Master Thesis Report

Implementing Distributed Storage System by Network Coding in Presence of Link Failure

Tanakorn Chareonvisal

Supervisor: Majid Gerami

Date: 17 September 2012

School of Electrical Engineering
Kungliga Tekniska Högskolan
Stockholm, Sweden
Acknowledgements

First of all, I would like to extend my deep gratitude to Dr. Ming Xiao and Majid Gerami for mentoring, guiding and giving valuable technical input through my master thesis. And I would like to extend my special thank you to Awassada Phutathum who is my best younger sister for helping me in technical knowledge. Moreover, I would like to say thank you for my Thai friend in Sweden for support me in everything.

Last but not the least; I would like to express my infinite gratitude to my family and my girlfriend for supporting me throughout all my studies at Sweden.

Tanakorn Chareonvisal
Stockholm, Sep 2012
Abstract

Nowadays increasing multimedia applications e.g., video and voice over IP, social networks and emails poses higher demands for sever storages and bandwidth in the networks. There is a concern that existing resource may not able to support higher demands and reliability. Network coding was introduced to improve distributed storage system. This thesis proposes the way to improve distributed storage system such as increase a chance to recover data in case there is a fail storage node or link fail in a network.

In this thesis, we study the concept of network coding in distributed storage systems. We start our description from easy code which is replication coding then follow with higher complex code such as erasure coding. After that we implement these concepts in our test bed and measure performance by the probability of success in download and repair criteria. Moreover we compare success probability for reconstruction of original data between minimum storage regenerating (MSR) and minimum bandwidth regenerating (MBR) method. We also increase field size to increase probability of success. Finally, link failure was added in the test bed for measure reliability in a network. The results are analyzed and it shows that using maximum distance separable and increasing field size can improve the performance of a network. Moreover it also improves reliability of network in case there is a link failure in the repair process.
Chapter 1

Introduction

1.1 Motivation

Nowadays to support the growth of demand for applications such as video, VoIP and social network, increasing a network capacity is inevitable. So the question we ask is: how to operate these increased traffics? To store a large data over a distributed storage systems (DSS) is a one of the solution which can solve this problem.

The distributed storage system (DSS) is to distribute an entire data to each storage node over a system. The DSS introduces redundancy to improve reliability into the system especially when node fails [1]. There are two schemes of redundancy: repetition code and erasure code. Repetition code is to replicate data in multiple storage nodes and has low complexity. For an erasure code is better storage efficiency than the replication code [2] but it has high complexity for implementation.

Maximum distance separable (MDS) code is one type of erasure code. Given n and k are positive integer which n > k. Initially dividing all data to M fragments and then encode these fragments into n storage nodes (of the same size). Each storage node stores $\frac{M}{k}$ fragments. For instance, it is explained in Figure 1.

![Figure 1 A (4, 2) MDS binary erasure code. Each storage node is storing two fragments that are linear binary combinations of the original data fragments $A_1, A_2, B_1, B_2$. In this example, the total stored data is $M = 4$ fragments. Observe that any $k = 2$ out of the $n = 4$ storage nodes contain enough information to recover all the data (adopted from [1]).](image)

MDS code is optimal storage providing same amount of reliability. This is because k packets stores the minimum amount of data required to reconstruct the entire file. In distributed storage systems, the storage nodes, stored the encoded fragments, spread over
the network. Number of failed storage node can be up to \( (n - k) \) storage nodes within the system.

However the distributed storage system is complicated because there are always some nodes fail or leave the system. So a repair process should be considered. Basically there are two types of repair problem: exact repair and functional repair. Exact repair is to regenerate the same data stored at the failed node. Functional repair is to reproduce the new data which still has a \((n, k)\) MDS property. This property is the receiver can retrieve the original file by connecting to any \(k\) out of \(n\) storage node.

For example Figure 2 shows exact repair when first node fails, the surviving nodes help the newcomer node to regenerate two fragments which is a same data as failed node. Generally if the newcomer node connects to any two surviving storage nodes, it can reconstruct the original file. Then it reproduces the two encoded fragments to store within the new node. Dimakis et, al. in [3] proposed the optimal repair bandwidth by using regenerating code. This regenerating code is to minimize the number of bit sent out from the surviving node when there is node failure. Regarding to Figure 2, the repair bandwidth can be only three fragments from three existing nodes to recover the same data at the failed node.

![Figure 2](image-url)

**Figure 2** Example of exact repair. Assume that the first node in the previous storage system failed. The issue is to repair the failure by creating a new node (the newcomer) that still forms a \((4, 2)\) MDS code. In this example, it is possible to obtain exact repair by communicating three fragments. (Fig 2 in [20])

MDS code has high complexity because it uses a random linear network coding in term of encoding and decoding data [4]. Furthermore we also focus on a link failure rate. Link fail will occur in a repair process by randomly drop one link that connects to the newcomer.
Nevertheless from the previous literatures e.g. [3], [5], [6] and [7], these authors only studied performance of network codes for distributed storage system in term of theory. This is not sufficiently conclusive, as they are based on simulation studies, rather than implementation. So in this report we practically implement the network codes for distributed storage system using MATLAB software.

1.2 Objectives

In this report we implement, compare and investigate the different schemes for distributed storage system (DSS). First we consider probability of success in reconstruction of the original file (P_s) and compare the result between replication and maximum distance separable (MDS) code. Secondly, we study P_s with different field size. Third we determine P_s of different field size in each stage by using minimum storage regenerating (MSR) and minimum bandwidth regenerating (MBR) method. The stage means that number of continuous failure node. Lastly, we investigated value of P_s when there is a link failure involve in a network while repairing process.

1.3 Outline

The thesis is organized as follows: In Chapter 1 some motivations and objective of the thesis are explained. In Chapter 2 we introduce the main idea and concepts involved in network coding and distributed storage systems. Chapter 3 shows the system model, problem description and performance metrics in our investigation. In Chapter 4 we analyze the numerical results with the different strategies of our practical test bed. Finally conclusions and the ideas for future work are discussed in Chapter 5.
Chapter 2

Background

In this chapter we provide a brief discussion of the essential background needed throughout the thesis. Firstly the network coding is illustrated including an information flow graph and max flow min cut theorem. Next the different strategies of network coding for distributed storage system (DSS) are discussed.

2.1 Network Coding

Initially Ahlswede et, al. in [8] proposed the network information flow and network coding. Communication network is represented by network information flow graph. So the question we ask is: how to increase efficiency of communication over a network? Network coding (NC) is an approach to improve efficiency of communication. The NC is an intermediate node between source and receiver can not only store and forward data but also combine independently different incoming data to generate one outgoing data. Then a receiver can decode data to reconstruct the original information sent by source. If combination is linear, it is called linear network coding (LNC) introduced by Li et, al. in [9].

However a complication arises, it is because resource bandwidth is limited. Therefore “max flow min cut theorem” is proposed to define the maximum transmitted amount of data over the network. This maximum transmitted data is assigned on the smallest total capacity of path from source to destination.

