IDENTIFICATION OF CRASH FAULT & VALUE FAULT FOR RANDOM NETWORK IN DYNAMIC ENVIRONMENT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology

in

Computer Science and Engineering

By

SANDEEP PANDA(10506011)

ARINDAM NAYAK(10506032)

Department of Computer Science and Engineering

National Institute of Technology

Rourkela

2009

IDENTIFICATION OF CRASH FAULT & VALUE FAULT FOR RANDOM NETWORK IN DYNAMIC ENVIRONMENT

A THESIS SUBMITTED IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF

Bachelor of Technology
In
Computer Science and Engineering

By:
Sandeep Panda
Roll No.: 10506011
Arindam Nayak
Roll No.: 10506032

Under The Guidance of:
Prof Pabitra Mohan Khillar

Department of Computer Science and Engineering
National Institute of Technology, Rourkela
May, 2009

National Institute of Technology

Rourkela

CERTIFICATE

*This is to certify that the thesis entitled, "**IDENTIFICATION OF CRASH FAULT & VALUE FAULT FOR RANDOM NETWORK IN DYNAMIC ENVIRONMENT**" submitted by Sri. Sandeep Panda and Sri. Arindam Nayak in partial fulfillment of the requirements for the award of Bachelor of Technology Degree in Computer Science and Engineering at National Institute of Technology, Rourkela (Deemed University) is an authentic work carried out by them under my supervision and guidance.*

To the best of my knowledge, the matter embodied in the thesis has not been submitted to any other University / Institute for the award of any Degree or Diploma.

Date:

Prof. P.M. Khillar
Dept .of Computer Science and Engineering
National Institute of Technology
Rourkela – 769008

ACKNOWLEDGEMENT

We would like to articulate our deep gratitude to our project guide Prof. P. M. Khillar, Department of Computer Science and Engineering, National Institute of Technology, Rourkela, for his valuable guidance and timely suggestions during the entire duration of our project work, without which this work would not have been possible. He has always been our motivation for carrying out the project.

We would also like to convey our deep regards to all other faculty members and staff of Department of Computer Science and Engineering, NIT Rourkela, who have bestowed their great effort and guidance at appropriate times without which it would have been very difficult on our part to finish this project work.

It is our pleasure to refer Microsoft Word exclusive of which the compilation of this report would have been impossible. An assemblage of this nature could never have been attempted with out reference to and inspiration from the works of others whose details are mentioned in reference section. We acknowledge out indebtedness to all of them. Last but not the least, our sincere thanks to all of our friends who have patiently extended all sorts of help for accomplishing this undertaking.

Sandeep Panda
Roll No: 10506011

Arindam Nayak
Roll No: 10506032

Contents

ABSTRACT

During the past few years distributed systems have been the focus of considerable research in computer science. Fault tolerance in distributed systems is a wide area with a significant body of literature that is vastly diverse in methodology and terminology. Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults. An extensive methodology has been developed in this field over the past few years, and a number of fault-tolerant machines have been developed but most dealing with random hardware faults, while a smaller number deal with software, design and operator faults to varying degrees.

Our work mainly focuses on the simulation of the system that deals with software faults means the faults that occur because of the failure or error in the internal software component. Our work is restricted to distributed diagnosis in dynamic fault environment. Basically we have created different not-completely connected random networks with number of nodes ranging from 8 to 256.Then we have induced faults to these networks dynamically using poison distribution. Three different algorithms have been implemented to detect the faults and the comparison among these algorithms, based on delay latency and number of message exchanges, has been represented graphically. The software faults that we had dealt with are crash fault and value fault in a distributed system (not-completely connected network). Although many researches have been done in the crash fault area but very less work has been done in diagnosing the value faults in dynamic fault environment.

List of Figures

List of Tables

CHAPTER 1
INTRODUCTION

1.1 INTRODUCTION

Distributed systems are those systems where the computer programming and the data to be worked on are spread out over more than one computer, usually over a network. Prior to low-cost computer power on the desktop, computing was centralized. Although such centers still exist, distribution networking applications and data operate more efficiently over a mix of desktop workstations, local area network servers, regional servers, Web servers, and other servers. One of the most critical characteristics of any distributed system is Fault Tolerance. Fault tolerance is the ability of a system to perform its function correctly even in the presence of internal faults [5]. The purpose of fault tolerance is to increase the dependability of a system. Fault-tolerant computing is the art and science of building computing systems that continue to operate satisfactorily in the presence of faults. Fault tolerance and dependable systems research covers a wide spectrum of applications ranging across embedded real-time systems, commercial transaction systems, transportation systems, and military/space systems, to name a few. The supporting research includes system architecture, design techniques, coding theory, testing, validation, proof of correctness, modeling, software reliability, operating systems, parallel processing, and real-time processing. These areas often involve widely diverse core expertise ranging from formal logic, mathematics of stochastic modeling, graph theory, hardware design and software engineering.

Our work basically deals with Software fault tolerance for not completely connected network. Software fault tolerance is the ability for software to detect and recover from a fault that is happening or has already happened in either the software or hardware in the system in which the software is running in order to provide service in accordance with the specification. Software fault tolerance is a necessary component in order to construct the next generation of highly available and reliable computing systems from embedded systems to data warehouse systems [7]. Software faults are all design faults. The software faults are the result of human error in interpreting a specification or correctly implementing an algorithm or program bugs that creates issues which must be dealt with in the fundamental approach to software fault tolerance.

