Chapter 3

Fault-Tolerant Distributed Node-to-Node Routing

In this chapter, we present an adaptive routing algorithm that finds n disjoint shortest paths from the source node s to target node d in an n-dimensional hypercube. The proposed algorithm exploits the path redundancy available in hypercube network by means of a multipath routing approach. This algorithm is able to tolerate node/link failures as well as capable to deal with congestion problems. Fault-tolerant routing over all shortest node-disjoint paths has been investigated to overcome the failures encountered during routing in hypercube networks.

In this chapter, we present an efficient methodology based on the use of disjoint paths to provide fault-tolerant routing which has been investigated on hypercube networks. The proposed approach is based on all shortest node-disjoint paths concept in order to find a fault free shortest path among several paths provided. The proposed algorithm is a simple uniform distributed algorithm that can tolerate a large number of process failures, while delivering all n messages over optimal-length disjoint paths. However, no distributed algorithm uses acknowledgement messages (*acks*) for fault tolerance. So, for dealing the faults, acknowledgement messages (*acks*) are included in the proposed algorithm for routing messages over node-disjoint paths in the hypercube network. Simulation results confirm that the proposed node-to-node routing algorithm provides an average of 10% improvement in the performance of hypercube network in comparison with the previously proposed routing algorithms–depth first search algorithm and unsafety vectors algorithm.

The remainder of this chapter is organized as follows. The overview about properties of hypercube is explained in section 3.1. The phases of fault-tolerant distributed routing are discussed in section 3.2. Then, preliminaries for routing algorithms are discussed in section 3.3. Section 3.4 describes the two types of algorithms for hypercube-based routing: one in the case of non-faulty and another one in the case of failures. Section 3.5 gives the proof of correctness. Section 3.6 provide the evaluation model of the contribution. Section 3.7 presents simulation platform and results with the help of examples. Finally, the method is summarized and some concluding remarks of this chapter is given in section 3.8.

3.1 Overview

Actually, the proposed fault-tolerant routing approach is based on the state information of source-destination paths. If there are no nodes/links failures along a particular path, then when message reaches to the destination node, the destination node sends the *acks* message to the intermediate node and intermediate node sends to the source node for confirming the delivery of message. If there is at least single node/link failure along a particular path, then intermediate node/destination node will not send the *acks* message to the source node. By this way to alert the source node about the failure in the path, the later action is to discover alternative paths by flipping the bit position corresponding to source-destination path. At the same instant, the message which has been sent along the path where the node/link failure has been occurred are rerouted for the destination node. This rerouting action is meant to be a quick response to node/link failures, it may not be the optimal solution. At this point, the proposed method disables the faulty path and improves the performance by searching an optimal path between source node and destination node. Altering the bit position every time, provides the number of alternative paths, which can be used to distribute the traffic burden on the network. Using alternative paths, the method is able to avoid node/link failures and by distributing load among these paths improves the system performance.

Let H_n be the n-dimensional hypercube. From the connectivity of H_n and Menger's theorem, we know that H_n can tolerate at most $n-1$ arbitrary faulty nodes for point-to-point communication or unicast (it is a communication between a single sender and a single receiver over any network). So, for optimal solution, much effort has been

committed to deriving the sufficient conditions that allow $n-1$ faulty nodes for unicast in H_n.

The diameter and bisection width of an n-cube network both are n [107]. The diameter of a network is the largest distance between any two nodes. So, it can be used to measure the maximum communication delay. Bisection width is the minimum number of links that must be removed in order to divide the network into two halves. So, it can be used to measure the capabilities for efficient applications.

Routing is a process of transmitting messages (say m) among processors, and its efficiency is crucial to the performance of a multiprocessor system. Efficient and reliable routing can be achieved by using internally node-disjoint paths, because they can be used to avoid congestion, accelerate transmission rate, and provide alternative transmission routes. Two paths are internally node-disjoint if they do not share any common node except their end nodes. The concept of disjoint paths arose naturally from the study of routing, reliability, and fault tolerance in parallel and distributed systems [62, 63, 94].

Fault-tolerance is the ability to perform its function correctly even in the presence of internal faults. The aim of the fault-tolerance is to increase dependability of the system. The reliability of data processing and data communication is very important in hypercube systems as in all parallel systems. The speedup and the fault-tolerance may be decreased if one or more nodes become faulty. In order to determine and avoid the faulty nodes in the data communication, there are many different kinds of methods to find the disjoint shortest paths [99] between the source and the target nodes.

There are many algorithms for node-disjoint paths routing in n-dimensional hypercubes over node-to-node [112], node-to-set [16], and set-to-set [53, 55]. However, no distributed algorithm uses acknowledgement messages (*acks*) for fault-tolerance. So, for dealing the faults, acknowledgement messages (*acks*) are included in the proposed algorithm for routing messages over node-disjoint paths in a hypercube network. In this chapter, we focus on the problem of constructing n disjoint paths from source node to the destination node in an n-cube network so that their total length and/or maximal length are minimized.

3.2 Fault-Tolerant Distributed Routing

3.2.1 Detection & Notification

Under normal conditions, when there are no failures, the proposed fault-tolerant routing algorithm routes the message from the source to the destination through path routing functions. When traversing routing paths, if each node receives *acks* message, then it means there is no failure in the system. Noticing a node/link failure to means *acks* message is not received by the source node. The use of this *acks* message is to alert source node that there is no failure along communication paths. Figure 3.1 graphically shows the normal message passing and Figure 3.2 shows detection and notification of node/link failures.

From Figure 3.1 to Figure 3.3, the dark blue colour vertices denote source/destination nodes and dark gray colour vertices denote the intermediates nodes for a particular path.

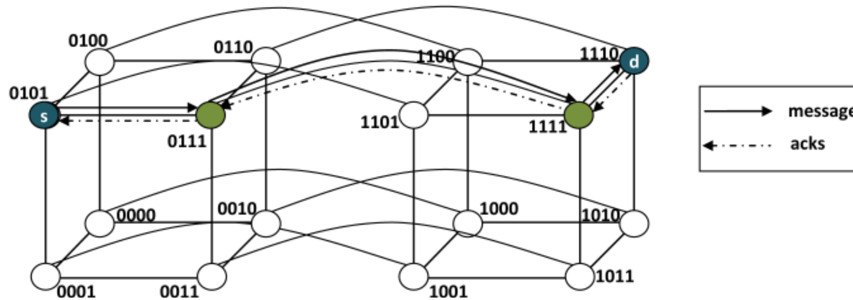

Figure 3.1: Message passing in normal condition.

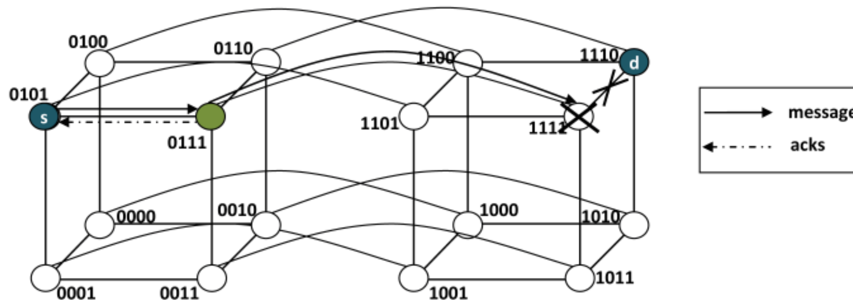

Figure 3.2: Notification of node/link failures.

3.2.2 Selection of Alternative Paths

The process of selecting an alternative paths is started after detection of node/link failures. This step provides the solutions for re-routing message. The node that has detected the node/link failure, since it has not received *acks* message from the neighbourhood node. After getting this information, the source node selects a set of non-faulty intermediate nodes for alternative path to the destination. Basically, alternative paths consist multi-step path of three or more path segments in three stages. The first stage contains the set of segments between the source node and the first intermediate node; the second stage corresponds to the segment between the first and the last intermediate nodes; at last, the third stage comprises the set of segments between last intermediate node and the final destination node. Figure 3.3 shows the selection of alternative path in the presence of node/link failures.

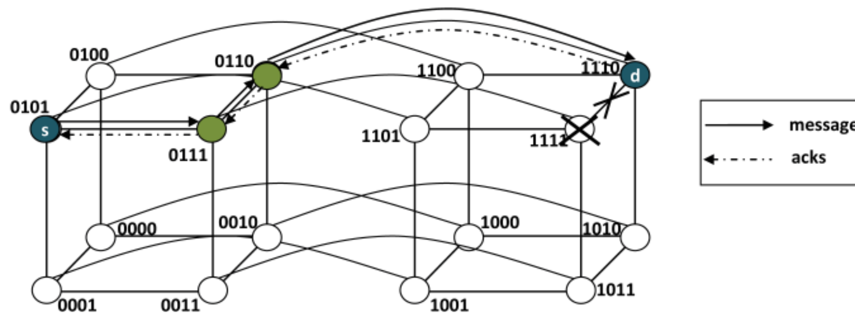

Figure 3.3: Selection of alternative path.