In the above paragraph we described about how to apply network codes for communication network. So we also illustrated the potential benefit of network coding. In this part we show two advantages of NC by two examples: increased throughput and security.

2.1.1 Information Flow Graph

A physical communication network is represented by a delay free acyclic directed graph G (V, E), where V is set of nodes in the network: source, intermediate node and receiver and E are set of edges or set of directed links between each node which are noiseless channel.

In this thesis we explained example of information flow graph in Figure 3 ([10]). It shows the communication network with multicast from source (s) to two receivers (y and
Considering the left side of Figure 3, node x does not have additional functionality to combine $b_1$ and $b_2$. So both receivers receive the entire file with different period of time. By the first time, node x sends data $b_2$ to receiver y and then data $b_1$ is transmitted from node x to node z. This shows the weak point of no coding in the network.

![Figure 3 Networks with Multicast from s to y and z][10]

From the right side of Figure 3, there is coding by using a linear combination between input $b_1$ and $b_2$. In this case total time that the two receivers retrieve all data is less than the former case. However for the latter case the receiver must have decoding algorithm to reconstruct the entire file. The encoding and decoding algorithm are explained latter.

**2.1.2 Max Flow Min Cut theorem**

This theorem is the most important within a network because it is to define the maximum transmitted number of bits per time or to assign the minimum total of edges’ capacity. This value is calculated by comparison each set of edges’ capacity to find the minimum capacity. The set of edges should be the same direction from source (S) to destination (D). This theorem is shown in Figure 4.

Regarding to Figure 4, there are 5 set of edges’ capacity which can separate a node to two parts: source (S) and destination (D). From these set of edges’ capacity, we compare the total capacity of each set to get the minimum capacity in the network. For example in Figure 2, set of edges’ capacity is $e_1$ has 4 bits/sec, $e_2$ has 4 bits/sec, $e_3$ has 3 bits/sec, $e_4$ has 5 bits/sec and $e_5$ has 5 bits/sec. The minimum capacity within network is $e_3$. Therefore the maximum transmitted number of bits is 3 bits/sec through the network.
2.1.3 Benefits in Network Coding

Due to applied network coding to communication network, throughput and security are better than communication network without coding [11]. These performances are only a preliminary idea of benefits in network coding for this report.

2.1.3.1 Throughput

Due to increased data traffic, the resource bandwidth should be operated by finding a new method to support increased data. The network coding is a one way to achieve it. This type of network coding is depicted in Figure 5.

Figure 5 The Butterfly Network. Sources S1 and S2 multicast their information to receivers R1 and R2 [11].
From Figure 5 (a) and Figure 5 (b), they show that R1 and R2 only use the all network resource respectively. These can conclude that the two receivers cannot retrieve the all data from the two sources in same time.

Therefore the network coding is applied to all receivers can receive the all information simultaneously shown in Figure 5 (c). By node C combines two input information to generate the encoded output information. This network coding is called “Butterfly Network” [11].

2.1.3.2 Security

The second benefit in network coding is security. Transmitting linear combinations of data is better than uncoded data. This is because uncoded data are easier to observe or copy data which called wiretapping than encoded data depicted in Figure 6.

2.2 Different Strategies of Network Codes for Distributed Storage Systems

Nowadays distributed storage systems (DSS) are very essential due to increased application such as multimedia and e-mail [12]. DSS is to distribute data by redundancy information at any storage node over a network. It improves reliability against node failures in a system. If a node fails, a receiver still gets the original information from any storage node. A new storage node connects to existing storage nodes to recover entire
information or to repair partial information [13]. In additional DSS has less latency for download data. This is because the receiver can retrieve data from any storage node in the network. Moreover DSS can save cost due to stored data at a small disk.

To improve efficiency of DSS, the network code should be applied to DSS. There are two strategies of network code for DSS in our implementation: replication and erasure code. The details for these network codes are explained in below.

2.2.1 Replication Code

Replication code is the simplest means for increasing reliability of storage system [14]. All file of size M bits is divided to k fragments at source node. Then to distribute these fragments to multiple storage nodes which store \( \frac{M}{k} \) bits. The identical copies of data are repeated in storage node spread over a network.

For a repair problem, a newcomer node only connects to a surviving node which stores a same data at the failed node.

Moreover this code has less complexity than other codes. It is because the replication code does not encode the data before distributing it to each storage node. For decoding, it just combines data from any k node to receive the entire file.

2.2.2 Erasure Code

Traditionally communication networks used erasure code because it efficiently stored data while protecting against a node fails. It is much more efficient than replication code. A source file is divided size of M to k fragments. These k fragments are encoded to n coded fragments using an (n, k) maximum distance separable (MDS) code [3], and store them at n nodes [15]. Then the original data can be reconstructed by connecting to any k storage node at the receiver.

For repair problem of a traditional erasure coding or (n, k) MDS code is a newcomer node connects to a surviving node. After reconstruction data, a new data is encoded to store at the new node (d = k). This shows that size of repair bandwidth equals to the original file size. Therefore Dimakis et, al. in [3] proposed a regenerating code. This code can reduce repair bandwidth discussed in a next subsection.

Encoding data for MDS code used random linear network coding (RLNC) [4], [16]. Acedanski et, al, in [6] demonstrated that the random linear network coding is efficient to retrieve the original file of system comparing to un-coded storage and a traditional erasure coding based storage. This is because a probability of available entire file for RLNC closes to one comparing to the remaining of schemes. The procedure for encoding and decoding algorithm is explained in next section.
a) Example of exact repair of a maximum distance separable (MDS) code. When a first node fails, a newcomer node connects to second node and third node. The second node and third node transmit $b_1$, $b_2$ and $a_1 + b_1$, $a_2 + b_2$ to new node respectively. At the newcomer node recovers an original file $(a_1, a_2, b_1, b_2)$ and then encode these data to get a same data $(a_1, a_2)$ stored at the failed node.

b) Example of exact repair of a regenerating code. When a first node fails, a newcomer node connects to three surviving node (node 2, 3 and 4). The second node, third node and fourth node transmit $b_2$, $a_2 + b_2$ and $a_1 + a_2 + b_2$ to new node respectively. At the newcomer node encode these data to get a same data $(a_1, a_2)$ stored at the failed node.

Figure 7 Example of exact repair of both maximum distance separable (MDS) code and regenerating code.

2.2.2.1 Regenerating Code

Regenerating code is a new class of erasure code. This code can reduce repair bandwidth by increasing number of surviving node connected by a new storage node. A difference between maximum distance separable (MDS) code and regenerating code is amount of data in repair problem when a node fails.
For (4, 2) MDS code when there is a failed node, a new node can regenerate encoded data from any two out of four storage nodes (d = k) and recover the original data. Then the new encoded fragments regenerate from the original data. On the other hand, regenerating code, a newcomer code connects to surviving nodes (k ≤ d ≤ n − 1) to rebuild a new encoded data while a node fails. Both different repair problems are shown in Figure 7 (a) and Figure 7 (b).

