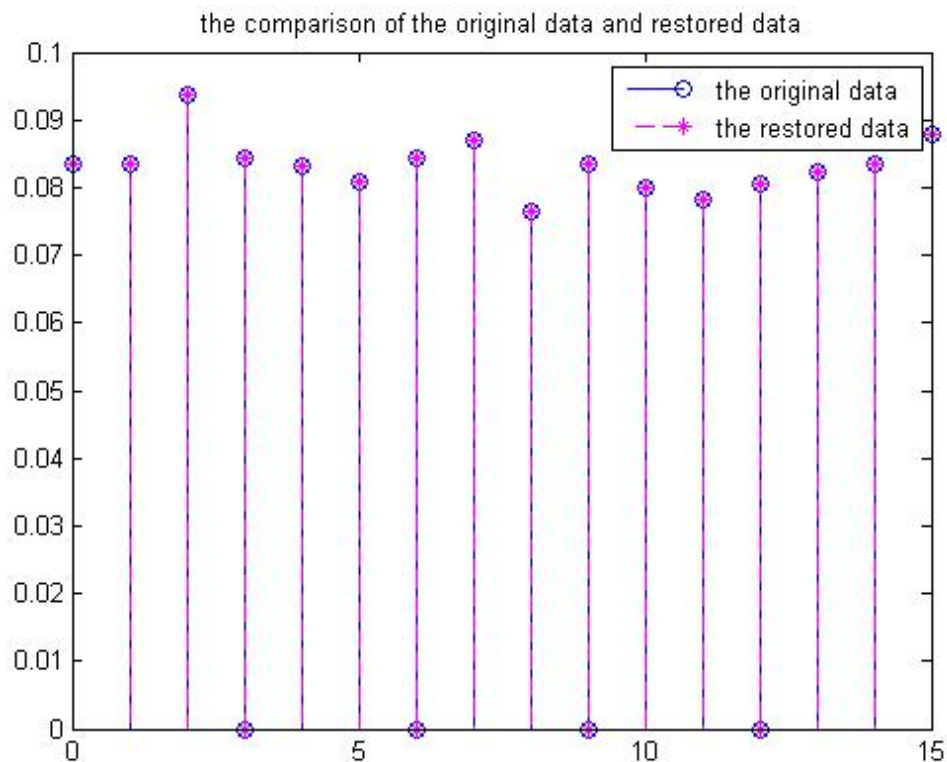


Figure 4

From the figure 4, we found the density distribution of the original data is the same as the restored data. There is no error at last, the algorithm performs well.

The whole programs are included in Appendix A

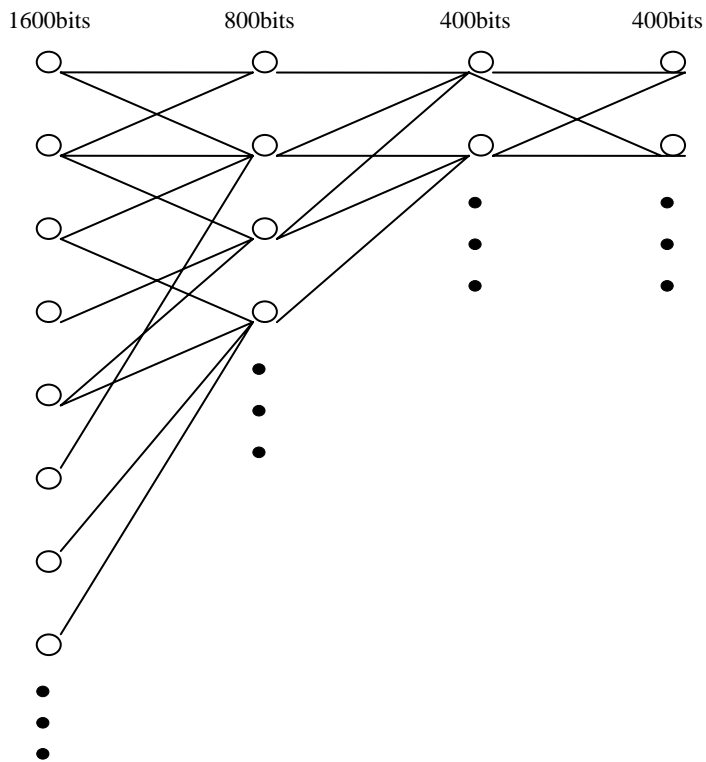
5.2 Implementation with Tornado Codes

5.2.1 Generate the file

There are 1600 bits in the original data, and we use 4 levels, the number of layers is

[1600, 800,400,400].