The set of intermediate network nodes works as a super-node which is used in alternative paths. Intermediate nodes are chosen according to their Dimension-Order Routing in order to maximize the number of possible paths to avoid node/link failures. Multi-step paths are segmented paths, which relies on the use of dynamic routing provided by hypercube topology. Our algorithm provides alternative paths to avoid node/link failures and configuring source-based alternative paths.

Dimension-Order Routing (DOR) is the most common deterministic routing scheme used in the direct networks. In DOR, a message traverses the network in proper sequence over an ordered set of dimensions (e.g., $dim_0, dim_1, dim_2, ...$) of path. For example, suppose source node $s = 0101$ and destination node $d = 1110$. According to the dimension-order routing, we have to alter the bit every time in new dimension and

in sequence. For this, first we calculate the hamming distance between s and d. So, the hamming distance is $0101 \oplus 1110 = 1011$. It means that we can select three shortest alternatives paths from s to d. These are as follows:

First path $(0101, 1110) = (0, 1, 3) = 0101 \rightarrow 0100 \rightarrow 0110 \rightarrow 1110$

Second path $(0101, 1110) = (1, 3, 0) = 0101 \rightarrow 0111 \rightarrow 1111 \rightarrow 1110$

Third path $(0101, 1110) = (3, 0, 1) = 0101 \rightarrow 1101 \rightarrow 1100 \rightarrow 1110$

In the first phase source node finds all active intermediate nodes by altering its bits one by one, in second phase intermediate nodes finds others neighbours intermediate nodes by the same process, and in last phase intermediates nodes find the destination node. These intermediate nodes are chosen by bit flipping.

But we have considered that the node 1111 is faulty, so we can't choose second path. We can choose first or third path from source to destination.

3.2.3 Configuration of Alternative Paths

In this phase, proper configuration is required on source-based routing paths between the source and destination affected by node/link failures. This is important since multiple node-disjoint paths were designed to provide alternative paths in the presence of node/link failures.

The selection of suitable paths is the main objective in the presence of components failures for source-destination pair. When a source node does not receive the *acks*, then the source node configures multiple alternative routing paths to the destination node. The configuration of these source-based alternative paths is similar to the one explained in the section 3.2.2. In the source-based routing, intermediate nodes do as scattering and gathering domains for the alternative non-faulty paths.

When the alternative paths have been configured, then the source node is able to monitor the congestion of each path. If the message reaches to the destination node through a non-faulty path, it mean source node received *acks* message successfully. The set of alternative paths between any source-destination pair is shown in Figure 3.4. The unique paths are given below–

$0100 \rightarrow 0101 \rightarrow 0111 \rightarrow 0011 \rightarrow 1011$

$0100 \rightarrow 0110 \rightarrow 0010 \rightarrow 1010 \rightarrow 1011$

0100 → 0000 → 1000 → 1001 → 1011

0100 → 1100 → 1101 → 1111 → 1011

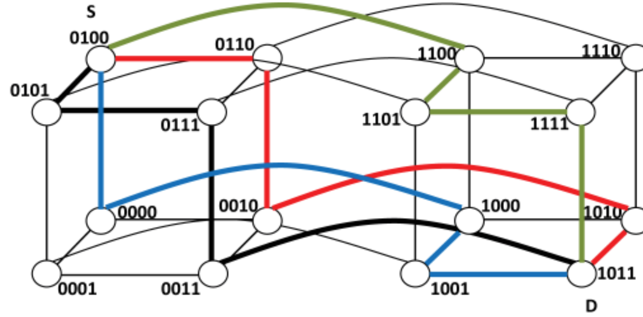

Figure 3.4: A multipath composed by 4 multistep paths in hypercube.

In Figure 3.4, four different colours of edges represent the unique path from source to destination. In H_4, there are four different possible paths since the degree is four.

The objective of the multipath configuration process is to determine the suitable number of alternative paths for any source-destination pair. Thus, the multipath configuration process is applied on the basis of topological properties of the network. The intention of this process is to provide alternative non-faulty paths in a network. In the worst case, the network congestion problems may vary after the occurrence of node/link failures.

The multipath selection procedure is invoked before any new message is came into the network. This procedure has two main goals:

- Avoid the use of faulty paths, and
- Distribute the communication load among the multipath.

3.2.4 Transient and Permanent Faults

Our proposed algorithm is able to deal with a large number of dynamic node/link failures. These methodologies are appropriate for handling permanent failures because they provide alternative communication paths based on network conditions (by avoiding faulty paths). Nevertheless, these alternative communication paths are not solely optimal, when dealing with transient node/link failures. For example, suppose there is a node/link failure in a source-destination pair and after configuration they use maximal

or congested paths for providing fault tolerance routing. This situation is acceptable because of the prevention of best suitable path caused by node/link failure in the system. Whenever the minimal paths are available in the network, the routing method immediately switch back to minimal paths. Thus, the overall system performance can be improved.

For maximum use of network resources, our methodology detects and applies different types of solutions to permanent and transient node/link failures. At first time, node/link failures are always considered and handled as transient failures. If failures continue to exist after a fix time interval, then its state changes to permanent. The detection and right solution of transient and permanent node/link failures is based on properly dealing of information about failures. Node/link failures notifications (when *acks* message not received) warn the source node about the existence of one or more node/link failures along the communication path and need to configure for choosing new source-destination path. In this way, the reception of *acks* message implies the absence of failures along the communication path.

As explained in the previous section, when source node does not receive *acks* message from its neighbour node along the source-destination path, source node considers that node/link failures as transient at first time. Source node once again resend the message to that node, if the source node does not receive *acks* message regarding the same node/link– that node/link failures treated as permanent. Otherwise stated, a node/link failure will be noted as permanent only after not receiving *acks* message notifications regarding that particular node/link. When source node does not receive *acks* message, the source node resends the message up to a threshold value. In our methodology, we set the threshold value by default to 15.

The reception of an *acks* message means that there is no node/link failure in the path of source-destination and message has reached to the destination node through the path. So, all the node/link state along this path is fault-free. On the other hand, upon not receiving the *acks* message, source node resends the message up to a specific threshold value (15 times) and after reaching this specific threshold value, the state of node/link failure is changed from transient to permanent. For clarity, the entire process related to reception of *acks* message has been shown in Figure 3.5 with the help of the flow diagram.

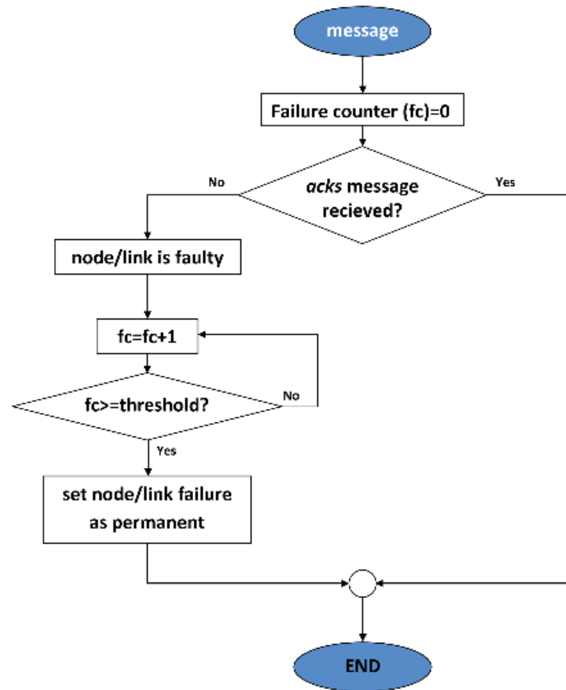

Figure 3.5: Reception of *acks* messages.

Source node is aware of the states of its source-destination paths. If there are one or more permanent node/link failures along the source-destination paths, then source node attempts to send the message via one or more alternative communication path and sends remaining messages through them. The entire process are detailed in Figure 3.6 with the help of the flow diagram.

The most important design parameters for solving the transient node/link failures are: the method to identify the failed node/link; and the method for setting a faulty node/link as permanent. Our algorithm uses:

♣ An *acks* based method where node/link failures are set as permanent only after not receiving *acks* message (source node waiting for *acks* message after resending the message 15 times).