We should accept that, relying on software techniques for obtaining dependability means accepting some overhead in terms of increased size of code and reduced performance. Also as the number of nodes increases the performance of the system slows down accordingly.

1.2 BASIC CONCEPTS

Fault tolerance is the property that enables a system to continue operating properly in the event of failure of some of the components. If its operating quality decreases at all, the decrease is proportional to the severity of the failure, as compared to a naively-designed system in which even a small failure can cause total breakdown.

Example-The Transmission Control Protocol (TCP) is designed to allow reliable two-way communication in a packet-switched network, even in the presence of communications links which are imperfect or overloaded. It does this by requiring the endpoints of the communication to expect packet loss, duplication, reordering and corruption, so that these conditions do not damage data integrity, and only reduce throughput by a proportional amount [9].

Implicit in the definition of fault tolerance is the assumption that there is a specification of what constitutes correct behavior. A failure occurs when an actual running system deviates from this specified behavior. The cause of a failure is called an error. An error represents an invalid system state, one that is not allowed by the system behavior specification. The error itself is the result of a defect in the system or fault. In other words, a fault is the root cause of a failure. That means that an error is merely the symptom of a fault. A fault may not necessarily result in an error, but the same fault may result in multiple errors.

Fault tolerant computing is the science of building computing systems that continue to operate satisfactorily in the presence of faults. A fault-tolerant system may be able to tolerate one or more fault-types such as i) transient, intermittent or permanent hardware faults, ii) software and hardware design errors, iii) operator errors, or iv) externally induced upsets or physical damage.

When ever any fault occurs in distributed environment, there are 10 different response stages to deal with the fault [10]. The 10 system fault response stages as follows

- **Fault confinement**- Limit the scope of fault affection into local area, or protect other areas of the system from getting contaminated by this fault.
- **Fault detection**- Locate the fault.
- **Fault masking**-Also called static redundancy. Fault masking techniques hide the effects of failures through the means that redundant information outweighs the incorrect information.

- **Retry** - In some cases, a second attempt to a operation is effective enough, especially for those transient faults which cause no physical damage.

- **Diagnosis** -Diagnosis stage becomes necessary when detection could not provide fault location and other fault information. Here we determine the type of fault occurred.
- **Reconfiguration**-The system may be able to reconfigure its components to replace the failed component or to isolate it from the rest of the system. The component may be replaced by backup spares.
- **Recovery-**After detection and maybe reconfiguration, the effects of errors must be eliminated. Normally the system operation is backed up to some point in its processing that preceded the fault detection, and operation recommences from this point.
- **Restart**- This might be possible in the case too much information is damaged by an error, or if the system is not designed for recovery.
- **Repair-** Replace the damaged component. It can be either off-line or on-line.
- **Reintegration**-After all, the repaired the device or module is reintegrated into the system.

Out of the above ten steps depicted we are only interested in fault diagnosis. Because the recovery of any system from failure is dependent upon the how fast the failure is located by the system. Basically all our work is confined to not-completely connected networks. Not-completely connected network [1] means those networks in which there is no direct connection for each and every node, unlike fully connected network. In not-completely connected network intermediate nodes relay messages between some source-destination pairs.

In the following chapters first we will be describing the related theory behind Fault tolerant computing and what are the works done till now in the area of distributed diagnosis in dynamic fault environment in chapter 2. Then in the next chapter, chapter 3 we will be describing what are the work that we have done towards distributed diagnosis and describe the algorithms implemented. In last chapter, chapter 4 we will be focusing on the output of our project such as comparison among different algorithms based upon latency and number of message exchange and their graphical representation and finally the conclusion.

1.3 MOTIVATION

Fault tolerant computing has gain importance over time due to increase in use of distributed system and increase in demand for reliability at different components. However, the present solutions available on fault diagnosis in distributed system are only applicable in diagnosing crash fault and are static in nature [4]. Hence present solutions can not be applicable to dynamic environments such as mobile adhoc network. So this problem has motivated us to simulate different algorithms in order to effectively diagnose the faults specially value faults in distributed dynamic environment.

1.4 OBJECTIVE

- To create random networks with different number of nodes ranging from 8 to 256 and to dynamically induce random number of faults into these networks using poison distribution.
- To implement different routing algorithms in the simulated system in order to diagnose the induced faults.
- To compare the number of message exchanges and the latency required to diagnose the fault for the implemented algorithms by considering different number of nodes (8-256).
- To be implemented in JAVA platform.

CHAPTER 2

BACKGROUND

2.1 CLASSIFICATION OF FAULTS

Research in fault-tolerant distributed computing aims at making distributed systems more reliable by handling faults in complex computing environments. Moreover, the increasing dependence of society on well-designed and well-functioning computer systems has led to an increasing demand for dependable systems, systems with quantifiable reliability properties. The faults in a system can be classified based on either duration or behavior or cause of the fault [2].

Based on duration, faults can be classified as transient or permanent. A **transient fault** will eventually disappear without any apparent intervention, whereas a **permanent fault** will remain unless it is removed by some external agency.