The structure is as follows:



The main codes are as follows:

`data_original = round(rand(1,1600));`

the distribution are as follows:

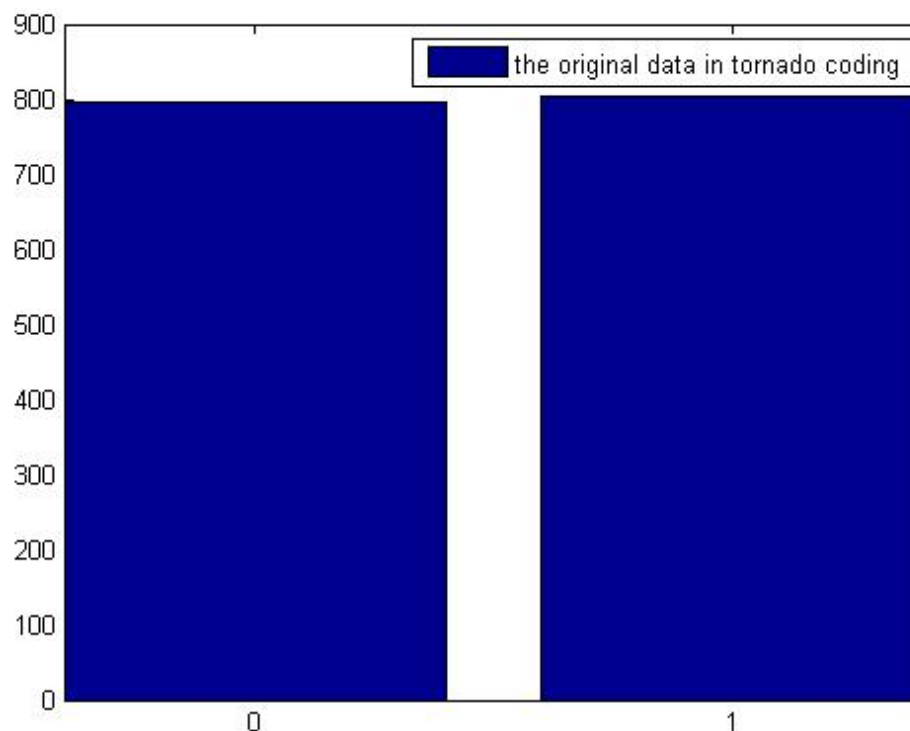


Figure 5

From the figure, we can see the bit 0 and bit 1 has almost the same number (around 800). And the data is binary, this is why there are only two columns.

5.2.2 Construct the redundancy

From the structure figure, we know the redundancy is calculated by **XOR** operator. Normally if a node collects n edges, we define the degree of the node as n . In order to have a good performance, the degree of the node from the right side should distribute as Poisson distribution. That means distribution of degree can be flexible.

The left degree sequence is described by the following truncated heavy tail distribution. Let $H(d) = \sum_{i=1}^d 1/i$ be the harmonic sum truncated at d . The average left degree α_l equals $H(d)(d+1)/d$. Recall that we require the average right degree, α_r to satisfy $\alpha_r = \alpha_l / \beta$. The right degree sequence is defined by the Poisson distribution with mean α_r . For all $i \geq 1$ the fraction of edges of degree i on the right equals:

$$\rho_i = \frac{e^{-\alpha} \alpha^{i-1}}{(i-1)!}$$

where α is chosen to guarantee that the average degree on the right is α_r . In other words, α satisfies $\alpha e^\alpha / (e^\alpha - 1) = \alpha_r$.