♣ Source based probes for identifying the state of failed node/link. Probes are sent for every 100 messages, 99 are sent via alternative paths and 1 via the faulty path.

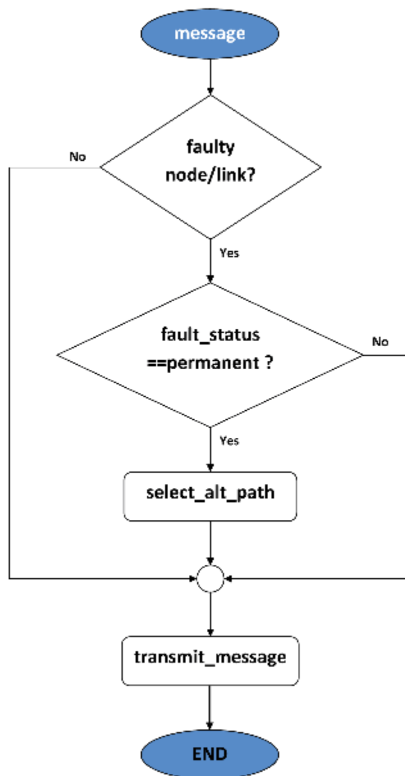

Figure 3.6: Message passing flow diagram.

3.3 Preliminaries

The node-disjoint paths problem is a problem of finding node-disjoint paths from a source node s to destination node d, where $d \neq s$. If $d = s$, then it means that the source node s and destination node d are the same. In this case, although message m can not be propagated, it will remain held in process s. Any two or more paths from s to d are said to be node-disjoint iff they do not have any common nodes except for their end-nodes. Disseminated solutions to the node-disjoint paths problem have many applications such as secure message passing, reliable routing, network survivability and extending available bandwidth.

An n-dimensional hypercube H_n consists of 2^n nodes, and $n.2^{n-1}$ links since the degree of each node is n [107]. For each node i of H_n, node id i is a unique n-dimensional bit label $a_{n-1}a_{n-2}\dots a_0$ where larger number indices denote most significant bit positions. Every node A has label or address $a_{n-1}a_{n-2}\dots a_0$ with $a_i \in \{0, 1\}$, where a_i is

called the ith bit or i^{th} dimension of the address. Sometimes, i is called i^{th} coordinate. In this case, a_i is called the value of the coordinate i. The important property of the hypercube is that, it can be constructed recursively from lower dimensional cubes (subcubes). Each subcube S_m (or m-subcube) has a unique address $s_{n-1}s_{n-2}\dots s_0$ with $s_i \in \{0, 1, *\}$, where exactly $m(0 \leq m \leq n)$ bits take the value $*$ ($*$ is a don't care symbol). A complete hypercube itself can be considered as a special subcube, where all the bits of its address take the $*$ value. Each node is also a special subcube in which no bit of its address takes the $*$ value, and is called 0-cube. Each link has one bit that takes the value $*$ and is called 1-cube and a quadrangle is called 2-cube and has two bits that take the value $*$. A cube is 3-cube, if its address has three $*$, etc.

The n-dimensional hypercube can be partitioned into two disjoint $(n-1)$- dimensional subcubes. Hypercube have a recursive structure, so this property is a key in developing the algorithms in this chapter. For $n \geq 1$, H^0_{n-1} and H^1_{n-1} be the $(n-1)$-dimensional subcubes of an n-dimensional cube H_n, i.e., the most significant bit of the node address of every node is 0 in H^0_{n-1} and 1 in H^1_{n-1}. H^0_{n-1} and H^1_{n-1} are both isomorphic to H_{n-1} and are connected to each other by edges in dimension i of H_n. H_n can be partitioned into H^0_{n-1} and H^1_{n-1} on any dimension i $(0 \leq i \leq n-1)$. Similarly, H_n can be further partitioned into four $(n-2)$-dimensional subcubes $H^{00}_{n-2}, H^{01}_{n-2}, H^{10}_{n-2}$, and H^{11}_{n-2} on dimensions i and j, i.e., H^{00}_{n-2} and H^{01}_{n-2} are the subcubes of H^0_{n-1}; and H^{10}_{n-2} and H^{11}_{n-2} are the subcubes of H^1_{n-1}.

In a hypercube there may be faulty nodes and/or faulty links. In this case the hypercube is called faulty hypercube. If non-faulty nodes and links exist in n-cube, it is called a complete cube. A cube is called a subcube of complete n-cube if at least one dependent (0 or 1) coordinate exists in the cube with dimension n.

3.3.1 Basic Hypercube Routing

Given two nodes u and v in H_n, the Hamming distance path is the minimal (shortest) path obtained by flipping the non matching bits in ascending order of dimensions. For example in H_4, the first Hamming distance $(0001, 1010) = (0, 1, 3) = 0001 \rightarrow 0000 \rightarrow 0010 \rightarrow 1010$, the second Hamming distance $(0001, 1010) = (1, 3, 0) = 0001 \rightarrow 0011 \rightarrow 1011 \rightarrow 1010$, and the third Hamming distance $(0001, 1010) = (3, 0, 1) = 0001 \rightarrow 1001 \rightarrow 1000 \rightarrow 1010$. As this routing strategy only provides $n-1$ disjoint paths and

we are looking for n disjoint paths from source node to destination node. So, we have to apply Coordinate Sequence (CS) in every dimensions.

Figure 2.10 depicts the structure of the dimensions. If we move in x direction, then we will change 0th bit position and it is called 0-dimension. If we move in y direction, then we will change 1st bit position and it is called 1-dimension. If we move in z direction, then we will change 2nd bit position and it is called 2-dimension, and so on.

We assume that the source and destination differ in n dimensions $\{0, 1, 2, \ldots, n-1\}$, denoted as a set, CS. We can define the coordinate sequence CS : $\langle 0, 1, 2, \ldots, n-1\rangle$ as the basic routing path. CS can be any permutation. CS determines how the multiple node-disjoint paths are constructed based on the resolution order of dimension differences. CS^i is defined as i circular left shifts of CS. $CS^0, CS^1, \ldots, CS^{n-1}$ will create n node-disjoint shortest paths:

- 1st path — given by $CS^0 : \langle 0, 1, 2, \ldots, n-1\rangle$
- 2nd path — given by $CS^1 : \langle 1, 2, \ldots, n-1, 0\rangle$
- 3rd path — given by $CS^2 : \langle 2, \ldots, n-1, 0, 1\rangle$

  $\ldots$
- n^{th} path — given by $CS^{n-1} : \langle n-1, 1, 2, \ldots, n\rangle$

Here, the paths generated from source s by coordinate sequence CS follow a matching process along dimension 0 to dimension $n-1$. In Figure 2.9, source 0000 and destination 1011 differ in 3 dimensions $\{0, 1, 3\}$ and similar in dimension 2 only. We can shift the bits of input sequence one by one cyclically from left. The node-disjoint paths are as follows:

- 1st path with bit-flip sequence $\langle 0, 1, 3\rangle$ is $(0000, 0001, 0011, 1011)$;
- 2nd path with bit-flip sequence $\langle 1, 3, 0\rangle$ is $(0000, 0010, 1010, 1011)$;
- 3rd path with bit-flip sequence $\langle 3, 0, 1\rangle$ is $(0000, 1000, 1001, 1011)$;
- 4th path with bit-flip sequence $\langle 2, 3, 0, 1, 2\rangle$ is $(0000, 0100, 1100, 1101, 1111, 1011)$.

Out of the four paths given, three of them are of length three, while one of them is of length five.

In hypercube routing, the coordinate sequence CS of a path is sent along with the message m. After a successful forwarding along dimension i, dimension i will be deleted from the sequence. Finally, the sequence becomes empty upon reaching the destination group.

According to the hypercube property [107], these multiple node-disjoint paths are composed of n shortest paths of length n when the source and the destination differ in n dimensions in an n-dimensional hypercube. All of these paths are generated from the coordinate sequence CS. The benefit of the node-disjointness is that the paths will not cross each other, which increases the efficiency of the routing.

3.4 Algorithms

The proposed algorithms traverse each path $P_i = s, \ldots, d$, where $0 \leq i \leq n-1$, with source s and destination d in the absence of faults as follows. If $s[i] = d[i]$, then message m can not be propagated and it will remain held in process s. Otherwise, the vertex after s on P_i is obtained by flipping the ith positions of s and the vertex before d on P_i is obtained by flipping the ith positions of d. Then, each remaining successive vertex on P_i is obtained by flipping the next significant bit position of the current vertex where s and d differ starting from position i, or we can say that if $s[i] \neq d[i]$, i.e. the ith positions of s and d differ, each successive vertex on P_i is obtained by flipping the next most significant bit position of the current vertex where s and d differ starting from position i.