Another different way to classify faults is by their underlying cause. **Design faults** are the result of design failures, like our coding example above. While it may appear that in a carefully designed system all such faults should be eliminated through fault prevention, this is usually not realistic in practice. **Operational faults**, on the other hand, are faults that occur during the lifetime of the system and are invariably due to physical causes, such as processor failures or disk crashes.

Finally, based on how a failed component behaves once it has failed, faults can be classified into the following categories:

- **Crash faults** --the component either completely stops operating or never returns to a valid state.
- **Omission faults** -- the component completely fails to perform its service.
- **Timing faults** -- the component does not complete its service on time.
- **Value faults** -- these faults when a node sends erroneous value to another.

Out of the above stated different states of classification we are mainly interested in crash fault and value fault. All the previous works focus on component failure (crash fault) where as our work is based on dealing with value faults in dynamic fault environment. The algorithms works for any number of nodes can change their state during the execution of the algorithm provided the network remains connected and also there is a limit on how frequently an individual node can change state.

2.2 GENERAL APPROACH

An important problem in distributed systems that are subject to component failures is the distributed diagnosis problem. In distributed diagnosis, each working node must maintain correct information about the status (working or failed) of each component in the system. Here we consider the problem of achieving diagnosis despite dynamic failures and repairs. Previous work has almost exclusively dealt with the static fault situation wherein statuses of nodes remain fixed for as long as it takes an algorithm to completely diagnose the system. While a few works have attempted to consider dynamic events, no formal models have been developed and so correctness proofs and algorithm evaluations inevitably have reverted to the use of static models.
This notion of correctness, referred to as bounded correctness, consists of three properties: bounded diagnostic latency, bounded start-up, and accuracy.

For **bounded diagnostic latency**, all working nodes must learn about each event (node failure or repair) within a bounded time L. For **bounded start-up** nodes that recover must determine a valid state for every other node within time S of entering the working state. Finally, **accuracy** ensures that no spurious events are recorded by any working node [1].

Before proceeding towards implementation area first we have to take the following assumptions.

2.2.1 ASSUMPTIONS

We consider not-completely connected networks where intermediate nodes relay messages between some source destination pairs. The number of node failures is limited such that the network remains connected at all times [2]. Diagnosis algorithm can use either unicast or multicast communication. We also assume a synchronous system in which the communication delay is bounded. This is an implicit assumption in all prior work on distributed diagnosis. Nodes directly connected by a communication link are called neighbors.

We consider crash and value faults in nodes. Links are assumed to be fault free. The network delivers messages reliably. A faulty node perform it's computation like a non faulty node but it may fail to send its value (crash fault) or it may send an erroneous value to another node (value fault). The status of a node is modeled by a state machine with two states, failed and working (i.e., 1 or 0). Working nodes execute the normal workload and diagnosis procedure. Since node failures and repairs are independent in this model, there are no restrictions either on the number of nodes that are in the failed state at any one time or on the number that can fail (or recover) at the same instant.

2.2.2 SOME TEMINOLOGIES

K-Connected Network- We define the connectivity of the network k is the minimum number of nodes, the removal of which can cause the network to become disconnected [1].

Send initiation time- It is the time between a node initiating a communication and the last bit of the message being injected into the network [2].

Minimum and maximum message delays- These are the minimum and maximum times, respectively, between the last bit of a message being injected into the network and the message being completely delivered at a working neighboring node [3].

State holding time- It is the minimum time that a node remains in one state before transitioning to the other state [1].

Bounded Correctness- The goal is for working nodes to learn about every event in the system as quickly as possible, to have their views of other nodes be out of date by only a bounded amount, and to not detect any spurious events. Bounded Correctness collectively refers to three properties such as bounded diagnostic latency, bounded start-up and accuracy [1].

2.2.3 A GENERAL MODEL

The fault models that are proposed previously follow a general model to send and receive messages from their neighboring nodes. Each message has the following fields

Node id: The ID of the node that initiated the heartbeat.

Sequence no: The physical sequence number of the heartbeat.

Value: Value associated with the message.

Delay: The minimum time the heartbeat message was in the network before being received.

The status messages are propagated throughout the network. When node A receives a new status message from node B, node A stores the status message in a buffer replacing any earlier message from node B. If node A times out waiting for node B's next status message, then node B's status message is removed from node A's buffer and node B is diagnosed as faulty by node A. Hence, the presence of node B's message in node A's buffer indicates node A believes that node B is working. If a neighbor of node A recovers, then node A sends the newly recovered neighboring node all messages stored in its buffer. This ensures propagation of status messages in a dynamic fault environment. When a node B initiates a new status message, it initializes the delay field to the minimum delay that will be encountered before the messages could reach its neighboring nodes and then sends them out. Also, at the same time, node B stores this new status message in its local buffer with the delay field set to zero. Nodes keep track of the amount of time each status message is stored in their buffers. For a message stored locally in the

originating node, the delay field will always be zero. When a node retransmits or relays a status message, it adds to the delay field the length of time the message was stored in its buffer and the minimum time it takes to traverse the next hop to reach the neighboring node before sending it out. Thus, a node keeps track of the minimum length of time the status message was stored in its buffer have existed in the network.