The D (maximum of degree) value here is 10;

According to the formulas above, we program like that:

```
d=1:D;
Hd=sum(1./d);
al=Hd*(D+1)./D;
beita=1./2;
ar=al./beita;
alf=6.4334;
i=2:D+1;
pdf=exp(-alf).*alf.^(i-1)./factorial(i-1);
pdf=pdf +(1-sum(pdf))./D;
```

The distribution of the degree is showed as follows:

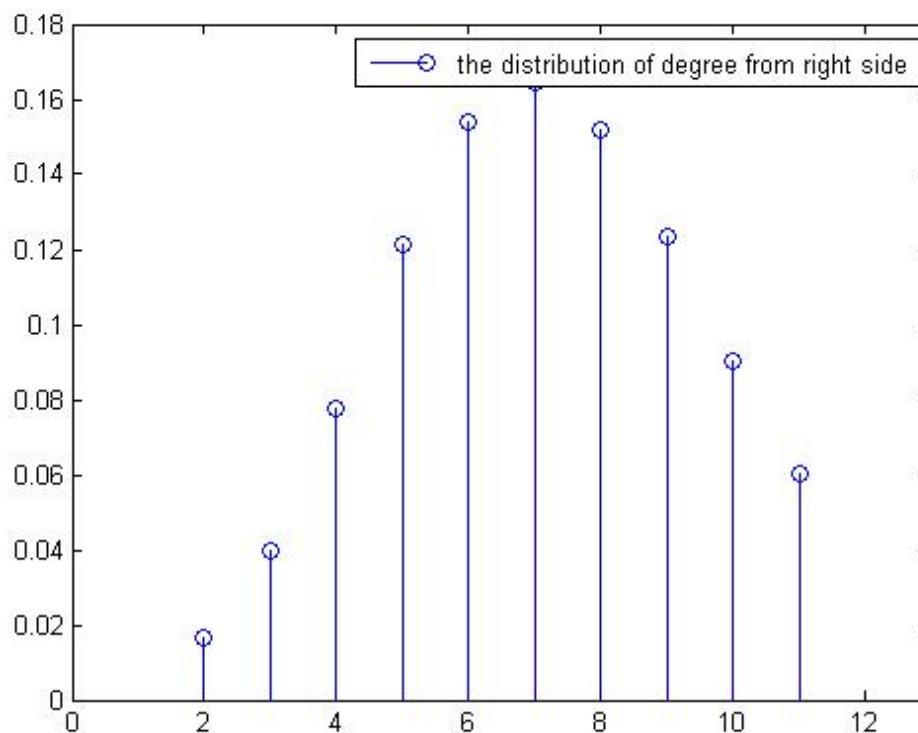


Figure 6

According to the degree distribution, we get the degree for each node and use **XOR** to get the redundancy .finally we put the redundancy, degree and corresponding

components from the left side into the database which we can use for decoding.

The main codes are as follows:

```
temp=[forming(degree(i,i1),i1,data(index(i-1)+1:index(i),4)',i);temp];
```

The following figure shows the distribution of data with redundancy.