To deal with faults, our proposed algorithm uses acknowledgement message (*acks*). Processes can fail only by crashing and when a crash is permanent, and up to $n-1$ nodes may crash at any given time. A process is correct if it does not crash during computation, otherwise, it is faulty. If sender process receives the acknowledgement message (*acks*) from the neighbourhood process, it means the neighbourhood process is correct (or non-faulty). If sender process does not receive the acknowledgement message (*acks*) from the neighbourhood process, it means the neighbourhood process is incorrect (or faulty).

If source node does not receive the acknowledgement message (*acks*) after sending a message as a result of the bit-flip operation on current node c, our proposed algorithm does not accomplish the bit-flip of the next most significant bit position x where s and d differ. Instead, it flips the next most significant bit position of c after x where s and d differ to obtain the id of the next node. Afterwards, bit-flip at position x is accomplished to obtain the next node. As a result, the same path traversed in the absence of faults is reached and followed until finding another faulty node.

The proposed algorithm sends n messages $m_0, m_1, \ldots, m_{n-1}$ from source process s to destination process d in two cases. In first case, the algorithm proceeds when there is no faults in the system and in second case when there is failure in the system. When $s \neq d$, each message $m_i, 0 \leq i \leq n-1$, is sent over a distinct node-disjoint path between s and d. For doing this, the source process s sends each message to a distinct neighbouring process with index (or dimension or bit-flip position) and target node id. The *id* of the neighbouring process is obtained from the id of s by flipping the most significant bit position. This information is used by an intermediate process to identify its successor in a node-disjoint path.

As each intermediate process identifies a bit-flip position to route a received message, a bit-flip sequence containing the sequence of bit-flip positions for each message m routed over a disjoint path is obtained. We now present the basis of the algorithm using the concept of bit-flip sequences. The proposed algorithm associates a distinct bit-flip sequence with each message m traversing a distinct disjoint path in a distributed manner to ensure the disjointness of the paths. The bit-flip sequence is associated with message m traversing a node-disjoint path such that the first bit-flip position in the sequence corresponds to the first process on the path, the second bit-flip position corresponds to the second process on the path and so on. Therefore, every intermediate process on a disjoint path has a corresponding bit-flip position in the associated bit-flip sequence.

We define two functions which are used by our proposed algorithm. Function $f_i(s, d)$ is the position of the ith bit or the dimension i, in which the labels s and d differ. For example,

$f_0(011101, 001001) = 2,$

$f_1(011101, 001001) = 4.$

Similarly, Function $g_i(s, d)$ gives the position of the ith bit or dimension i between s and d, where the bits are common. For example,

$g_0(011101, 001001) = 0$,

$g_1(011101, 001001) = 1$.

If s and d have all/no bits in common, f_i/g_i assumes the value ϕ. For example,

$f_0(001101, 001101) = \phi$,

$g_0(001101, 110010) = \phi$.

In the remainder of this section, we present the proposed algorithm in two steps. First, Algorithm 1 explains the message routing in the fault-free condition. Second, Algorithm 2 presents the routing algorithm in the presence of node/link failures [120].

3.4.1 Routing Without Failures

When a process receives a message m, it propagates the message m through functions $f_i(s, d)$ and $g_i(s, d)$ to its neighbourhood process. After receiving a message m, the neighbourhood process performs the corresponding bit-flip in the sequence to propagate the message m to the next process on the path. Upon receiving message m and the position of the bit-flip i, an intermediate process s determines the next bit-flip position b in the bit-flip sequence corresponding to the path traversed by message m. For that purpose, process s searches (using modulo-n) the next most significant bit position b after position i, wherein s and d differ by using the function $f_i(s, d)$ and function $g_i(s, d)$ provides the path where the bit position are common. Although, paths given by $g_i(s, d)$ usually are longest in comparison of provided by the function $f_i(s, d)$. Subsequently, we can find the successor process on the node-disjoint path by flipping the bth bit of process id s. This simple deterministic algorithm always finds a path of minimum length. This length is at most n, and the computation overhead of the routing process is very small. The routing algorithm is provided in Algorithm 1 for hypercubes with no crash failures. For Algorithm 1, the inputs are addresses of source and destination nodes and output is minimum length path between source and destination.

Algorithm 1 Algorithm for node-to-node routing in a hypercube

Require: s is the label of current process and d is the label of destination process.

```
1: for all i, 0 ≤ i ≤ n − 1 do
2:     send message through f_i(s, d) and g_i(s, d)
3:     if s = d then hold the message m in process s;
4:     else
5:         propagate message m to next processor through f_i(s, d) and g_i(s, d)
6:     end if
7: end for
```

3.4.2 Routing With Failures

The objective of selecting alternative path is to configure source-based routing paths between source and destination affected by node/link failures. This is an important task because designing alternative paths provides fast responses to node/link failures. The first step of this objective is the selection of suitable communication paths for source-destination pairs struck by node/link failures. Upon occurrences of a node/link failure in the system, the source node configures one or more alternative routing paths to the corresponding destination.

Now, in the case of faulty nodes in the hypercube, our algorithm uses the acknowledgement message (*acks*), when sender sends the message m, it waits for acknowledgement message (*acks*). If *acks* are received, it means neighbourhood process is non-faulty, otherwise it is faulty. When a process receives a message m, it performs the corresponding bit-flip in the sequence, as long as the identified neighbour or the link connecting the process to the neighbour is non-faulty. If the identified neighbour or the link connecting the process to the neighbour is faulty, the corresponding bit-flip sequence needs to be altered to avoid the faulty node/link, while maintaining the disjointness of the paths. The bit-flip sequence is uniformly altered to avoid each faulty node/link while maintaining the disjointness property in a distributed manner based on only local knowledge as follows.

After getting a faulty node/link, the process postpones the bit-flip that would lead the message m to the failed link/process, and instead performs the subsequent bit-flip in the sequence to send the message m to a different neighbour. The postponed bit-flip in the sequence is carried out by the neighbouring process on the altered node-disjoint

path. The proposed bit-flip sequence alteration corresponds to a swapping of two consecutive bit-flips implementing a detour of the message around the failed process/link in a distributed manner. The proposed mechanism preserves node-disjointness along with all other properties addressed in the next section.

Then, process s checks by receiving acknowledgement message (*acks*) whether or not neighbouring process or the link connecting s to neighbouring process is faulty. If sender receives the *acks*, then the process and the link are non-faulty, message m is forwarded to neighbouring process along with function $f_i(s,d)$ and $g_i(s,d)$, which consists destination id d and the computed bit-flip position b. Otherwise, process s postpones the flip at bit position b by searching (using modulo-n) the next most significant bit position b_n after position b wherein s and d differ by using the $f_i(s,d)$ function and same by using the $g_i(s,d)$ function again. Observe that the postponed bit-flip b in the sequence must be effected at some point to reach the destination process; it is effected by the next intermediate process on the path, by providing $(((i+1)\ mod\ n)-1)\ mod\ n$ to the next intermediate process as the bit-flip position, ensuring that the postponed bit-flip b is the next effected one. As a result, the algorithm successfully bypasses the faulty process/link by swapping the order of two consecutive bit-flips b and b_n in the sequence. The proposed fault-tolerant algorithm is provided in Algorithm 2. For Algorithm 2, the inputs are addresses of source and destination nodes with acknowledgement message and output is minimum length path between source and destination.

Algorithm 2 Algorithm for node-to-node routing in a faulty hypercube

Require: s is the label of current process and d is the label of destination process. *acks* is the acknowledgement message.

1: **for all** $i, 0 \le i \le n-1$ **do**
2: send message through $f_i(s,d)$ and $g_i(s,d)$
3: **if** $s = d$ **then** hold the message m in process s;
4: **else if** *acks* not received to s **then**
5: $i \leftarrow (((i+1)\ mod\ n)-1)\ mod\ n$;
6: propagate message m to next processor through $f_i(s,d)$ and $g_i(s,d)$
7: **else**
8: $i \leftarrow (i+1)\ mod\ n$
9: propagate message m to next processor through $f_i(s,d)$ and $g_i(s,d)$
10: **end if**
11: **end for**

All the paths from a source node to a destination node, among which there is the shortest path, are found by the algorithm. The found paths are fixed to avoid complication of the algorithm. The shortest path may be determined by way of comparison of the lengths of the found paths.

3.5 Proof of Correctness

Lemma 3.5.1. If the message m is correct then it should reach from source s to destination d.