Regarding to Figure 7(a), the newcomer node just connects to any 2 nodes (node 2 and 3) to rebuild the original file (a1, a2, b1, b2). These two surviving nodes sent b1, b2 and a1 + b1, a2 + b2 to new node respectively. Then the new code encoded this original to get the same data (a1, a2) stored at a failed node.

For repair problem in case of regenerating code Figure 7(b), the new node connects to three surviving nodes (node 2, 3 and 4) to create the new encoded data by transmitting b2, a2 + b2 and a1 + a2 + b2 to new node respectively. The same data (a1, a2) stored at a failed node reproduced at the newcomer node. This procedure shows that the repair bandwidth in term of regenerating code is better than MDS code. It causes total amount of data transferred from surviving node for regenerating code is less than total amount of data for MDS code.

We can conclude that the regenerating code optimally trades between a bandwidth for repair data when any node fails and amount of data stored per node in a network [3], [17]. Next we describe a tradeoff between storage and bandwidth.

### 2.2.2.1.1 Storage – Bandwidth Tradeoff

As a concept of regenerating code, when there is a failed node, a new comer node allows connecting d survive nodes for data repair process. Repair process and regenerating process require bandwidth to transfer a data to the redundancy node which each survives nodes storing α bits. During repair process, each survives d nodes allow to transfer β bits to redundancy node. Therefore, the total repair bandwidth (γ) is equal to γ = dβ . For each set of parameter (n, k, d, α, γ) must be an integer. In this case, if there is one fail node, new comer node can connect to most of the survives which cause k ≤ d ≤ n − 1

**Theorem 1** [3]: For any α ≥ α∗(n, k, d, γ), the point (n, k, d, α, γ) is feasible, and linear network codes suffice to achieve them. It is information theoretically impossible to achieve points with α < α∗(n, k, d, γ). The threshold function α∗(n, k, d, γ) is the following:

\[
α^∗(n, k, d, γ) = \begin{cases} \\
\frac{M}{k}γ \in [f(0), +\infty) \\
\frac{M−g(i)γ}{k−i} γ \in [f(i), f(i − 1)] \\
\end{cases}
\]  

(1)
where
\[ f(i) \triangleq \frac{2Md}{(2k - i - 1)i + 2k(f - k + 1)} \]
\[ g(i) \triangleq \frac{(2d - 2k + i + 1)i}{2f} \]

where \( d \leq n - 1 \), the minimum repair bandwidth can be calculate \((\gamma_{\text{min}})\) from
\[ \gamma_{\text{min}} = f(k - 1) = \frac{2Md}{2kd - k^2 + k} \]  \( (2) \)

From theorem 1, there are two extreme points on the storage-repair bandwidth tradeoff curve which called minimum storage regenerating (MSR) codes and minimum bandwidth regenerating (MBR) codes.

MSR represent the point that storage node has minimum of amount data. From theorem 1, we can derive data storage in each node and repair bandwidth as following:

\[ (\alpha_{\text{MSR}}, \gamma_{\text{MSR}}) = \left( \frac{M}{k}, \frac{M}{k(d - k + 1)} \right) \]  \( (3) \)

In the other hand, MBR represent the point that has a minimum repair bandwidth. By using theorem 1, minimum repair bandwidth can be achieved by following:

\[ (\alpha_{\text{MBR}}, \gamma_{\text{MBR}}) = \left( \frac{2Md}{2kd - k^2 + k}, \frac{Md}{2kd - k^2 + k} \right) \]  \( (4) \)

2.3 Encoding and Decoding Algorithm

Encoding and decoding information is important when implementing network coding. The network coding can achieve a capacity of system. These algorithms are explained in next subsection.

2.3.1 Encoding Algorithm

The linear network coding (LNC) is widely used for encoding data. LNC is to combine all input data to produce one output data. Ho et al. [16] is the first concept of random linear network coding. By network node transmits the linear combination of input data on each outgoing link. It is to describe the relationship between input vector \((\vec{X})\) and output vector \((\vec{Z})\) by using transfer matrix \((M)\) [10]. This is shown in (5).

\[ \vec{Z} = M\vec{X} \]  \( (5) \)

Value of \( \vec{X}, \vec{Z} \) and \( M \) are an element over the finite field size \( F_q \) where \( q \) is power of 2 \((2^n)\) and \( n \) is integer. These elements are chosen independently and randomly.
2.3.2 Decoding Algorithm

After considering input data at the receiver, the decoding algorithm should be investigated. There are two methods to examine whether the input data that it is a linear independence of output matrix whether. Other word is it can recover the entire file or not.

First method is computation of M determinant polynomials does not equal to zero over the finite field $\mathbb{F}_q$. This is shown in (7).

Define

$$M = \begin{bmatrix} m_{11} & m_{12} & m_{13} \\ m_{21} & m_{22} & m_{23} \\ m_{31} & m_{32} & m_{33} \end{bmatrix}$$

(6)

$$\det(M) \neq 0$$

(7)

Second method is to calculate rank of $M$. Matrix $M$ must be full rank to retrieve the entire file illustrated in (8).

$$\text{rank}(M) = 3$$

(8)

These methods show that the decoding complexity algorithm is depending on a finite field $\mathbb{F}_q$ depicted in the next subsection.
Chapter 3

System Model and Method

This chapter describes about a model and method using in this implementation of a distributed storage system. The system model is depicted by using different types of network coding. There are three types of network coding applied in implemented network. For repair problem, we use exactly repair for replication code but using functional repair for the rest. Next we implemented the network considering probability of successful downloading in each network coding. Furthermore we study a probability of successful when network has a link failure while repair process.

3.1 System Model

The system model for implementation is to build a distributed storage system (DSS) by using different types of network coding. The DSS consists of a one source node, four storage nodes and one data collector. Source node will distributed data to four storage nodes and data collector will retrieve data from two selected storage nodes. The whole processes of this implementation are doing on MATLAB programing which using wireless system to transfer information’s.

3.1.1 Network Configuration

In this implementation, communication establishes by using commands in MATLAB to create a link between each node via KTH Wi-Fi network which make source node has a directed link to connect to the four storage nodes to distribute information. For the data collector, it’s also has directed link to connect to any storage nodes to retrieve the original file sent by the sender.

This implementation used six laptops as network test based: one is a source node, four laptops are storage nodes and the last one is a data collector. These nodes are connected to KTH Wi-Fi and all information is transferred via IP address which has assigned to nodes over the network. Moreover, these communications are based on TCP/IP protocol and IEEE 802.11b version [18].