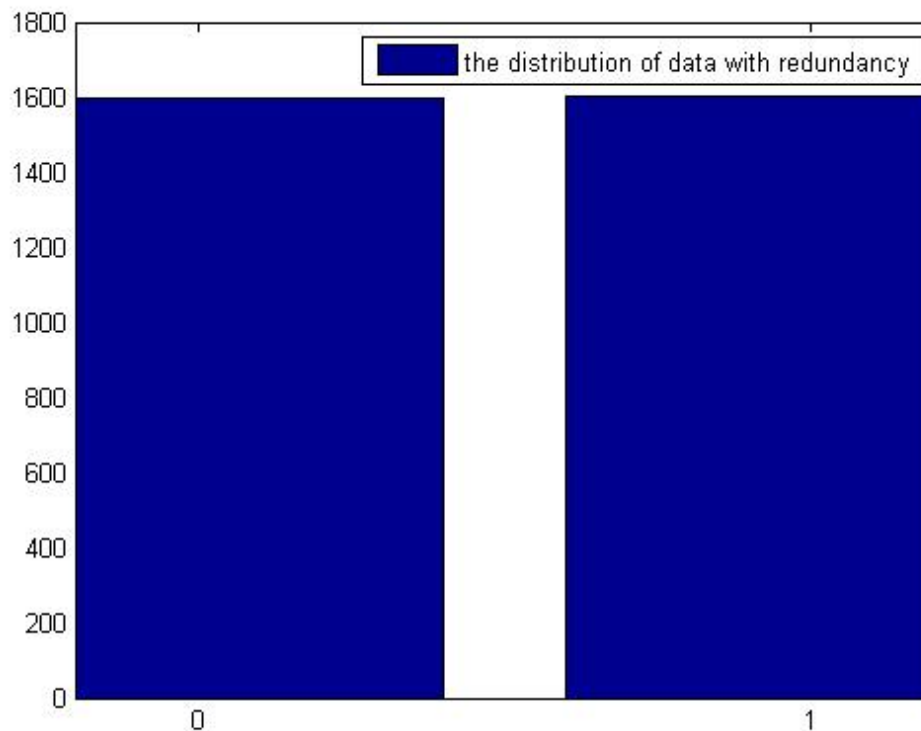


Figure 7

We can see the whole num of data with redundancy is 3200. And bit 0 and bit 1 are almost the same (around 1600).

5.2.3 Transfer the file

In theory, we say that the decoding inefficiency is $1 + c$, if $(1 + c)k$ (k is the number of original data) encoding packets are required to reconstruct the source data. In my program, I use $c=1./16$.

In order to simulate the channel, we receive the $(1 + c)k$ disordered bits. Our subprogram is as follows:

```
epxlong=100;
length_r= length(data_original)+epxlong;
for i=1:index(5)
    i
```

```

order=ceil(rand(1,2)*(index(5)-1));
a=order(1);
b=order(2);
c=data_m(a,:);
d=data_m(b,:);
data_m(b,:)=c;
data_m(a,:)=d;
end

data_r=data_m(1:length_r,:);

```

The distribution of the received data is showed in the following figure:

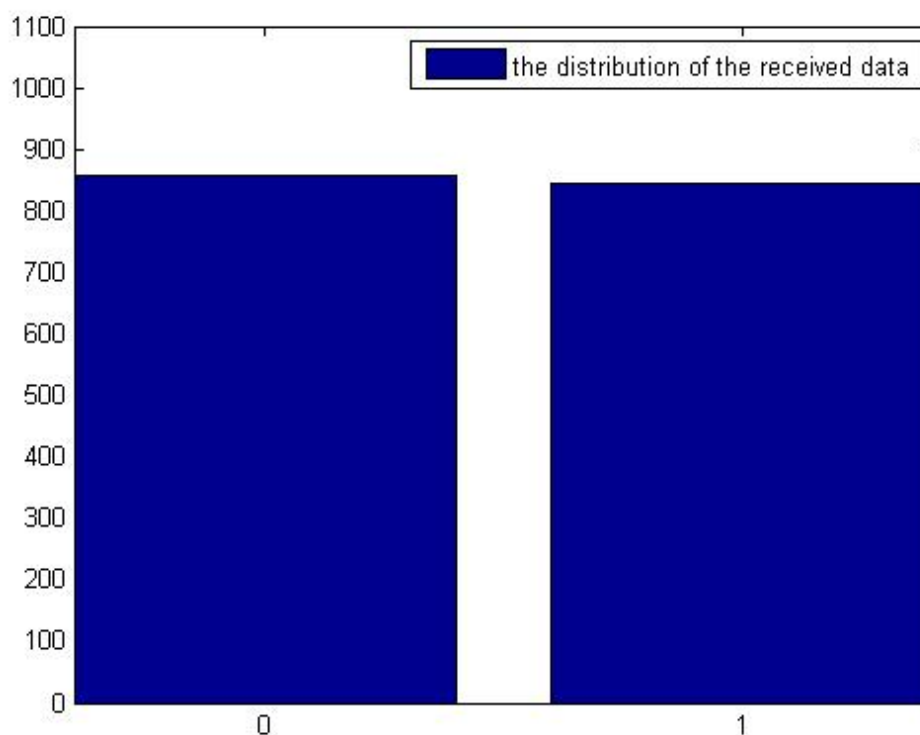


Figure 8

We can find that the total number is 1700 bits. And bits 0 and bits 1 are around 850 bits respectively.

5.2.4 Reconstruct the data

We decode the received data from the rightmost level to the leftmost level. First we find the node who has just miss one adjacent left node, and calculate the missed one with **XOR** operator. And then repeat the same operation on the node in the same layer.