Proof. If the message m is not affected, then algorithm compares the bits of d and s where they differs by function $f_i(s, d)$ and where the bits are same by function $g_i(s, d)$. By these functions one can found simple paths and each path found by the algorithm is the shortest node-disjoint path from s to d. Hence, the lemma follows. □

Lemma 3.5.2. The proposed fault-tolerant algorithm sends n correct messages via node disjoint paths from s to d.

Proof. Suppose there is no faulty adjacent nodes of s and d, then algorithm alter the bit positions through $f_i(s, d)$ and $g_i(s, d)$. Moreover, if at most one faulty node exist in the adjacent of any nodes in the hypercube, then fault-tolerant algorithm propagates the message with sufficient condition by swapping the bits within the pair of nodes. When the algorithm has a faulty node, the next altering the bits is accomplished by both the functions and effected by the adjacent node on the path of hypercube. Furthermore, these paths terminate at d by Lemma 1. Hence, the lemma follows. □

Lemma 3.5.3. Even in the presence of faulty node, correct message m traverses an optimal path length of at most $n + 1$.

Proof. When we applied both functions $f_i(s, d)$ and $g_i(s, d)$ together, then algorithm traverses path length of at most $n+1$, even in the presence of faulty node in a hypercube. Actually, it is the longest possible path in a hypercube that a message m traverses. Hence, the lemma follows. □

Lemma 3.5.4. In the proposed algorithm, each of the n normal messages sent by source process s traverses a shortest all node-disjoint path and reaches destination process d.

Proof. Both functions of the algorithm provides all node-disjoint path in all the cases. Furthermore, due to faulty nodes, there is no effect on the length of the paths and the paths are the shortest all node-disjoint paths by Lemma 2. As these messages are reached from s to d by Lemma 1. Hence, the lemma follows. □

Theorem 3.5.5. In a n-dimensional H_n, the proposed algorithm passes n messages from source node s to destination node d through n all node-disjoint paths of optimal length $n + 1$.

Proof. In the proposed algorithm, each of the n messages sent by source node s to destination node d traverses a shortest all node-disjoint path by Lemma 4 and all message reached to destination node d in at most $n + 1$ round by Lemmas 3. Hence, the theorem follows. □

3.6 Evaluation Model

The evaluation of the contributions of this chapter is done with the help of Coloured Petri nets (CPN or CP-nets) tools simulator. The primary reason for selecting the CPN tools is because it is a tool for editing, modelling, simulating and validation of systems in which concurrency, communication, and synchronisation play a major role. User interaction with CPN Tools is based on direct manipulation of the graphical representation of the CPN model using interaction techniques, such as tool palettes and marking menus. A CPN model of a system is an executable model representing the states of the system and the events (transitions) that can cause the system to change state. The CPN language makes it possible to organise a model as a set of modules, and it includes a time concept for representing the time taken to execute events in the modelled system. CPN Tools is an industrial-strength computer tool for constructing and analysing CPN models. Using CPN Tools, it is possible to investigate the behaviour of the modelled system using simulation, to verify properties by means of state space methods and model checking, and to conduct simulation-based performance analysis [103, 122].

The CPN model of the hypercube topology for routing the messages from source to destination is shown in Figure 3.7. This model describes a sequence of messages is sent from the source node to the destination node through communication link where

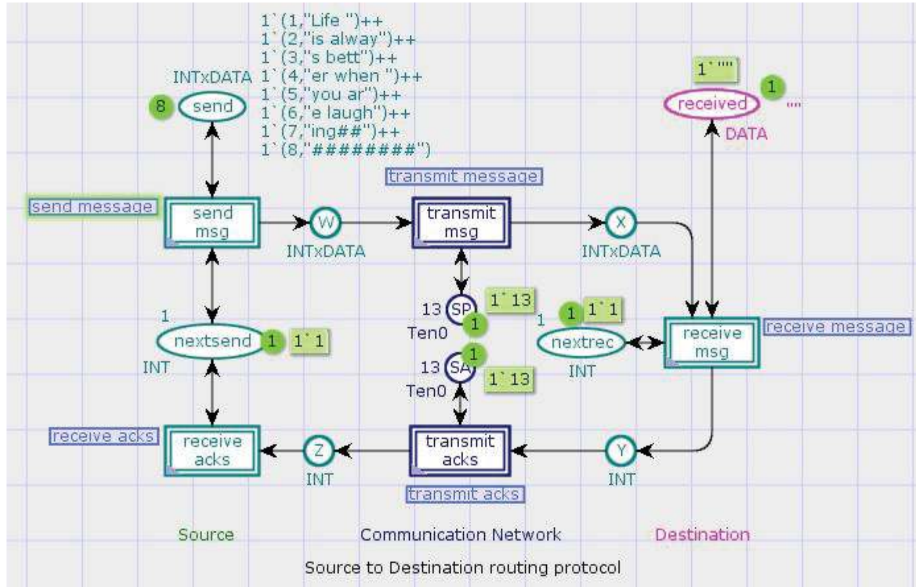

Figure 3.7: CPN model of the system.

messages may be delayed or lost due to node/link failures. In particular, the CPN model consists following three parts:

• *Source*: The source part has two transitions, one for sending the messages (say, *send msg*) and another for receiving acknowledgements (say, *receive acks*).

• *Communication Network*: The communication part has two transitions, one for transmitting the messages (say, *transmit msg*) and another for transmitting the acknowledgements (say, *transmit acks*).

• *Destination*: Finally, the destination part has a single transition in which it can receive messages and send acknowledgements (say, *receive msg*).

The interface between the source and the communication network consists of the places W and Z, while the interface between the communication network and the destination consists of the places X and Y.

The messages to be sent are positioned at the place *send*. Each token on this place contains a message number (INT type) and the data contents of the message (DATA type). The place *nextsend* contains the number of the next message to be sent. Initially this number is 1, and it is updated each time an acknowledgement is received.

The content of the received message is kept at the place *received*. This place contains

a single token with a text string (DATA type) which is the concatenation of the text strings contained in the received messages. Initially the text string at *received* is empty, i.e., "". At the end of the transmission we expect *received* to contain the text string "Life is always better when you are laughing". The place *nextrec* contains the number of the next message to be received. Initially this number is 1, and it is updated each time a message is successfully received.

We do not model how the source splits a messages into a sequence of messages or how the destination reassembles the messages into a message. Neither do we model how the tokens at source and destination are removed at the end of the transmission or how the message numbers in *nextsend* and *nextrec* are reset to 1. Now let us take a closer look at the five different transitions in the protocol system.

1. *send msg* sends a message to the communication network by creating a copy of the message on place W. The number in *nextsend* specifies which message to send. It should be noted that the message is not removed from source. Neither is the counter at *nextsend* increased. The reason is that the message may be lost due to faulty node/link and hence need to be retransmitted. Our protocol is pessimistic, in the sense that it continues to repeat the same message - until it gets an acknowledgement telling that the message has been successfully received.

2. *transmit msg* transmits a message from the source site of the hypercube network to the destination site by moving the corresponding token from W to X. The boolean expression $Ok(s, r)$ determines whether the message is successfully transmitted or lost. The variable r will be bound to an arbitrary value in its colour set (i.e., to any integer between 1 and 15). CPN Tools makes a fair choice between the 15 values. The Ok function returns true if the value of r is less than or equal to the value of s. This means that the probability of successful transmission is determined by the token at place SP. We have given SP a token with value 13. Hence we have 87% chance for successful transmission. However, it is easy to modify the success rate, simply by changing the token value at SP.

3. *receive msg* receives a message and checks whether the message number n is identical to the number k in *nextrec*. When the two numbers match, the number in *nextrec* is increased by 1 and the text string in the message is concatenated

to the text string in *received* unless it is stop = “♯♯♯♯♯♯♯♯”, which by convention indicates end-of-message. Otherwise, the message is ignored and the number in *nextrec* is left unchanged. In both cases an acknowledgement is sent containing the number of the next message which the source should send.

4. *transmit acks* transmits an acknowledgement from the destination site of the hypercube network to the source site by moving the corresponding token from Y to Z. The transition works in a similar way as *transmit msg*. This means that the acknowledgement may be lost, with a probability determined by the token at place SA.

5. *receive acks* receives an acknowledgement and updates the number in *nextsend* by replacing the old value with the one contained in the acknowledgement.