For these storage nodes, there is an additional different functionality depending on each type of network coding. For example, data collector, it just connects to any two out of four storage nodes to retrieve the original sent by the sender. Network typology and implemented system model is shown in Figure 8 and Figure 9.
Figure 8: Network Topology

Figure 9: Implementing System Model for Distributed Storage System with a One Source, Four Storage Nodes and one Data Collector
3.2 Method of Process-to-Process Communication for Distributed Storage System

A basic process-to-process communication consists of one server and many clients. A socket transmission control protocol (TCP) for the basic communication is created to send message for communication each other which the server socket is bounded to one port (e.g. 9090). The port number should be greater than 1,023 because these port numbers are properly client/server process. Other the word, the port number (less 1,024) is used for the life of client is normally short [19]. The socket is created when the server socket accepts the socket from the client. However there are possibly many client sockets to connect to the server socket. Rather there is only one server socket that can connect to the client. We applied this method to a source file distributed to each storage node placed over a network. Furthermore each storage node needs open socket TCP between each storage node to send surviving data when a failed node happens. Moreover both server and storage node have to open socket TCP to a receiver (data collector) to recover the original file send by server. Therefore an end-to-end communication for distributed storage system (DSS) used many socket TCP to send message between each other.

After the socket was opened, the message can transmit to each other within this socket. The success of process-to-process communication for DSS consists of three elements: socket, image file and message using communication.

3.2.1 Socket

A stream socket TCP is used to send the message which a server, storage node and data collector socket are bounded to port number. Port number should be greater than 1,023. There are three important functions used to create socket.

3.2.1.1 tcpip

The tcpip function is to create a TCPIP object that is a client interface for a server socket.

```
ser_soc = tcpip('host IP address', port number, 'NetworkRole', 'client');
```

which details for each variable in this function illustrated in Table 1.

<table>
<thead>
<tr>
<th>Variable of Input Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>host IP address</td>
<td>IP address of a node connecting to</td>
</tr>
<tr>
<td>port number</td>
<td>Assign port number of a node connecting to</td>
</tr>
<tr>
<td>NetworkRole</td>
<td>Specify that a node is server or client within system</td>
</tr>
</tbody>
</table>

After the socket was opened, the message can transmit to each other within this socket. The success of process-to-process communication for DSS consists of three elements: socket, image file and message using communication.
which specify role in the next input parameter.

For example:

```matlab
sv = tcpip('130.229.155.3',9199,'NetworkRole', 'client');
```

From this example is to create TCPIP object that is a client interface for a server socket. A server is assigned IP address and port number to '130.229.155.3' and 9199 respectively. Furthermore we created at a client node so the network role is set 'client'.

Moreover if we would like to set any parameter such as timeout, input buffer's size or output buffer's size, we can set them in tcpip function by using

```matlab
ser_soc = tcpip(…, 'PropertyName', PropertyValue, …);
```

ser_soc is to create a TCPIP object with the specified property name and property value pairs.

For example:

```matlab
ser_soc= tcpip(…,'Timeout',100,'InputBufferSize',1000,'OutputBufferSize',1000);
```

From this example, timeout is 100 seconds to wait to complete a read or write operation. If a time-out occurs, the read or write operation aborts.

InputBufferSize is 1,000 bytes that can be stored in the input buffer during a read operation. A read operation is terminated if the amount of data stored in the input buffer equals the InputBufferSize value.

OutputBufferSize is 1,000 bytes that can be stored in the output buffer during a write operation. An error occurs if the output buffer cannot hold all the data to be written.

Note: The detail of read and write operations are illustrated in the next section.

### 3.2.1.2 fopen

The fopen function is to connect the TCPIP object to the host or remote.

```matlab
fopen(ser_soc);
```

### 3.2.1.3 fclose

The fclose function is to disconnect the TCPIP object from the host or remote.

```matlab
fclose(ser_soc);
```
3.2.2 Image File

After the connection is established, an image is read from graphic file specified by the string filename and format of the file.

\[
\text{image} = \text{imread('filename.fmt');}
\]

which details of input parameter in this function illustrated in Table 2.

<table>
<thead>
<tr>
<th>Variable of Input Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>filename</td>
<td>Name of image file saved in a folder</td>
</tr>
<tr>
<td>fmt</td>
<td>Format-Specific Information (e.g. tif, jpeg, png, gif)</td>
</tr>
</tbody>
</table>

For example:

\[
\text{image} = \text{imread('circles.tif');}
\]

From this example, we would like to send the circles image stored in tif format specific information.

In this thesis we initially send the image file that element of file is 0 or 1 \([\text{GF}(2)]\). However we would like to compare the element of data over different finite field \((\text{F}_q)\). So the question we ask is: how to convert element of data over GF(2) to other GF? There are six functions to achieve conversion from GF(2) to other GF explained below.

- **double**
  Convert logical to double precision.
  \[
  \text{double_image} = \text{double(image);} 
  \]

- **num2str**
  The num2str function converts numbers to their string representations.
  \[
  \text{char_image} = \text{num2str(double_image);} 
  \]

- **reshape**
  The reshape function returns the m-by-n matrix char_image_1 whose elements are taken column-wise from char_image.
  \[
  \text{char_image_1} = \text{reshape(char_image', m, n);} 
  \]
For example:

\[
\text{char_image}_1 = \text{reshape}(\text{char_image}', 1, [])
\]

From this example the transpose of char_image is converted to vector (char_image_1).

- **regexprep**
The regexprep function replaces all occurrences of the regular expression `expr` in string `str` with string `repstr`.

\[
\text{string_image} = \text{regexprep}(\text{str}, \text{expr}, \text{repstr})
\]

For example:

\[
\text{string_image} = \text{regexprep}(\text{char_image}_1, '[^\w''\'], ')
\]

From this example replaces beginning of the numeric and space of char_image_1 to no space between numeric shown in string_image.

- **strcat**
The strcat function is to combine strings in arrays s1, s2, ..., sN.

\[
\text{group_data} = \text{strcat}(s1, s2, ..., sN)
\]

For example:

\[
\text{group_data} = \text{strcat}(\text{string_image} (1), \text{string_image} (2))
\]

From this example, it changed GF(2) to GF(4) by combine string in 2 arrays.

- **bin2dec**
The bin2dec converts binary number string to decimal number.

\[
\text{bin_data} = \text{bin2dec}(\text{group_data})
\]

After we have done six functions, the element of data is different depending on finite field. Its value is between 0 to GF-1.

### 3.2.3 Message

Before sending image file, there should be eight messages sent among server, storage node and data collector. In this case we only consider at one storage node to easily
understand. However the same messages are sent to all storage nodes from one server or one data collector.

3.2.3.1 Finite field

A server sends a finite field message to both storage node and data collector. Sending this message is to specify what the finite field is used in each implementation.

3.2.3.2 Loop size

A server sends a loop size message to both storage node and data collector. This message is sent to indicate how many times that have to run in each implementation. It is very important because network coding for distributed storage system is random. Therefore we should find an average value. The average value makes the result of implement is reliable.

3.2.3.3 Number of repair stage

Both storage node and data collector receives a number of repair stage message from a server. This message specifies how many times for repair problem that we would like to consider.

3.2.3.4 Failed node

A server is random a failed node within system and then only sending the failed node message to a storage node. When the storage node receives this message, the data stored in the failed node deletes.