The main part of source codes (take fourth layer as example) are as follows:

```

restore4=[];
for i=1:length(layer4)
    [mat,sign]=check_degree1(layer3,layer4(i,2),layer4(i,3),layer4(i,:));
    if sign==1
        for ii=index(3)+1:index(4)
            if mat(3)==data_c(ii,5)
                restore4=[restore4;[mat(2),data_c(ii,2),data_c(ii,3),mat(1),mat(3)]];
            end
        end
    end
end
end
end

```

The distribution of the restored data is as follows:

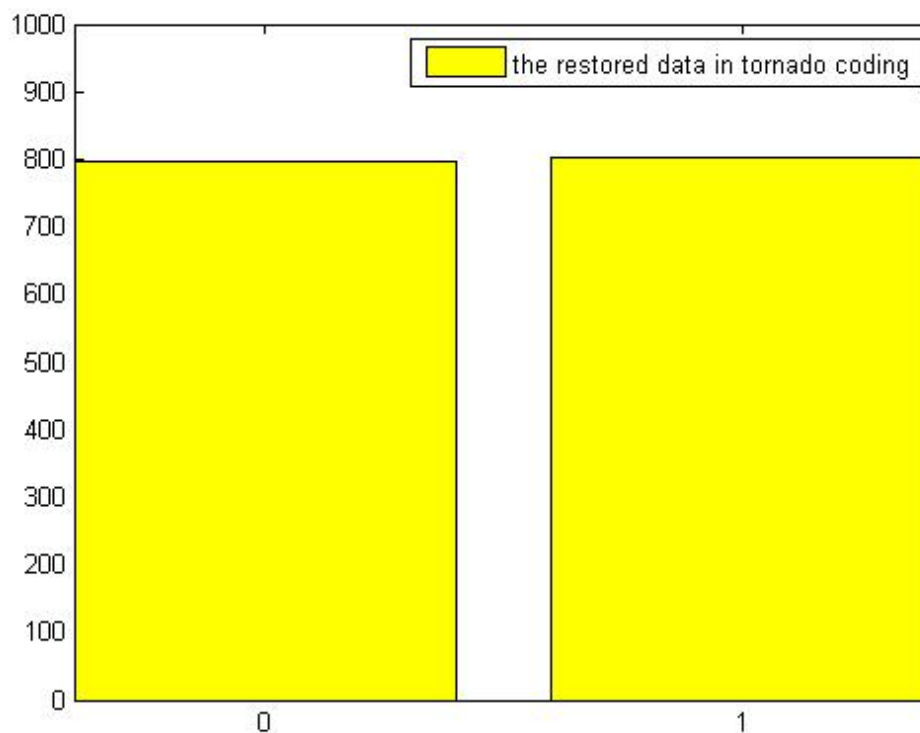


Figure 9

5.2.5 Compare the result

The above two figures show the comparison of the original data and the restored data. And they fit each other very well and the error rate is 0. So the performance is good.

The main program and subprogram are contained in the Appendix B.

6. Conclusion

In this paper, we showed with comparison among different P2P systems and simulation by using Reed-Solomon Codes and the Tornado Codes.

Despite advantage of ECC, there are still some P2P systems not using the code technology. The result is that it will affect stability of the Internet, degrade performance of the system and implementation complexity. It is presented that similarity and difference between Bit Torrent and Avalanche, the former don't use ECC but the latter use. According to Microsoft, users are able to complete the download by Avalanche even if the last seed goes offline. However, the author of Bit Torrent doesn't think so. He believed that Microsoft research didn't simulate varying transfer rate abilities, transfer rate abilities varying over time, or endgame mode and Avalanche will give rise to secure issue. To verify the work of ECC, We showed a P2P storage system, OceanStore, using ECC. It is obvious that OceanStore is able to provide incremental scalability, secure sharing, and long-term durability by using ECC.

To prove the algorithms, we have implemented the two algorithms, Reed-Solomon codes and Tornado codes respectively by Matlab. The performances of two codes are good according to compare their original data with the restored data.

And compare the two codes, if there are k bits original data, Tornado needs $(1 + \varepsilon)k$ bits to rebuild the data. But Reed-Solomon only needs k bits. There are only one operator (**XOR**) in Tornado, but there are several operators (**XOR**, multiply divide, ...) in Reed-Solomon. This is why Tornado Codes perform faster than Reed-Solomon Codes. It sacrifices a little bit efficiency for speed.