After executing few number of steps, the CPN model may have reached the intermediate state as shown in Figure 3.8. From Figure 3.8 the source is sending message number two and two copies of this message are available at places W and X. The string “Life is alway” has been *received*. This is the lists of the first two messages and now the *destination* is waiting for message number three. Thus, the messages on W and X will be ignored when they arrive the *destination*. One acknowledgement is present at place Z. When *receive acks* occurs, *nextsend* will be updated to three, and then *source* will start sending message number three.

After executing all the steps, the *destination* received all the messages successfully. When the last message with “♯♯♯♯♯♯♯♯” is received, *nextrec* gets the value 9. This value will be communicated to the *source* via *transmit acks*, *nextsend* will be updated to 9, and the sending will stop because no message with this number exists. The final CPN model after executing all steps as shown in Figure 3.9.

For easy representation of CPN model, one can organize it in multiple nets. Now we will discuss all five transitions one by one. We add one extra place, named *sendmsg* in *send msg* transition. It describes the number of individual messages have been sent. Figure 3.10 shows that message number one has been sent 5 times, message number two 4 times, message number three 12 times, message number four 4 times, message number five 6 times, message number six 11 times, message number seven 15 times and message number eight 17 times.

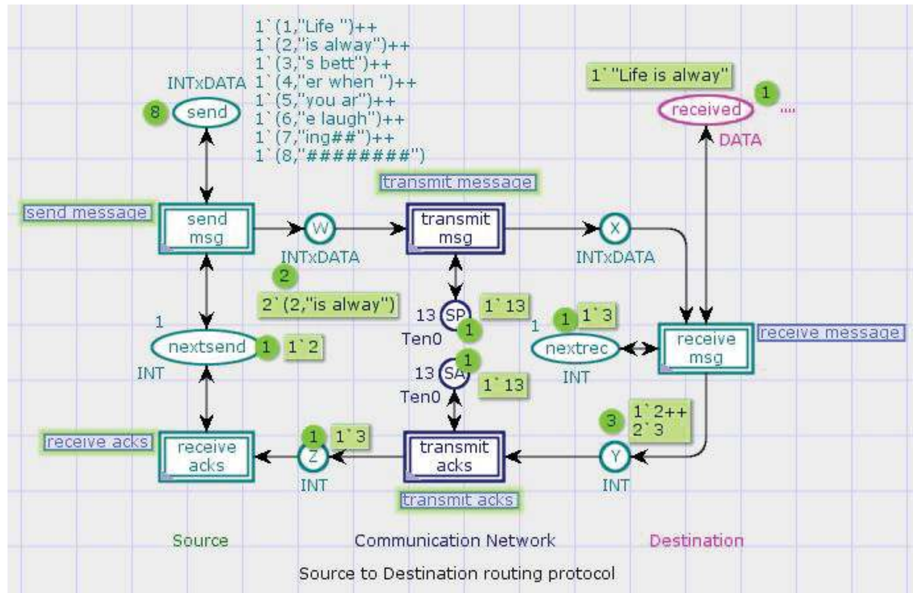

Figure 3.8: CPN model of the system after few steps.

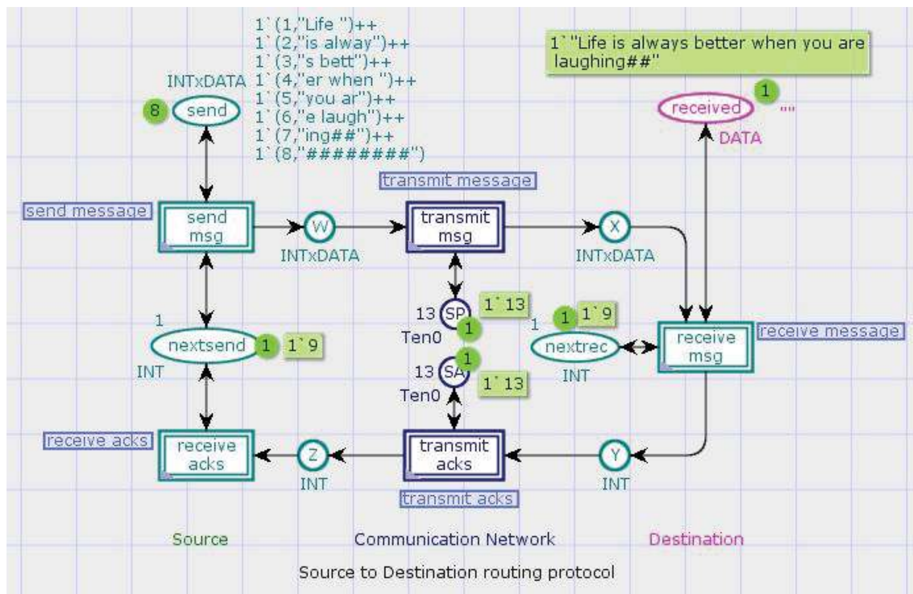

Figure 3.9: CPN model of the system after all steps.

Then we add one extra place, named *count* in *receive acks* transition. It provides the sequence number for acknowledgements as shown in Figure 3.11.

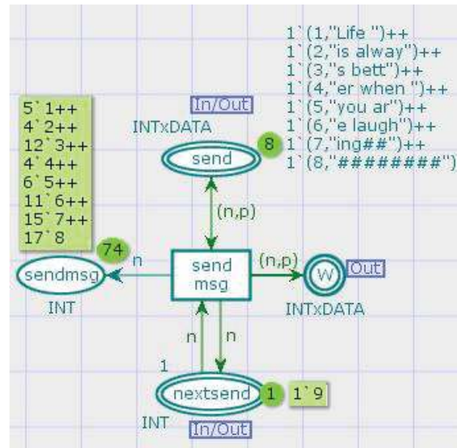

Figure 3.10: CPN model for the Sender Side.

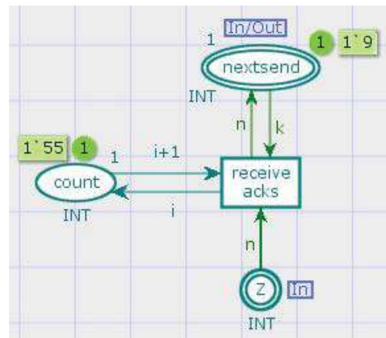

Figure 3.11: CPN model of the Receiver acks.

The place *lostmsg* in transition *transmit msg* says about the lost messages due to node/link failures. In Figure 3.12, we have lost two copy of message number 3, only one copy of message number 4, three copy of message number 6 and two copy of message number 7.

The place *lostacks* in transition *transmit acks* says how many acknowledgements have lost/transmitted. In Figure 3.13, we have lost 12 acknowledgement and successfully transmitted 54.

The place *recmsg* in transition *receive msg* says about successfully received messages. In Figure 3.14, first we received message number 1 with data “Life ”, then message number 2 with data “is alway”, message number 3 with data “s bett”, message number 4 with data “er when ”, message number 5 with data “you ar”, message number 6 with

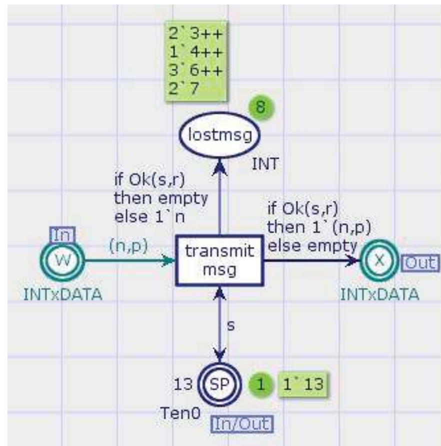

Figure 3.12: CPN model of the Transmit Message.

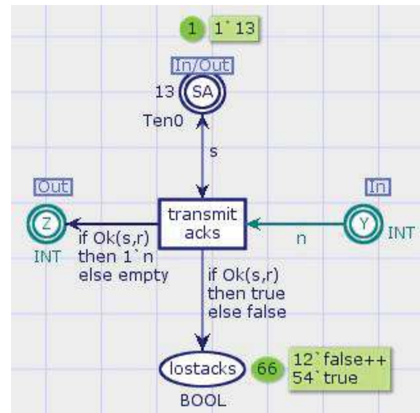

Figure 3.13: CPN model of the Transmit acks.

data "e laugh", message number 7 with data "ing ♯♯", and finally message number 8 with data "♯♯♯♯♯♯♯♯".

3.7 Implementation & Simulation Results

The algorithm has been tried statically for hypercubes of various dimensions. The following assumptions are made to test the correctness of the algorithm.

- Source node of the hypercube sends messages to each neighbour node to check the availability of nodes.