2.3 RELATED WORKS

The bulk of the work in system diagnosis has assumed a static fault situation, i.e., the statuses of nodes do not change during execution of the diagnosis procedure. Some works have considered the dynamic situations but with assumptions such as existence of centralized diagnosis entity and regular network topology. The relevant algorithm such as Hi-ADSD and its variants have latencies of at least ($\log_2 n$) rounds [1], [2], [3]. However, these algorithms do not allow both failure and recovery events in dynamic fault environment and assumes a fully connected network. Recently, in Forward Heartbeat algorithm [4] the authors have proposed an algorithm in dynamic fault environment but assumes crash fault model. However, the authors have not considered more realistic fault model of value faults in nodes which may frequently occur in runtime due to incorrect computations while executing the distributed workload such as clock monitoring process, load balancing process etc. Then there is the problem of distributed diagnosis for not completely connected networks (DDNCN) [2] in dynamic fault environment under more realistic fault models such as crash and value fault has been investigated. The algorithm works for any number of nodes can change their state during the execution of the algorithm provided the network remains connected and also there is a limit on how frequently an individual node can change state.

CHAPTER 3

FAULT DIAGNOSIS ALGORITHM & ITS IMPLEMENTATION

3.1 SOCKET PROGRAMMING IN JAVA

3.1.1 SOCKETS

Normally, a server runs on a specific computer and has a socket that is bound to a specific port number. The server just waits, listening to the socket for a client to make a connection request.

On the client-side: The client knows the hostname of the machine on which the server is running and the port number on which the server is listening. To make a connection request, the client tries to rendezvous with the server on the server's machine and port. The client also needs to identify itself to the server so it binds to a local port number that it will use during this connection. This is usually assigned by the system.

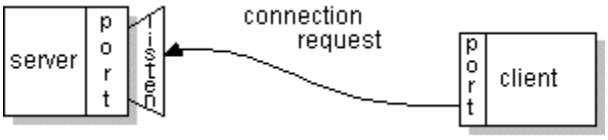

Figure 3.1 :Connection request

If everything goes well, the server accepts the connection. Upon acceptance, the server gets a new socket bound to the same local port and also has its remote endpoint set to the address and port of the client. It needs a new socket so that it can continue to listen to the original socket for connection requests while tending to the needs of the connected client.

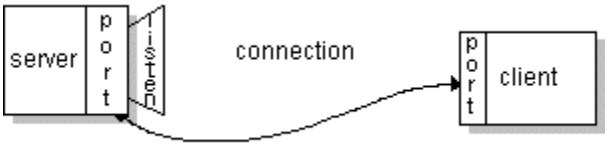

Figure 3.2 :Server response

On the client side, if the connection is accepted, a socket is successfully created and the client can use the socket to communicate with the server. The client and server can now communicate by writing to or reading from their sockets.

A socket is one end-point of a two-way communication link between two programs running on the network. Socket classes are used to represent the connection between a client program and a server program. The java.net package provides two classes--Socket and ServerSocket--that implement the client side of the connection and the server side of the connection, respectively [11].

3.1.2 STREAM COMMUNICATION

The stream communication protocol is known as TCP (transfer control protocol). Unlike UDP, TCP is a connection-oriented protocol. In order to do communication over the TCP protocol, a connection must first be established between the pair of sockets. While one of the sockets listens for a connection request (server), the other asks for a connection (client) [5]. Once two sockets have been connected, they can be used to transmit data in both (or either one of the) directions.

TCP provides a reliable, point-to-point communication channel that client-server application on the Internet use to communicate with each other. To communicate over TCP, a client program and a server program establish a connection to one another. Each program binds a socket to its end of the connection. To communicate, the client and the server each reads from and writes to the socket bound to the connection.

3.1.3 OBJECT STREAM

Just as data streams support I/O of primitive data types, object streams support I/O of objects. Most, but not all, standard classes support serialization of their objects. Those that do implement the marker interface serializable.

The object stream classes [11] are ObjectInputStream and ObjectOutputStream. These classes implement ObjectInput and ObjectOutput, which are sub interfaces of DataInput and DataOutput. That means that all the primitive data I/O methods covered in Data Streams are also implemented in object streams. So an object stream can contain a mixture of primitive and object values.

If readObject() doesn't return the object type expected, attempting to cast it to the correct type may throw a classNotFoundException. In this simple example, that can't happen, so we don't try to catch the exception. Instead, we notify the compiler that we're aware of the issue by adding ClassNotFoundException to the main method's throws clause.

3.1.4 OUTPUT AND INPUT OF COMPLEX OBJECTS

The writeObject and readObject methods are simple to use, but they contain some very sophisticated object management logic. This isn't important for a class to encapsulate primitive values. But many objects contain references to other objects. If readObject is to reconstitute an object from a stream, it has to be able to reconstitute all of the objects the original object referred to. These additional objects might have their own references, and so on. In this situation, writeObject traverses the entire web

of object references and writes all objects in that web onto the stream. Thus a single invocation of writeObject can cause a large number of objects to be written to the stream.

For example, if the following code writes an object ob twice to a stream:

```
Object ob = new Object();
out.writeObject(ob);
out.writeObject(ob);
```

Each writeObject has to be matched by a readObject, so the code that reads the stream back will look something like this:

```
Object ob1 = in.readObject();
Object ob2 = in.readObject();
```

This results in two variables, ob1 and ob2, that are references to a single object.

3.1.5 SERIALIZING OBJECT IN JAVA

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time. To persist an object in Java, we must have a persistent object. An object is marked serializable by implementing the java.io.Serializable interface, which signifies to the underlying API that the object can be flattened into bytes and subsequently inflated in the future.