3.2.3.5 Selected node

A selected node message is sent to storage node by a data collector. It specifies which node that the data collector would like to connect to reconstruct the original file.

3.2.3.6 Ready to receive

A ready to receive message is generated by a data collector sent to a storage node. This message is to show that the data collector is ready to receive data file sent from the storage node. On the other word, this message is to indicate that the storage node can start to transmit the data and coefficient.
3.2.3.7 Acknowledgement for next repair stage

An acknowledgement for next repair stage is sent by a data collector to a server. It is to show that the server restarts the next repair stage. This shows that process-to-process communication is synchronous.

3.2.3.8 Acknowledgement for next loop size

An acknowledgement for next loop size is sent by a data collector to a server. It is to show that the server restarts the next loop size. This shows that process-to-process communication is synchronous.

Procedure for sending these messages is described in Figure 10.

3.2.4 Other functions

After knowing about basic function of process-to-process communication for distributed storage system, in this section we proposed other functions to communicate each node efficiently, reliably and correctly.

3.2.4.1 Write/Read data

To write or read data to socket is a method which uses to transmit or receive data. There are two functions using for writing data: fwrite and fprintf. A fread function uses to read data.

- fwrite

A fwrite function is to write data to object or socket. The input parameters consist of name of socket or object and data sending.

fwrite(Name of Object, Sent Data);

For example:

fwrite(ser_soc, 65:74);

From this example is to write integers from 65 to 74 to server socket (ser_soc).
Figure 10 Sent message of process-to-process communication
• **fprintf**

A fprintf function is different from fwrite function because in this function it needs to specify type of data transmitting.

\[
\text{fprintf} \text{(Name of Socket, 'Type of data', 'Sent Data');}
\]

For example:

\[
\text{fprintf(q_1,'\%c', 'Y');}
\]

In this example, a host node writes Y to q_1 object which %c is to specify that only sends one character to a remote node.

• **fread**

A fread function is to read data from socket. Reading data should be specified size of received data.

\[
\text{fread} \text{(Name of Object, Size of data);}
\]

For example:

\[
\text{fread(ser_soc, 10);}
\]

In this example, a receiver reads 10 sizes of data from server socket (ser_soc).

### 3.2.4.2 Change element of data

This function is to change element in data over a finite field (F_q). On the other word, it creates a Galois field array from the matrix x.

\[
x_{\text{gf}} = \text{gf}(x, m);
\]

The Galois field has \(2^m\) elements where m is an integer between 1 and 16. Default of m is 1. The element of x must be integers between 0 and \(2^{(m-1)}\).

The output x_gf is a variable that MATLAB recognizes as a Galois field array, rather than an array of integers. Therefore before using this array, we have to convert to double array.

\[
x_{\text{non}\_gf} = \text{double}(x_{\text{gf}.x});
\]

.x is field of a carry the numerical value.
For example: if we would like to change element in x to GF(4), m should be 2. The element of output (x_gf) is integers between 0 and 3.

\[ x_{gf} = \text{gf}(x, 2); \]

3.2.4.3 Operation

Due to defined Galois arrays of element in data, there are three operations using in this implementation: addition (+), matrix multiplication (*) and matrix right division (\).

For instance for matrix right division is:

\[ Z = MX \]
\[ X = M\backslash Z \]

However in this thesis, we also implemented the low complexity code. This code does not relate to Galois field array due to low complexity. So an exclusive-OR function is proposed.

\[ C = \text{xor}(A, B); \]

This function performs an exclusive OR operation on the corresponding elements of arrays A and B. Moreover we should change logical array to numerical array before we took them to compute.

\[ D = \text{double}(C); \]

3.2.4.4 Create Transfer Matrix

This function is to create coefficient transfer matrix at each storage node. Element in transfer matrix is created randomly over an assigned finite field size. It is used to encode data. In this thesis, we create this function called “create_r” function.

```
function [output] = create_r(n, k, GF)
    member_r = [0, GF - 1];
    r = \text{gf}([\text{randi}(member_r, \text{row}, \text{col}), \text{log2}(GF)]);
    output = r;
```
where

\[ n \] = number of storage node in a system
\[ k \] = number of storage node connected to recover an original file
\[ GF \] = value of Galois field assigned
\[ row \] = number of row in \( r \) matrix
\[ col \] = number of column in \( r \) matrix

Normally this function is called from main function to create the transfer matrix by return output variable.

### 3.2.4.5 Create data for a repair problem

When a failed node occurs, data stored in this failed node deletes. So the question we ask is: how to transmit data from surviving node to regenerate a new encoded data? We create function to generate data transmission due to a repair problem called “combine_matrix” function.

```matlab
function [combine_result, combine_coeff] = combine_matrix(in_a, in_b, coeff_a, coeff_b, GF)

member_r = [0, (GF-1)];
r = randi(member_r,1,2);
r = gf(r, log2(GF));

in_a = gf(in_a, log2(GF));
in_b = gf(in_b, log2(GF));
out_a = r(1,1)*in_a;
out_b = r(1,2)*in_b;

combine_result = out_a + out_b;
combine_result = double(combine_result.x);

coeff_in_a = gf(coeff_a, log2(GF));
coeff_in_b = gf(coeff_b, log2(GF));
coeff_out_a = r(1,1)*coeff_in_a;
coeff_out_b = r(1,2)*coeff_in_b;

combine_coeff = coeff_out_a + coeff_out_b;
combine_coeff = double(combine_coeff.x);
```

24
This function only implemented at the storage nodes. Firstly we generate two random numbers over a finite field because there are only two fragments stored in each storage node. Therefore number of random value depending on number of stored fragment in a node.

Next we brought two random numbers multiply to each encoded fragments independently. Then combination these two encoded fragments to generate the one fragment transmitting to the failed node. In the same time, we did the same method with transfer matrix created.

From code in this function, we described meaning of each input parameter where

- in_a = first data stored in a node
- in_b = second data stored in a node
- coeff_a = first transfer matrix stored in a node
- coeff_b = second transfer matrix stored in a node
- GF = value of Galois field assigned

### 3.2.4.6 Generate new encoded data at a failed node

This function is to regenerate new encoded data from transmitted data in each surviving storage node.

For implementation this function only implemented at each storage node. Firstly we are random three values to multiply to each sent data because number of surviving node connected are three nodes. So it shows that number of random values depend on number of existing node in the system ($d = n - 1$). Next the new encoded fragment is regenerated by combining three fragments sent from the surviving nodes.