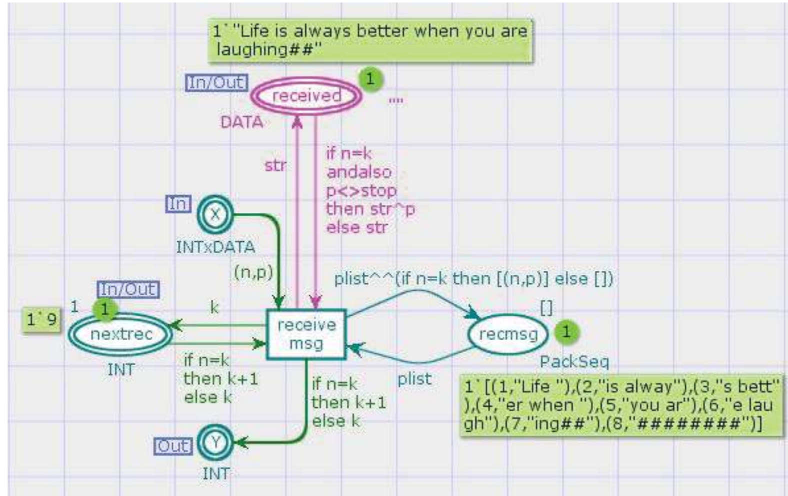

Figure 3.14: CPN model of the receiver.

- If one or more nodes are faulty, discard these nodes permanently.
- The fault tolerant routing algorithm executed at each non-faulty node then determines the correct routes.

We have implemented this algorithm using the CPN Simulator running over 32-bit Windows operating system with core i7 and 4 GB RAM for various hypercube dimensions and injected faults at various nodes and shown that the algorithm provides correct routes in the event of failed nodes.

The proposed algorithm is compared with two routing algorithms– depth first search algorithm and unsafety algorithm. The depth first search routing algorithm is based on graph search and the unsafety routing algorithm is based on vectors. The proposed algorithm is simulated for 30, 40, 50 & 60% of faulty nodes in a 8-dimensional hypercube network (H_8) and got results. Figure 3.15, Figure 3.16, Figure 3.17 and Figure 3.18 presents simulation results of average message latency. Each diagram presents results for different message lengths 60, 80, 100, 120, 140, 160, 180 and 200. Due to, the most important issues in the networks is the average of message latency, we focused on this factor in the simulations. In Figure 3.15, when the number of faulty nodes are about 30% in H_8. Our algorithm provides an average 14% improvement in the performance of the network. In Figure 3.16, when the number of faulty nodes are about 40% in

H_8. Our algorithm provides an average 12% improvement in the performance of the network. In Figure 3.17, when the number of faulty nodes are about 50% in H_8. Our algorithm provides an average 20% improvement in the performance of the network. In Figure 3.18, when the number of faulty nodes are about 60% in H_8. Our algorithm provides an average 14% improvement in the performance of the network. With this analysis we can observe that our proposed routing algorithm with different number of faulty nodes and different message lengths provides better average message latency and increases the reliability of the network.

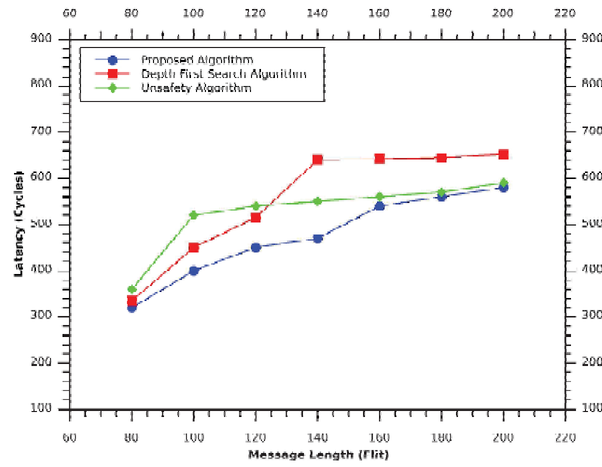

Figure 3.15: Simulation result for different message lengths with 30% faulty nodes in H_8.

Figure 3.19, Figure 3.20, Figure 3.21 and Figure 3.22 presents simulation results of average message latency for H_8 with 30, 40, 50 & 60% faulty nodes, respectively. Each diagram presents results for different generation rates. In Figure 3.19, when the number of faulty nodes are about 30% in H_8. Our algorithm provides an average 7% improvement in the performance of the network. In Figure 3.20, when the number of faulty nodes are about 40% in H_8. Our algorithm provides an average 5% improvement in the performance of the network. In Figure 3.21, when the number of faulty nodes are about 50% in H_8. Our algorithm provides an average 4% improvement in the performance of the network. In Figure 3.22, when the number of faulty nodes are about 60% in H_8. Our algorithm provides an average 4% improvement in the performance of the network. Our approach provides better performance when we are increasing the

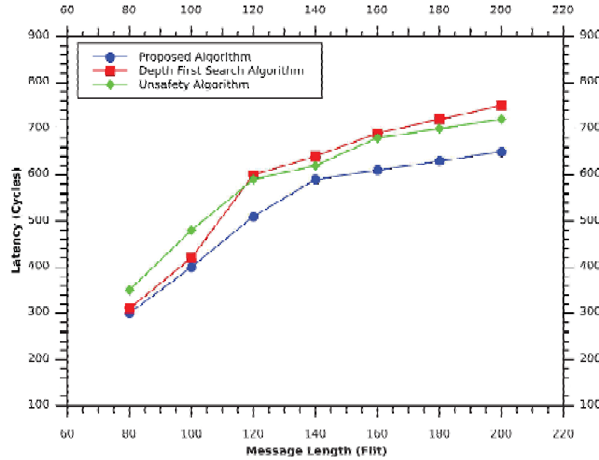

Figure 3.16: Simulation result for different message lengths with 40% faulty nodes in H_8.

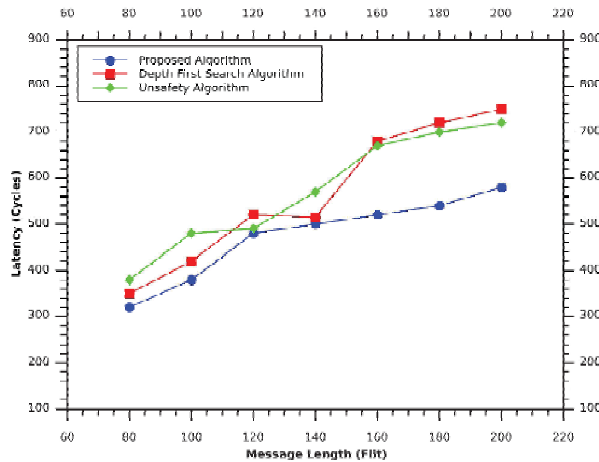

Figure 3.17: Simulation result for different message lengths with 50% faulty nodes in H_8.

generation rate. With this analysis we can observe that the proposed algorithm has the more reliability than previous algorithms. It also has the accepted performance comparing with them.

Simulation results confirm that the proposed node-to-node routing algorithm provides an average of 10% improvement in the performance of hypercube network in comparison with the previously proposed routing algorithms–depth first search algo-

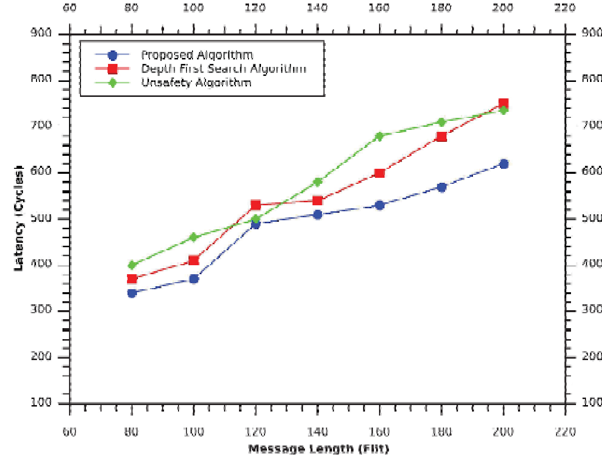

Figure 3.18: Simulation result for different message lengths with 60% faulty nodes in H_8.

rithm and unsafety vectors algorithm.

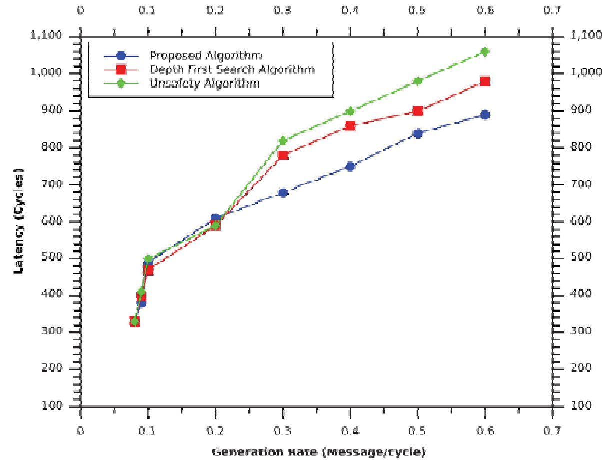

Figure 3.19: Simulation results for different generation rates with 30% faulty nodes in H_8.