The serialized objects are JVM independent and can be re-serialized by any JVM. In this case the "in memory" java objects states are converted into a byte stream. This type of the file can not be understood by the user [6]. It is special types of object i.e. reused by the JVM (Java Virtual Machine). This process of serializing an object is also called deflating or marshalling an object. Default serialization mechanism for an object writes the class of the object, the class signature, and the values of all non-transient and non-static fields.

Class ObjectOutputStream extends java.io.OutputStream implements java.io.Serializable.

From now on all our implementation/coding is done using the above fundamentals of JAVA.

3.2 IMPLEMENTAION FOR CRASH FAULT

Crash Fault: This is the type of fault where one or more number of components of the system has completely failed and hence they neither send any status message to neighbors nor receive any message from others.

We have considered not-completely connected networks. Each node runs in separate thread. To induce the crash fault for a node, we suspend the thread corresponding to that of faulty node. We have used **poisson probability distribution function** to create random number of faulty nodes dynamically. Even if some node becomes faulty still then the network remains connected.

3.2.1 Generating Random numbers using Poisson Distribution Function

This routine generates random numbers [6] according to the Poisson distribution with mean equal to

$$f(k;\lambda) = \frac{\lambda^k e^{-\lambda}}{k!},$$

Where,

e is the base of the natural logarithm (e = 2.71828...)

k is the number of occurrences of an event - the probability of which is given by the function.

k! Is the factorial of k

λ is a positive real number, equal to the expected number of occurrences that occur during the given interval.

The Poisson distribution has mean and variance λ, the routine assumes the existence of uniform random numbers on the unit interval generated by function Rand.

The method is simple conceptually but somewhat involved programmatically (for efficiency). Usually many random numbers will be drawn from the same mean, so when a new mean is encountered, the routine initializes itself by generating the cumulative probability distribution C(n) in tabular form

$$C(n) = \Sigma_{k=0,n}\, e^{-\lambda}\, \lambda^n/k!$$

For each n until C(n) becomes very close to 1. Then upon each call, a random number u uniformly distributed on the unit interval is generated, the first table entry with C(n) > u is located, and the n is returned. To achieve efficiency, a large array of pointers is used to locate the proper table entry, so that typically only one probe of the list for C(n) need be made.

Now coming back to the implementation part since faulty node can't send and receive message, all other working node must have knowledge about faulty node and work properly by detecting, excluding the faulty node.

3.2.2 Algorithm used for crash fault diagnosis

- First each node will have a table containing the status of connectivity with other nodes in the network. The table is called connection table. The table is implemented by using a 2D matrix.
- Each row of the table corresponds to each node of the network. Each column corresponds to the connectivity of that row with nodes representing the column. i.e for any element A[i,j] represents the status of node j as viewed by node i.
- The elements of the matrix will have value
    - A[i,j]=1 if i is connected to j
    - A[i,j]=0 if i is not connected to j
    - A[i,j]= -1 if i==j.
- Each node transmits its own row to every other node so that each node after receiving the row values from other nodes maintains the whole 2D matrix which is nothing but the connection table.
- Suppose node j becomes faulty then for every i, element A[i,j]=0 since node j is faulty it will not remain connected to any node i. So both the jth row and jth column will be zero.
- When at every node the table has jth column and row equals to zero then every node registers jth node as a faulty node.
- The latency to detect fault (Register a faulty node) node can be calculated by the total time required to send all the rows of the table from every node to every other node.

In this algorithm server sends to adjacent node except from the source from which it comes. For this algorithm a large number of messages are exchanged. When server sends a request from client, it checks the adjacency matrix element from corresponding row.

```
for(i=0;i<no_of_nod;i++)

    if(adjMat[req_nod][i]==1)
```

Adjacent elements found by making such comparisons. Server writes to corresponding object output stream except for source node.

[Pseudo code for flooding algorithm]

```
for each node in network
        find adjacent node
        send to node
end for
```

Now we give a simple example that depicts the above algorithm for small network and shows the connection tables at both fault free and faulty stage.

3.2.3 Example for Crash fault

Let's consider a network consisting of 4 nodes. As we can see this is a not-completely connected network.

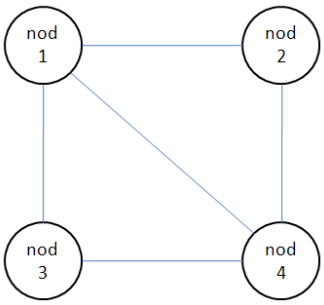

Figure 3.3 :Network before crash fault

Initially status table contents are:

	Nod1	Nod2	Nod3	nod4
nod1	-1	1	1	1
nod2	1	-1	1	1
nod3	1	1	-1	1
nod4	1	1	1	-1

Table 3.1 :Status table before crash fault

Each node sends a message continuously at 60 millisecond interval. After some time, one node (let's say node 2) becomes faulty. So that node 2 will not be able to send and receive message from any other node.

Network connection after node 2 become faulty

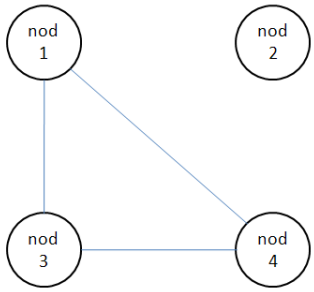

Figure 3.4 :Network after crash fault

All adjacent node of faulty node convey the information [node 2 is faulty] to all other node. Status table will be updated as given below.