This function is called “combine_matrix_3” function.
function [combine_result,combine_coff] = 
combine_matrix_3(in_a,in_b,in_c,GF,coeff_mat_1,coeff_mat_2,coeff_mat_3)

    member_r = [0,(GF-1)];
    r = randi(member_r,1,3);
    r = gf(r,log2(GF));

    in_a = gf(in_a,log2(GF));
    in_b = gf(in_b,log2(GF));
    in_c = gf(in_c,log2(GF));
    out_a = r(1,1)*in_a;
    out_b = r(1,2)*in_b;
    out_c = r(1,3)*in_c;
    combine_result = out_a + out_b + out_c;
    combine_result = double(combine_result.x);

    coeff_mat_1 = gf(coeff_mat_1,log2(GF));
    coeff_mat_2 = gf(coeff_mat_2,log2(GF));
    coeff_mat_3 = gf(coeff_mat_3,log2(GF));
    in_coeff_1= r(1,1) * coeff_mat_1;
    in_coeff_2= r(1,2) * coeff_mat_2;
    in_coeff_3= r(1,3) * coeff_mat_3;
    combine_coff = in_coeff_1+in_coeff_2+in_coeff_3;
    combine_coff = double(combine_coff.x);

where

    in_a = first data stored in a node
    in_b = second data stored in a node
    in_c = third data stored in a node
    coeff_mat_1 = first transfer matrix stored in a node
    coeff_mat_2 = second transfer matrix stored in a node
    coeff_mat_3 = third transfer matrix stored in a node
    GF = value of Galois field assigned

3.2.4.7 Check (n, k) MDS property

This function is to check property of network coding in a system that this code can
reconstruct an original file or not. It is considered at a receiver or data collector. Checking
a \((n, k)\) MDS property is to check all possible combination for each network coding. Number of all combination is calculated by (1).

\[
\binom{n}{k} = \frac{n!}{(n-k)!k!}
\]  

(1)

Therefore the question we ask is: what is function that we used to check MDS property? A rank function is proposed which it provides an estimate of the number of linear independent rows or columns of a full matrix.

\[
\text{rank}(x);
\]

The rank function computes rank of \(x\) matrix that \(x\) is a square matrix.

In this thesis we implemented by sending all data and all transfer matrix stored in all storage node to the data collector to check all possible combination. It is called “check_full_rank” function.

```matlab
function [output_check] = check_full_rank(data_mat,coeff_mat, GF)
    for
        % take all transfer matrix stored in each storage node
        for
            % take all transfer matrix stored in each storage node but should take them in different node comparing for loop before
            check_matric = combine all transfer matrix from each for loop
            check_rank = rank(check_matric);
            if check_rank == number of fragment
                figure is recovered.
                continue
            else
                return
            end if
        end second for
    end first for
```
where

\[
\begin{align*}
\text{data_mat} &= \text{all encoded data from every storage node} \\
\text{coeff_mat} &= \text{all transfer matrix from every storage node} \\
\text{GF} &= \text{value of Galois field assigned} \\
\text{number of fragment} &= \text{amount of an original file divided}
\end{align*}
\]

**Note:** Number of for loop depends on k value. From the code, k is 2 so there are two loop for in this function.

### 3.3 Problem Description

Regarding to the literature review, they almost proposed the theoretical study about network coding in distributed storage system (DSS). Some literature further used network coding for DSS in term of practical using C++ implementation [19]. These two reasons make us have motivation to build the DSS with the different types of network coding by using functions in MATLAB programming environment.

As already mention in a background, regenerating code give advantages in storage space, bandwidth and reliability more than replication code. So in this report, we implement these coding methods and focus on results based on these points.

### 3.4 Implementation Network coding in Distributed Storage System

Different types of network coding are applied to a distributed storage system (DSS). Process for distribution and repair data in each type is very different. So in this section, we explain how to distribute data from a sender to four storage nodes. Furthermore how to repair data at a newcomer storage node while any storage node fails.

To distribute and repair data, there is a storage bandwidth tradeoff that needs to be concern, which is mentioned previously in the background, is an important factor in the implemented network model. This is because it specifies the amount of data should store in each node. In addition it also used to assign amount of data in term of repair problem. These specified data can make us to get the optimal network coding in DSS.

In our implementation, we starting with replication coding and follow by more complexity as MDS after that link failure will be added in the MDS system.
3.4.1 Implementation Replication code

In this code an entire file is divided to four fragments \( (M = 4) \) then distributed two of systematic fragments to each storage node \( (\alpha = 2) \). There are four storage locations for this implementation \( (n = 4) \). If there is a fail node, the redundancy data will repeat one time per fragment over the network. In the final, to retrieve the original file, the data collector connects to any two out of four nodes \( (k = 2) \).

Considering repair problem the newcomer node connects to the surviving node which stores the same data at the failure node.

The implemented network topology is depicted in Figure 11.

![Figure 11 Implementation Replication Code: \((n = 4, k = 2, \alpha = 2, \gamma = 2)\)](image)

3.4.2 Implementation Regenerating Code

3.4.2.1 Implementation Regenerating Code with MSR

In this code an entire file is divided to four fragments \( (M = 4) \) and then distributed them to four storage locations \( (n = 4) \). The elements in each fragment are over field size \( F_q \) which \( q \) is power of 2. At these storage nodes are added extra functionality. Its additional functionality is to sum the independent input information flows to generate the two encoded fragments at the each storage node \( (\alpha = 2) \) to can retrieve the original data by data collector connects to any two out of four storage nodes.
In term of repair problem, the newcomer node connects to the three existing storage nodes \((d = 3)\) for reconstruct the new encoded data at the new storage node. By these three surviving nodes send only one encoded fragment \((\beta = 1)\) to the new node. Then the new node combines these three fragments to produce two fragments at the newcomer node.

This implementation is shown in Figure 12. In case of both distribution and repair, the regenerating code has MDS property. The MDS property is the original file can be recovered by data collector connects to any two out of four nodes. Moreover, finite field size has been change to observe an effect.

![Diagram of Implementation Regenerating Code: \((n = 4, k = 2, d = 3, \alpha = 2, \gamma = 3)\)]

**2.4.2.2 Implementation Regenerating Code with MBR**

As mention in Theorem 1, all variable must be integer so we have to recalculate all parameter by using equation 4. This implementation is process same as MSR case but in this case an entire file is divided to ten fragments \((M = 10)\) and then distributed them to four storage locations \((n = 4)\). The elements in each fragment are over field size \(F_q\) which \(q\) is power of 2. At these storage nodes are added extra functionality. Its additional functionality is to sum the independent input information flows to generate the six encoded
fragments at each storage node ($\alpha = 6$) to retrieve the original data by data collector connects to any two out of four storage nodes.

In term of repair problem, the newcomer node connects to the three existing storage nodes ($d = 3$) to reconstruct the new encoded data at the new storage node. By these three surviving nodes send two encoded fragment ($\beta = 2$) to the new node. Then the new node combines these six fragments to produce two fragments at the newcomer node. Moreover, finite field size has been change to observe an effect.

### 2.4.2.3 Implementation Regenerating Code with multiple fail stage

From normal case of regenerating (both MSR and MBR) in previous implement, the repair process will end their process only on first stage of repair. But in this implement, after system finish repair process we continue random one node to be a fail node for start second repair process. In the end we ran our test bed until reach 10th stage which we measure probabilities of success recover original data in every stage. Moreover, we have changing finite field size to measure an effect.