7. References

- [1] <http://en.wikipedia.org/wiki/BitTorrent>
- [2] <http://bramcohen.livejournal.com/1416.html>
- [3] <http://soft.yesky.com/info/170/2020170.shtml>
- [4] <http://www.afterdawn.com/news/archive/6549.cfm>
- [5] <http://www.afterdawn.com/news/archive/6557.cfm>
- [6] <http://bramcohen.livejournal.com/20140.html?thread=224172>
- [7] Christos Gkantsidis and Pablo Rodriguez Rodriguez. Network Coding for Large Scale Content Distribution. IEEE Infocom 2005
- [8] Elizabeth Haubert, Joseph Tucek, Larry Brumbaugh and William Yurcik. Tamper-Resistant Storage Techniques for Multimedia Systems.
- [9] Y. Chen, J. Edler, A. Goldberg, A. Goettlieb, S. Sobti, P. Yianilos "Prototype Implementation of Archival Intermemory", in *Proc. of IEEE ICDE*, February 1996.
- [10] H. Weatherspoon, J. D. Kubiatowicz, "Erasure Coding vs. Replication : A Quantitative Comparison", in *Proc. of IPTPS '02*, March 2002.
- [11] http://www.4i2i.com/reed_solomon_codes.htm
- [12] <http://carlstrom.com/stanford/quals/mirror/swig.stanford.edu/pub/summaries/networks/tornado.html>
- [13] <http://nislalab.bu.edu/nislalab/projects/bchsync/tornado.html>

8 Appendixes

8.1 Codes using Reed-Solomon

```
function [ ]=reed_solomon(n,m)

%initial condition w,h
w=4;
h=1000;

%generate file data
D1=round(rand(n,h*w));
D=zeros(n,h);
for i1=1:n
    for i2=1:h
        D(i1,i2)=D1(i1,(i2-1)*w+1)*2.^3 + D1(i1,(i2-1)*w+2)*2.^2 +
D1(i1,(i2-1)*w+3)*2 + D1(i1,(i2-1)*w+4);
    end
end

%generate the Vandermonde matrix
A=zeros(m,n);
for ii=1:m
    for jj=1:n
        A(ii,jj)=jj.^(ii-1);
    end
end

% get the matrix C
C=A*D;
A=[eye(n);A];

% get the matrix E
E=[D;C];

%fault happend randomly in different devices
error_num=round(rand*(m-2)+2);
error_array=floor(rand(error_num,h)*16);
error_index=zeros(1,error_num);
for ii=1:error_num-2
    error_index(ii)=round(rand*m+n);
end
```

```

end
D2=D;

%plot data with errors
D2(1,:)=floor(rand(1,h)*16);
D2(n-1,:)=floor(rand(1,h)*16);
D2=reshape(D2,1,n*h);
hist(D2,20)
legend('Distribution of the data with errors ')

% compare the original data with error data
figure
[c1,d1]=hist(D2,20);
distance1=d1(4)-d1(3);
d1=round(d1);
stem(d1,c1/length(D2)/distance1);
hold on
temp=D;
[a2,b2]=size(temp);
temp=reshape(temp,1,a2*b2);
[c2,d2]=hist(temp,20);
distance=d2(4)-d2(3);
d2=round(d2);
stem(d2,c2/length(temp),'r-*');
title('the comparison of the original data and error data');
legend('the original data','the error data')
hold off
H=sort([error_index,1,n-1]);

%reconstruct the matrix A1,E1
A1=[];
E1=[];
for j1=1:(m+n)
    sign=0;
    for j2=1:error_num
        if j1 ==H(j2)
            sign=1;
        end
    end
    if sign==0
        A1=[A1;A(j1,:)];
        E1=[E1;E(j1,:)];
    end
end
end