The routing algorithm has been applied to the following examples and how the algorithm finds the unique path is illustrated—

Example 3.7.1. Consider the source node $s = (0000)$, destination node $d = (1011)$ and faulty nodes $\{f_1 = 0001, f_2 = 0011, f_3 = 0100, f_4 = 0101, f_5 = 0111, f_6 = 1010, f_7 =$

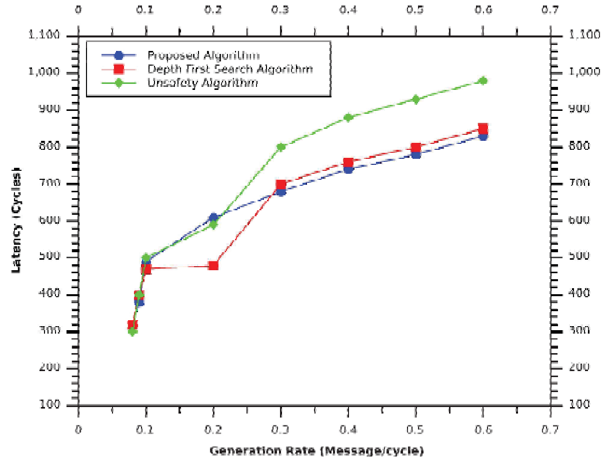

Figure 3.20: Simulation results for different generation rates with 40% faulty nodes in H_8.

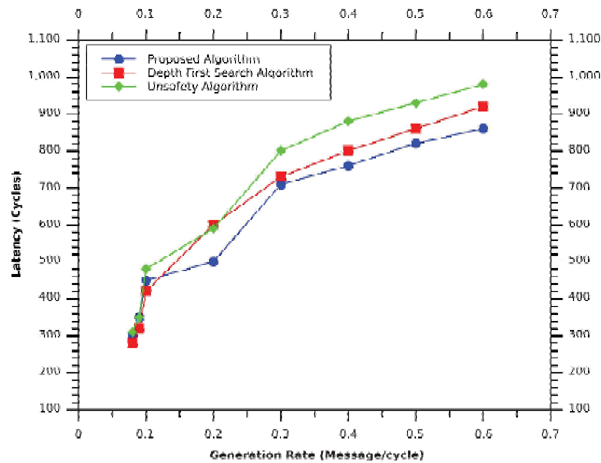

Figure 3.21: Simulation results for different generation rates with 50% faulty nodes in H_8.

$1100, f_8 = 1110\}$ in H_4. This problem provides the maximum 4 node-disjoint paths, but we are looking for shortest node-disjoint non-faulty paths. So the unique possible shortest computed path according to the algorithm is $P(0000, 1011) = \{3, 0, 1\} = 0000 \rightarrow 1000 \rightarrow 1001 \rightarrow 1011$.

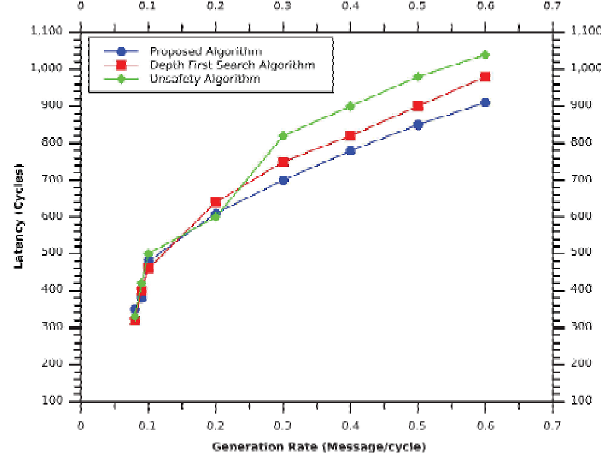

Figure 3.22: Simulation results for different generation rates with 60% faulty nodes in H_8.

Example 3.7.2. Consider the source node $s = (000000)$, destination node $d = (111001)$ and faulty nodes $\{f_1 = 000001, f_2 = 000010, f_3 = 000011, f_4 = 000100, f_5 = 000101, f_6 = 000110, f_7 = 000111, f_8 = 001010, f_9 = 001100, f_{10} = 010111, f_{11} = 011011, f_{12} = 011100, f_{13} = 100000, f_{14} = 110101, f_{15} = 111010, f_{16} = 111101, f_{17} = 111110, f_{18} = 111111\}$ in H_6. This problem provides the maximum 6 node-disjoint paths, but we are looking for shortest node-disjoint non-faulty paths. So there are two possible shortest paths according to the algorithm. The computed first path $P(000000, 111001) = \{3, 4, 5, 0\} = 001000 \rightarrow 011000 \rightarrow 111000 \rightarrow 111001$ and computed second path $P(000000, 111001) = \{4, 5, 0, 3\} = 000000 \rightarrow 010000 \rightarrow 110000 \rightarrow 110001 \rightarrow 111001$.

Example 3.7.3. Consider the source node $s = (00000000)$, destination node $d = (10101010)$ and faulty nodes $\{f_1 = 00000010, f_2 = 00000100, f_3 = 00001010, f_4 = 00011011, f_5 = 00101000, f_6 = 00110101, f_7 = 00111100, f_8 = 01010111, f_9 = 01011111, f_{10} = 10000000, f_{11} = 10001000, f_{12} = 10001010, f_{13} = 10100101, f_{14} = 10111001, f_{15} = 10111101, f_{16} = 10111111, f_{17} = 11100011, f_{18} = 11100111, f_{19} = 11110000, f_{20} = 11111000, f_{21} = 11111001, f_{22} = 11111100\}$ in H_8. This problem provides the maximum 8 node-disjoint paths, but we are looking for shortest node-disjoint non-faulty paths. So the unique possible shortest computed path according to the algorithm is $P(00000000, 10101010) = \{5, 7, 1, 3\} = 00000000 \rightarrow 00100000 \rightarrow 10100000 \rightarrow 10100010 \rightarrow 10101010$.

Our algorithm generates n paths which are mutually node-disjoint in the case when there is no failures in the system. The node-disjoint property follows from the fact that, we are computing the path from the source node to the destination node and at any stage the hypercube has two subcubes: the 0-subcube and the 1-subcube. Since n-dimensional hypercubes have n links connecting n nodes, so in the presence of failures they choose the alternative paths with the help of functions. Thus, the paths computed at any stage of recursion cannot intersect with each other. The advantageous of node-disjoint paths is if a task tries to move to a faulty node, it will try an alternative route.

3.8 Conclusion

Throughout this chapter, we have described the proposed fault-tolerant node-to-node adaptive routing algorithm. In this chapter an optimal algorithm is proposed that determines the shortest node-disjoint paths from a source node to any destination node. Our algorithm allows to find all the paths between source to destination and is independent of the number of faulty nodes and links in a hypercube networks. For dealing the faults, acknowledgement messages (*acks*) are included in the proposed algorithm for routing messages over node-disjoint paths in a hypercube network. The task of the algorithm may be finished after determining the first path, the length of which is equal to Hamming distance. The algorithm is effective to provide fault tolerance for the hypercube system, which may have a great number of faulty nodes and, or links.

Our proposed algorithm rely on dynamic reconfiguration process to deal failures in parallel systems without halting the computing system. The proposed algorithm is based on multipath adaptive routing method and needs no virtual channels for deadlock avoidance. On the other hand, the routing methodology proposed by Gomez et al. [49] and Dong Xiang [126] uses static fault mode. In this mode, the system must be restarted. For this new routes are calculated and system restarts from the last safe state. This problem is not introduced in our proposal because our algorithm has been designed to handle with both static and dynamic components failures.

Koibuchi et al. [69] proposed a method which deals dynamic failures but uses virtual channels for routing. Actual HPC systems usually rely on simple solutions. For

instance, Titan and Blue Gene/Q supercomputers [1] may route messages either deterministically using dimension order routing (xyz) or dynamically, but the hardware does not have the capacity to route around faulty nodes and/or links.

This approach may be applied to the routing algorithms with local information about faulty nodes and links and also to broadcasting and multicasting algorithms and task allocation problems.