	Nod1	**Nod2**	**Nod3**	**nod4**
nod1	-1	0	1	1
nod2	0	-1	0	0
nod3	1	0	-1	1
nod4	1	0	1	-1

Table 3.2 :Status table after crash fault

Then we have measured latency (in millisecond) for all nodes

	Nod1	**Nod2**	**Nod3**	**nod4**
nod1	-1	8	5	7
nod2	0	-1	0	0
nod3	4	14	-1	8
nod4	5	12	9	-1

Table 3.3 :Latency table

Since node 2 didn't receive any message latency for node is 0.For this network we measure the average latency as (8+14+12)/3=11.33.Here we have ignored -1 as node can't receive from itself.
Similarly maximum latency is 14 ms. Minimum latency is 8 ms.

For this we can conclude that maximum it will take 14 ms to detect faulty node, average time will be 11.33ms & minimum time will be 8 ms.

3.3 IMPLEMENTATION FOR VALUE FAULT

The major portion of our project covers implementing three different algorithms for detecting value fault in dynamic fault enviroment.

3.3.1 General idea for simulating fault diagnosis system

Every node in network runs on separate thread. In real environment each node is machine situated at different places. As we are simulating it in single system each node uses different port address.Each node has a client and server part.Server part of each node send message to each adjacent node.Client part receive those messages and send a request to its server to send that message to further adjacent nodes till all message circulates over the network.Each node sends messages in some specified interval to all adjacent nodes.At certain instant node is explicitly made faulty.Then the faulty node sends message to all node.Each node compares the message from faulty node with the previously received message from same node. Each node maintains a status table to know about the status of every other node & another table for maintain latency. Latency for a specified a node is defined as delay or minimum time required to receive message from concerned node.

Our main aim to fault detection in distributed environment or network with minimum latency and minimum number of message exchanges.

We have followed 3 algorithms.

- Flooding
- Variant of Flooding
- BFS

Then compared average, maximum, minimum latency for not-completely connected network consisting of 8, 16, 32, 64,128,256 number of nodes and also compared the messages exchanged for same network.

Value Fault: Here the faulty node sends some garbage message instead of sending original message. Each node uses a buffer to store the previously received values from all other nodes in buffer. When a node receives a message, it compares the message received earlier. And then accepts the message if it is correct.Each store the time required to know about status of other nodes whether they are faulty. Our aim is to minimize the latency. We have taken Poisson probability distribution function to create random number of faulty nodes.

3.3.2 Basic Steps Followed:

Generation of Random network

We have taken an adjacent matrix whose all elements are 1 s except the diagonal. Diagonal element has value as -1.We have considered k-connected network. Here we have taken k as 3. K-connected network is defined as a network consisting of nodes and each node is connected to at least 3 nodes. Then for each node we have generated a random number which represents number of node that is connected to the earlier node directly. Then we generate the nodes that are not connected directly and make the value as 0.

It uses a function getNetwork(int number_of_node). This function generates random network and save it in a file named as “nod<number_of_node>.rtf”.

Example of network file

	0	1	2	3	4
0	-1	1	1	1	0
1	1	-1	1	0	1
2	1	1	-1	1	1
3	1	0	1	-1	1
4	0	1	1	1	-1

Table 3.4 :Adjacency Matrix

It can be noted that elements of adjacency matrix is diagonally symmetric. Since algorithms have to be applied to same set of network files, we have to store it in file. Latter on when required file is read and a 2 dimensional array is created from it.

Implementation of different algorithms

1. Flooding

In this algorithm server sends to adjacent node except from the source from which it comes. For this algorithm a large number of messages are exchanged. When server sends a request from client, it checks the adjacency matrix element from corresponding row.

```
for(i=0;i<no_of_nod;i++)
        if(adjMat[req_nod][i]==1)
```

Adjacent elements found by making such comparisons. Server writes to corresponding object output stream except for source node.

```
[Pseudo code for flooding algorithm]

for each node in network
     find adjacent node
     send to node
end for.
```

2. Variant of Flooding

This algorithm uses a common array “chkval[]” to store the information about sent nodes. Server program maintains a common array to check if node has already get message. If message is found then corresponding chkval is set to “1”. “createSeq” is a module which maintains the array.

It uses “getSeq(int req_nod,int no_of_nod)” to get corresponding 1 D array representing sequence in which message has to be sent . Here “req_nod” represents the requested node number and “no_of_nod” represents the total number of nodes in the network.

```
Procedure getSeq(int req_nod,int no_of_nod)
  read adjacency matrix store it in adjMat[][][]
  initialize chkval[] by setting all elemnt to “0”
  initialize seq[] with -10
  chkval[req_nod]=1 ;  // indicate source node always get from itself
  for each element “I” in adjacency matrix
       if ( adjMat[req_nod][i]==1)
          seq[count]=i;
          chkval[i]=1;
       end if
  end  for
  while (!Fillup())
       repeat
       for each element “i” in adjacency matrix
          if ( adjMat[req_nod][i]==1) && (chkval[i]==0)
               seq[count]=i;
               chkval[i]=1;
```

```
            end if
          end for
    end while
end Procedure

Procedure Fillup()
      for each element i in chkval
            if(chkval[i]==1)
                  temp+=chkval[i];
            end if
      end for
      if(temp==no_of_node)
            return true;
      else
            return false;
end procedure
```

3. **BFS Tree**

This algorithm creates a BFS tree for every node and sends message to node along the tree. “createTree” is a module which create a tree for every node. It uses a function “getTree(int req_nod,int no_of_nod)” which return a 2-D array representing a tree. Here tree is represented as 2-d array. here 1st row contains the immediate adjacent nodes.