```

```

%reconstruct D
[a,b]=size(A1);
if a~=b
    A1=A1(1:b,:);
    E1=E1(1:b,:);
end
D_restore=inv(A1)*E1;
E_restore=A*D_restore;

%plot D
[a1,b1]=size(D);
D=reshape(D,1,a1*b1);
figure
hist(D,20);

distance1=d1(4)-d1(3);
d1=round(d1);
stem(d1,c1/length(D)/distance1);
legend('Distribution of the original data ')

%plot D_restore
figure
[a2,b2]=size(D_restore);
D_restore=reshape(D_restore,1,a2*b2);
[c2,d2]=hist(D_restore,20);
bar(d2,c2,'y')
distance=d2(4)-d2(3);
d2=round(d2);
stem(d2,c2/length(D_restore)/distance,'m--*');
legend('Distribution of the restored data')

% combine the two curves
figure
[a1,b1]=size(D);
D=reshape(D,1,a1*b1);
[c1,d1]=hist(D,20);
distance1=d1(4)-d1(3);
d1=round(d1);
stem(d1,c1/length(D)/distance1);
hold on
[a2,b2]=size(D_restore);
D_restore=reshape(D_restore,1,a2*b2);
[c2,d2]=hist(D_restore,20);

```

```

distance=d2(4)-d2(3);
d2=round(d2);
stem(d2,c2/length(D_restore)/distance,'m--*');
title('the comparison of the original data and restored data');
legend('the original data','the restored data')
hold off

% get the error rate
[num,error_rate]=symerr(D,D_restore);
num
error_rate

```

8.2 Codes using Tornado

Main program:

```

% tornado coding
%three layers 1600, 800, 400, 400

%generate the original data
data_original=round(rand(1,1600));

% plot original data
[g,h]=hist(data_original,2);
figure
bar(round(h),g,'y')
legend('the original data in tornado coding')
ylim([0,1000])
pause
figure
stem(round(h),g./1600,'r+')
xlim([-0.2,1.2])
layer=[1600,800,400,400];
D=10;

% the degree distribution is poisson distribution in the right side
d=1:D;
Hd=sum(1./d);
al=Hd*(D+1)./D;
beita=1./2;
ar=al./beita;
alf=6.4334;
i=2:D+1;

```

```

pdf=exp(-alf).*alf.^(i-1)./factorial(i-1);
pdf=pdf+(1-sum(pdf))./D; %normalize
figure
stem(i,pdf)
xlim([0,13]);
legend('the distribution of degree from right side')
pause

% get the degrees (truncated heavy tail distribution), for the right side
degree=zeros(length(layer),D+1);
index=[0,1600,2400,2800,3200];
data=zeros(sum(layer),4);
temp=[ones(1600,1),zeros(1600,1),zeros(1600,1),data_original'];
data(1:layer(1),:)=temp;

for i=2:length(layer)
    temp=[ ];
    for i1=2:D+1
        degree(i,i1)=round(pdf(i1-1)*layer(i));

        temp=[forming(degree(i,i1),i1,data(index(i-1)+1:index(i),4)',i);temp];
    end
    size(temp)
    data(index(i)+1:index(i+1),:)=temp;
end
data_c=zeros(sum(layer),5);
for i=1:length(index)-1
    for j=index(i)+1:index(i+1)
        data_c(j,:)=[data(j,:),j-index(i)];
    end
end
plot original data with redundancy
[g1,h1]=hist(data_c(:,4)'.',2);
figure
bar(round(h1),g1)
legend('the distribution of data with redundancy')
figure
stem(round(h1),g1./3200)
xlim([-0.2,1.2])

% reshape the original data to simulate the channel and get the received
data_m=data_c;
epxlong=100;
length_r= length(data_original)+epxlong;

```

```

for i=1:index(5)
    order=ceil(rand(1,2)*(index(5)-1));
    a=order(1);
    b=order(2);
    c=data_m(a,:);
    d=data_m(b,:);
    data_m(b,:)=c;
    data_m(a,:)=d;
end
data_r=data_m(1:length_r,:);

%plot receive data
[g2,h2]=hist(data_r(:,4)',2)
figure
bar(round(h2),g2)
ylim([0,1100])
legend('the distribution of the received data')
pause
figure
stem(round(h2),g2./length(data_r(:,4)'))
xlim([-0.2,1.2])

% decoding part
while length(data_r)<3200
layer1=[];
layer2=[];
layer3=[];
layer4=[];
for ii=1:length(data_r)
    if data_r(ii,1)==1
        layer1=[layer1;data_r(ii,:)];
    elseif data_r(ii,1)==2
        layer2=[layer2;data_r(ii,:)];
    elseif data_r(ii,1)==3
        layer3=[layer3;data_r(ii,:)];
    elseif data_r(ii,1)==4
        layer4=[layer4;data_r(ii,:)];
    end
end
end
restore4=[ ];
for i=1:length(layer4)
    [mat,sign]=check_degree1(layer3,layer4(i,2),layer4(i,3),layer4(i,:));
    if sign==1
        for ii=index(3)+1:index(4)