![Figure 13: Implementation Regenerating code with multiple fail stage](image-url)
2.4.2.4 Implementation Regenerating code when has link fail in repair process

In this implementation, its use same setting as normal regenerating code as shown in Table 3, but main difference is when there is a fail node, new coming node will connects to the three existing storages node for a repair process. But this case, there will be a link failure involve as shown in Figure 14. Due to a link failure which new coming node doesn't receive a repair data from link fail node so new coming node need to request an extra data (extra $\beta$) from two existing node to complete repair process.

![Figure 14: Implementation Regenerating code with link failure while in repair process (4$\beta$ case)](image)

In normal situation of repair process, new coming node need to receiver at least $3\beta$ to be able to make success in repair. These $3\beta$ came from calculation in Theorem 1 [3]. However in link fail situation, new comer node receiver only $2\beta$ from existing nodes which it need to be greater than $3\beta$. So, this is the reason that new comer node need to request an extra data (extra $\beta$) to be success while data collector retrieve data. Moreover, in this implementation we will study the effect of $\beta$ when new comer receives only one $3\beta$ and $4\beta$ which can refer from Figure 14 and Figure 15.
3.5 Scenarios Parameters

In our implementation, we can categorize our implementation into four scenarios which are replication code, regenerating code with MSR and MBR methods, regenerating code with multiple fail stage and regenerating code with link fail. All of scenarios, we use common setting of parameter as shown in Table 3 but there will have small change in case of implementation for MBR due to parameter $(n, k, d, \alpha, \gamma)$ must be an integer. New setting of parameter must be used in order to measure matrices performances which show in Table 4. All connections are established through Wi-Fi and coding by MATLAB program.
Table 3: Common Parameters

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Assigned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Storage Node (n)</td>
<td>4</td>
</tr>
<tr>
<td>Number of fragments divided from an original file (M)</td>
<td>4</td>
</tr>
<tr>
<td>Number of any storage node to retrieve an original file (k)</td>
<td>2</td>
</tr>
<tr>
<td>Amount of fragment using to store in each storage node (∝ =M/k)</td>
<td>2</td>
</tr>
<tr>
<td>Number of surviving storage node to repair a new data (d = n - 1)</td>
<td>3</td>
</tr>
<tr>
<td>Amount of fragment for repair bandwidth in each storage node (β)</td>
<td>1</td>
</tr>
<tr>
<td>Amount of fragment for repair bandwidth all storage nodes (γ=dβ)</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 4: Parameters in Regenerating code (MBR)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Assigned Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of Storage Node (n)</td>
<td>4</td>
</tr>
<tr>
<td>Number of fragments divided from an original file (M)</td>
<td>10</td>
</tr>
<tr>
<td>Number of any storage node to retrieve an original file (k)</td>
<td>2</td>
</tr>
<tr>
<td>Amount of fragment using to store in each storage node (∝ )</td>
<td>6</td>
</tr>
<tr>
<td>Number of surviving storage node to repair a new data (d = n - 1)</td>
<td>3</td>
</tr>
<tr>
<td>Amount of fragment for repair bandwidth in each storage node (β)</td>
<td>2</td>
</tr>
<tr>
<td>Amount of fragment for repair bandwidth all storage nodes (γ=dβ)</td>
<td>6</td>
</tr>
</tbody>
</table>
Chapter 4

Numerical Results

4.1 Performance Metrics

There are many performance metrics to measure performance in each coding type. In this session we use probability to measure coding performance. Detail of performance metrics that we used to evaluate will be explained in this session.

4.1.1 Success Probability

We use probability to indicate the performance of network coding. A success probability means the probability that receiver can recover an original data. Each network coding has its formula to calculate its probability which we will explain in this session. Moreover we will explain how we measure probability in each implementation.

4.1.1.1 Replication Code

Replication code, we divide data in to four fragments and distributed two fragments to each storage node as shown in Figure 11. Once data collector needs to retrieve a data, it needs to connect any two nodes out of four nodes to download two fragments from each selected nodes. Somehow, four fragments that data collector have retrieve may not able to recreate original data.

In theory, we can calculate the probability that data collector can successful recreate original data as $P_{succ} = \frac{\binom{2}{1}\times\binom{2}{1}}{\binom{4}{2}} \approx 0.67$. This implies 67%, the original data can reconstruct back as original data.

From implementation, we measure the number that data collector is successful recover a data divide by number of iteration, in this case we ran out implement as 200 rounds. The result of implementation will show in next session.

4.1.1.2 Regenerating Code

Path of fragments were distribute to storage node over network, after that data collector randomly select two nodes to retrieve a data. To measure a performance of network coding, we can calculate probability by using equation 9. This probability use to indicate the chances that retrieve data are success for recovers a data.

\[
P_r(D_r = d) = \frac{\prod_{i=0}^{d-1}(q^{kr} - q^i) \prod_{i=0}^{d-1}(q^m - q^i)}{q^{mkr} \prod_{i=0}^{d-1}(q^d - q^i)}
\] (9)
where:  
m is number of storage node
r is amount of fragment in each storage node
k is number of node can rebuild the original at data collector
d is amount of fragment in all storage node

In implementation, we measure probability by using number that data collector is successful recover a data divide by number of iteration, in this case we ran out implement as 200 rounds. The compare result between theory and implementation will show in next session.

4.2 Implementation Result

In this path, we will compare result in each implementation that we have done with our test bed and compare performance in each case.

4.2.1 Result of Implementation Replication Code

In implementation, as we already explain earlier, the success probability will measure when data collector was retrieve data and success recovers original data which means all fragment of data must be completely received. From theory, successful probability of replication of n=4 and k=2 with 4 fragment of data is equal to 67%. And from our implementation success probability is equal to 66%. From Figure 16, result shows that implementation correlate with theory result.

![Compare Success Probability between Repetition (4,2) and Regenerating (4,2)](image)

**Figure 16: Compare Success Probability between Replication (4,2) and Regenerating (4,2) in term of implementation and Theory**
Moreover, we compare probability between replication code and regenerating code on (4,2) by using same field size (q=2) in both network coding. From result in Figure 16, show that replication coding has better performance for recover original data back. Effect of field size will show in the next session.

### 4.2.2 Result of Implementation Regenerating code

In this path, we have done our implementation on our test bed. Original data is dividing in to four fragments then encode and distributed onward to four storage nodes. Therefore, elements in each fragment are based on field size $F_q$. On our test bed, success probability count from a successful recovers original data from two random selected storage nodes. In the end, we vary a field size to see the effect of it’s with success probability.

![Comparison Success Probability of Regenerating Code (4,2) between implementation and theory](image)

**Figure 17: Comparison Success Probability of Regenerating Code (4,2) between implementation and theory**

From a result in Figure 17, we use an equation 9 to compare an effect of field size with success probability. As the calculation result from equation 9, it shows that when increasing filed side in term of power of 2. A probability of success rate is increasing and approaches one as a field size increase. And from implementation, it also show a same trend as theory shows. Its imply implementation result correlate with theory result.