Subsequently it stores the net child node in next row

Each time server receives request from client it constructs a tree and send to all nodes which are child nodes of requested client.

```
[Pseudo code of creating tree]
procedure createTree((int req_nod,int no_of_nod)
initialize tree[][] by setting all element to -10
initialize chkval[] by setting all elemnt to 0
for(i=0;i<no_of_nod;i++) // Assing adjacent node as level 0 hop
{
      if(adjMat[req_nod][i]==1)
      {
            ret[nod_index][adj_count]=i;
```

```
            chkval[i]=1;
            adj_count++;
      }
}
ret[nod_index][adj_count]=999; // 999 indicate END OF Imediate Node In
Same hop or same level
while(!Fillup(no_of_nod)) // while all nodes are found from source
{
  for(i=0;i<no_of_nod;i++) // check next immediate node
      {
            imd_nod=tree[nod_index][i];
            if( (imd_nod!=999) && (imd_nod!=-10) )
            {
                  t_nod_indx++;
                  tree[t_nod_indx]=getImdNode(imd_nod,no_of_nod); //
store next level or hop node to next level of "ret[]" array
            }
      }
      nod_index++; // continue untill all nod found by incrementing
level
}
tree[t_nod_indx+1][0]=998; // END of ALL levels found by 998

return tree;
end procedure;
procedure getImdNode(int nd,int no_of_nod)
      for(i=0;i<no_of_nod;i++)
      {
            if( (adjMat[nd][i]==1) && (chkval[i]==0) ) // check imediate
      by adjacency matrix's 1 value and "chkval[]"
            {
                  imdnd [count]=i;
                  chkval[i]=1;
                  count++;
            }
      }
      ret[count]=999;//indicate end os node at same level

      return imdnd;
End procedure
```

```
Procedure Fillup()
      for each element i in chkval
            if(chkval[i]==1)
                  temp+=chkval[i];
            end if
      end for
      if(temp==no_of_node)
            return true;
      else
            return false;
end procedure
```

3.3.3 EXAMPLES SHOWING IMPLEMENTATION OF ALGORITHMS

We have taken a small k-connected network (k=3). This network consists of 9 nodes.

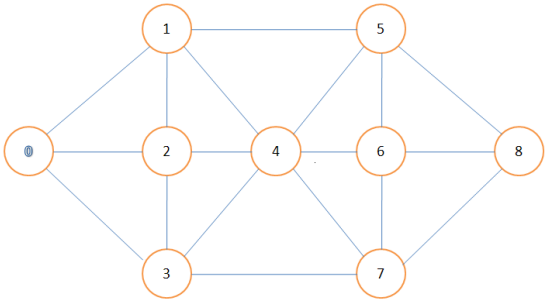

Figure 3.5 : k-conncted network

Flooding:

In this method each node sends to adjacent node. Adjacent node can be found from adjacency matrix. Each node sends to all adjacent nodes even if node got the message earlier.This method is simple but lots of message exchange will take place which may cause network congestion and bandwidth is wasted by sending such redundant messages

Variant of Flooding:

In this method each node maintains an array to check if it has send message to particular node or earlier or not. Size of table is = total number of node -1. The array is initialized with 0. Let's say node 7 is sending message to all node. After node 7 has sent it simply makes element of array as "1" for corresponding index.

0	1	2	3	4	5	6	8
0	0	0	1	1	0	1	1

Table 3.5 : Status table for node 7

Next time when other node receives the same message they compare with the array. Suppose node 4 receives the message & sent it to its adjacent node. Node 6 has adjacent nodes (1, 2, 3, 5, 6, and 7). As node 7 is source node and nodes 6 ,8 have already get the same message, node 4 only send to nodes(1,2,5). After node 4 send array content will be.

0	1	2	3	4	5	6	8
0	1	1	1	1	1	1	1

Table 3.6 : Status table for node 7 after broadcasting of node 4

This process continues until all element of array becomes "1".

BFS Tree Method:

Each node creates BFS tree taking root node as source. For example node 7

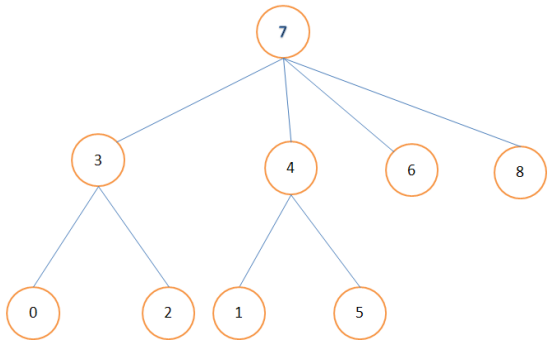

Figure 3.6 : Tree Taking root node as 7

BFS Tree creation:

First level of tree consists of immediate adjacent nodes of source which can be found from adjacency matrix. Entire tree is stored in a 2-dimenasional array. Elements of array are initialized to -10. First row is filled up by immediate adjacent node. End of adjacent node is indicated by "999".