```

```

        if mat(3)==data_c(ii,5)
            restore4=[restore4;[mat(2),data_c(ii,2),data_c(ii,3),mat(1),mat(3)]];
        end
    end
end
end
restore3=[ ];
for i=1:length(layer3)
    [mat,sign]=check_degree1(layer2,layer3(i,2),layer3(i,3),layer3(i,:));
    if sign==1
        for ii=index(2)+1:index(3)
            if mat(3)==data_c(ii,5)
                restore3=[restore3;[mat(2),data_c(ii,2),data_c(ii,3),mat(1),mat(3)]];
            end
        end
    end
end
end
restore2=[ ];
for i=1:length(layer2)
    [mat,sign]=check_degree1(layer1,layer2(i,2),layer2(i,3),layer2(i,:));
    if sign==1
        for ii=index(1)+1:index(2)
            if mat(3)==data_c(ii,5)
                restore2=[restore2;[mat(2),data_c(ii,2),data_c(ii,3),mat(1),mat(3)]];
            end
        end
    end
end
end
data_r=[data_r;restore2;restore3;restore4;layer1];
end
layer_r1=zeros(layer(1),5);
for k=index(1)+1:index(2)
    for kk=1:length(data_r)
        if data_r(kk,1)==1 && data_r(kk,5)==k;
            layer_r1(k,:) ==data_r(kk,:);
        end
    end
end
end
layer_r2=zeros(layer(2),5);
for k=index(2)+1:index(3)
    for kk=1:length(data_r)
        if data_r(kk,1)==2 && data_r(kk,5)==k;
            layer_r2(k,:) ==data_r(kk,:);
        end
    end
end

```

```

        end
    end
end
layer_r3=zeros(layer(3),5);
for k=index(3)+1:index(4)
    for kk=1:length(data_r)
        if data_r(kk,1)==3 && data_r(kk,5)==k;
            layer_r3(k,:) ==data_r(kk,:);
        end
    end
end
layer_r4=zeros(layer(4),5);
for k=index(4)+1:index(5)
    for kk=1:length(data_r)
        if data_r(kk,1)==4 && data_r(kk,5)==k;
            layer_r4(k,:) ==data_r(kk,:);
        end
    end
end
data_restore=[layer_r1;layer_r2;layer_r3;layer_r4];
plot data_restore
[g3,h3]=hist(data_restore(:,4),2);
bar(round(h3),g3)
figure
stem(round(h3),g3./length(data_restore(:,4)))
symerr(data_c,data_restore)

```

subprogram1:

```

%num data with the same right degree in the layer
function [a]=forming(num,degree,data,layer_index)
%layer_index is for the redundancy not for data
layer=[1600,800,400,400];
i2=0;
ii=1;
no1=floor(length(data)/degree);
for i=1:num
    if layer_index==4 && degree*(ii+1) > length(data)
        i2=1;
        ii=i-no1;
    end
    temp=data((ii-1)*degree+1+i2:degree*ii+i2);

    temp1=temp(1,1);

```

```

    for i1=1:degree-1
        temp1=bitxor(temp1,temp(i1+1));
    end
    a(i,:)= [layer_index,(ii-1)*degree+1+i2,degree*ii+i2,temp1];
    ii=i;
end

```

subprogram2

```

function [x,y]=check_degree1(data,in1,in2,nn)
index=in1:in2;
[a,b]=size(data);
rr=[ ];
ur=[ ];
for i=1:length(index)
    sign=0;
    for ii=1:a
        if index(i)==data(ii,5)
            rr=[rr;data(ii,:)];
            sign=1;
        end
    end
    if sign~=1
        ur=[ur,index(i)];
    end
end
y=0;
x=888;
if length(ur)==1
    rr=[rr;nn];
    x=rr(1,4);
    [q1,q2]=size(rr);
    for iii=2:q1
        x=bitxor(x,rr(iii,4));
    end
    x=[x,data(1,1),ur];
    y=1;
end

```