Moreover, if we compare result between replication and regenerating code as show in Figure 16, it shows that replication has better result in field size equal to 2. But in the end result, increase field size starting from 4, regenerating code has better performance.
4.2.2.1 Result of Implementation Regenerating code with MSR and MBR

In this implementation, we use regenerating code as a main method to distribute our encoded data. But we will mainly focus between storage and bandwidth tradeoff as we already mention in chapter 2.

We divide our original data and use setting as we shown in Table 3 for MSR and Table 4 for MBR. Then distribute data over the network to storage node. After that, data collector selected two nodes to retrieve encoded data and reconstruct in to an original file. A success probability in this case will count when data collector can reconstruct original file from all possible case from four nodes which means reconstruct data must be success from all six cases in all four nodes.

From Table 5, “Without Repair” means an initial stage that no node fails or repair process involves. As the result, it shows that minimum storage regenerating has lower performance for recover original data in the lower field size. But while increasing field size MBR shows it has better performance to recover data starting from GF (16), probability of recover data is reach one which means all six cases can be recover data.

Table 5: Compare Success Probability between MSR and MBR in without repair

<table>
<thead>
<tr>
<th>Tradeoff</th>
<th>Field Size (q)</th>
<th>GF 2</th>
<th>GF 4</th>
<th>GF 8</th>
<th>GF 16</th>
<th>GF 32</th>
<th>GF 64</th>
<th>GF 128</th>
<th>GF 256</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSR</td>
<td></td>
<td>0.01</td>
<td>0.145</td>
<td>0.405</td>
<td>0.635</td>
<td>0.825</td>
<td>0.91</td>
<td>0.945</td>
<td>0.975</td>
</tr>
<tr>
<td>MBR</td>
<td></td>
<td>0.205</td>
<td>0.88</td>
<td>0.985</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

4.2.2.2 Result of Implementation Regenerating code with repair process and multiple fail stage

In this implementation, to begin first repair stage we random select one node as a fail node. Survives nodes will send a repair data to repair this fail node. After that we continue select one fail node to start next repair stage process. In the final, we ran our implementation in to 10th stage in each field size and each coding method (MSR and MBR).

From result in Figure 18 and Figure 19, higher in field size give a result with higher probability. But success probability is decreasing due to it need to concern from a first stage of repair to that stage. So, the probability is a multiply by first stage with second stage and so on to that stage which give a decreasing result in success probability.
Figure 18: Compare Success Probability of different field size in each stage [MSR]

Figure 19: Compare Success Probability of different field size in each stage [MBR]
By compare between initial stage and other state, from result we can calculate percentage which drops from initial stage (without repair case). As the result in Figure 20 and Figure 21, it shows that MBR give lower percentage drop than MSR method. Moreover, by changing field size help to improve percentage drops as it shown in result.
In term of tradeoff between bandwidth and space for storage data as already mention in Theorem 1. From Figure 22, it shows that MSR spend 0.5 storage data in a storage node but it need to more bandwidth as 0.75 for repair failure node. In the other hand, MBR store more data as 0.6 in storage nodes but spend only 0.6 for repairing bandwidth.

![The Optimimal Tradeoff (4,2) between MSR and MBR](image)

**Figure 22: The optimal tradeoff curves between storage and repair bandwidth**

### 4.2.2.3 Result of Implementation Regenerating code when has link fail in repair process

In this implement, we setup our test bed by using setting in Table 3 and network typology as shown in Figure 14 and Figure 15. But in a repair process, we assume that there is a link failure involve in a network. This link fail is depend on link fail probability value which will indicate a chance that one random link that connected from survive node to new coming node will be fail. By using Theorem 1 and setting in Table 3 the minimum number data that new coming node need to receive during repair process in order to have a chance to success in repair is equal to $3\beta$. Since there is a link fails in the system which means number of survive node will be less ($d_{n-2} = 2$) than without link fail case ($d_{n-1} = 3$), new coming node need an extra data to perform successful repair process. These extra data must greater or equal to data that use in without link fail case or it implies that $\beta_{n-2} \geq \beta_{n-1}$. So, in this implement we ran our experiment by varying a repair data that will send out from surviving node to measure a performance of regenerating code when it has link fail occur.

By comparing between Figure 23 and Figure 24, it show that when surviving node send more data ($4\beta$) to new coming node for repair process, it increase a probability of
success and it also improve their reliability and maintain their MDS property as probability of link fail increasing, probability of success still maintain their level no matter link fail probability change. In the other hand, surviving nodes send less data in this case ($3\beta$), the success probability is decreasing every step of link fail probability increase.

![Figure 23: Probability of Success rate when $d = n-2, \gamma=3$ (Link fail)](chart1)

![Figure 24: Probability of Success rate when $d = n-2, \gamma=4$ (Link fail)](chart2)
Now, consider percentage drop from initial stage (without repair case). It shows that when surviving nodes send out a repair data equal to \((2\beta + \beta_{extra})\), probability of success rate is decreasing every step of link fail probability. But once increase send out repair data to \((2\beta + 2\beta_{extra})\), as shown in Figure 26 success rate is improve and approximate constant in each change of link fail probability. Moreover, increase field size help to improve success rate.

Figure 25: Compare percentage drop from without repair case \(d = n-2, \gamma=3\)

Figure 26: Compare percentage drop from w/o repair case \(d = n-2, \gamma=4\)
Chapter 5

Conclusion and Future Works

In this thesis, we have studied different schemes of distributed storage system such as replication code, maximum distance separable and regenerating code which has been implemented in our test bed using MATLAB programming. Second, we have measured a probability of success between MSR and MBR in repair process. Moreover, we have investigated probability of success by adding multiple fail stage to both MSR and MBR case. Lastly, we have investigated a successful rate for recover original data while network has a link failure.

As a result, regenerating give a higher success rates for recover an original data. Increasing field sizes also improve a chance to recover an original data. But there are some advantages and disadvantages points which need to be considered. Since regenerating and MDS has a high complexity, it not easy to implement these methods to a real system. Moreover, regenerating code provides better storage efficiency than replication. In the other hand, replication code is a simple network coding which easier to implement.

In repair process, regenerating provide high performance for recover an original data but its need to select a proper parameter between storage space per node and bandwidth for repairing process. There are two extremely points that are MSR and MBR which shown in this report. Since MSR give lower in storage space but spend more bandwidth during repair process. Vice versa, MBR spend more space to keep their data in storage node but use lower bandwidth for repairing failure node. For overall, it needs to select a proper parameter for bring out the best performance for network coding. Moreover, by sending additional data from survive nodes to new coming node during repair process while it has a link failure involve in the network also improve a success rate for recover an original data.

For a future work, we can implement our test bed to a large scale network such as mobile cellular. Since, when a new coming user enter same area and wish to download some data which previous user already downloaded or user in that area contain some partial data. By using regenerating coding, new user can retrieve some partial data and encode that data to an original data. This will give a lower outgoing bandwidth and decrease response time for gartering data from that cell site to a server.
References