Until all nodes are present in tree, it scans each element of row. Let's say "node 3".Node 3 has adjacent node as 0,2,4,7. As node 7 is source node it is discarded. Also node 4 is present in first row so t is discarded. Then remaining nodes (0, 2) become child node of 3.Similarly for node 4 child node will be node1 and node5.Nodes 6 and 8 has no child node. After this step all node become part of tree. So first element of next row stores 998 to indicate end of tree. 2-Dimensional array representing the tree.

index	0	1	2	3	4	5	6	7	8
0	3	4	6	8	999	-10	-10	-10	-10
1	0	2	999	-10	-10	-10	-10	-10	-10
2	1	5	999	-10	-10	-10	-10	-10	-10
3	999	-10	-10	-10	-10	-10	-10	-10	-10
4	999	-10	-10	-10	-10	-10	-10	-10	-10
5	998	-10	-10	-10	-10	-10	-10	-10	-10
6	-10	-10	-10	-10	-10	-10	-10	-10	-10
7	-10	-10	-10	-10	-10	-10	-10	-10	-10
8	-10	-10	-10	-10	-10	-10	-10	-10	-10

Table 3.7 : Array representaion of tree taking node 7 as root

Each node sends message to its adjacent node i.e. element at first row of tree array. When a node received any message from any node, it checks the tree for the corresponding source node and sends to only child nodes, thus minimizing total number of message exchanges.

CHAPTER 4

SIMULATION RESULTS

4.1 Simulation Results

Because creating large scale networks is expensive and time-consuming, network routing algorithms are evaluated by simulation. The implementation of flooding, variant of flooding and BFS is done using Java. We have taken each node as different port addresses. Each node runs in different thread. In real time situation when fault occurs each system either fails for each fault or for value fault it sends garbage messages. But as we have simulated it, the thread running as node either suspends or send message as garbage messages. We have simulated the 3 algorithms namely flooding, variant of flooding and BSF for 8, 16, 32, 64, 128, and 256 nodes. We have taken message preparation time as 2 milliseconds. Each node sends message at every 60 millisecond. Waiting time for each node is defined as maximum latency for previous transmission.

Then we have measured the maximum latency, average latency & number of message exchanges. Latency is measured in milliseconds. X-Axis represents number of nodes. Y-Axis represents latency (in millisecond).

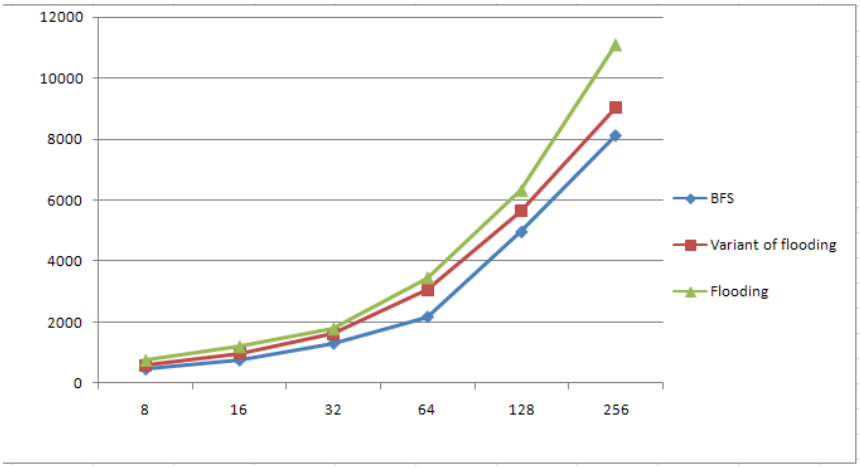

Figure 4.1: Maximum Latency V/s Number of nodes

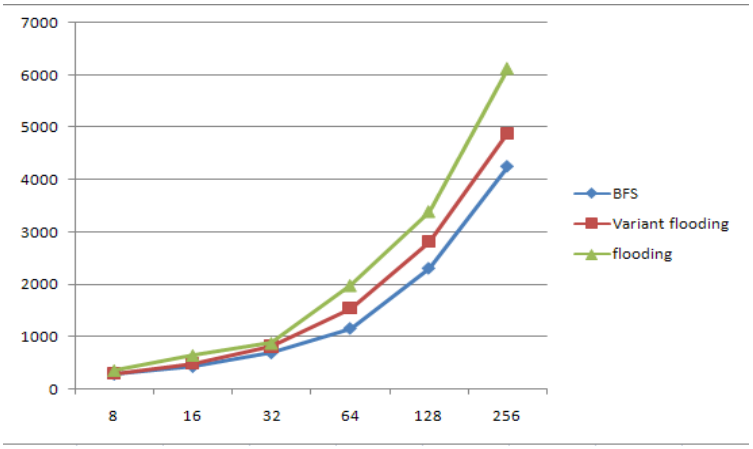

Figure 4.2: Average Latency V/s Number of nodes

We have measured the number of message exchanged between the nodes. The following figure represents the graph between number of node and number of message exchanged. X-Axis represents number of nodes. Y- Axis represents number of message exchanged. Number of message exchanged is taken as average of five instances of message exchanges.

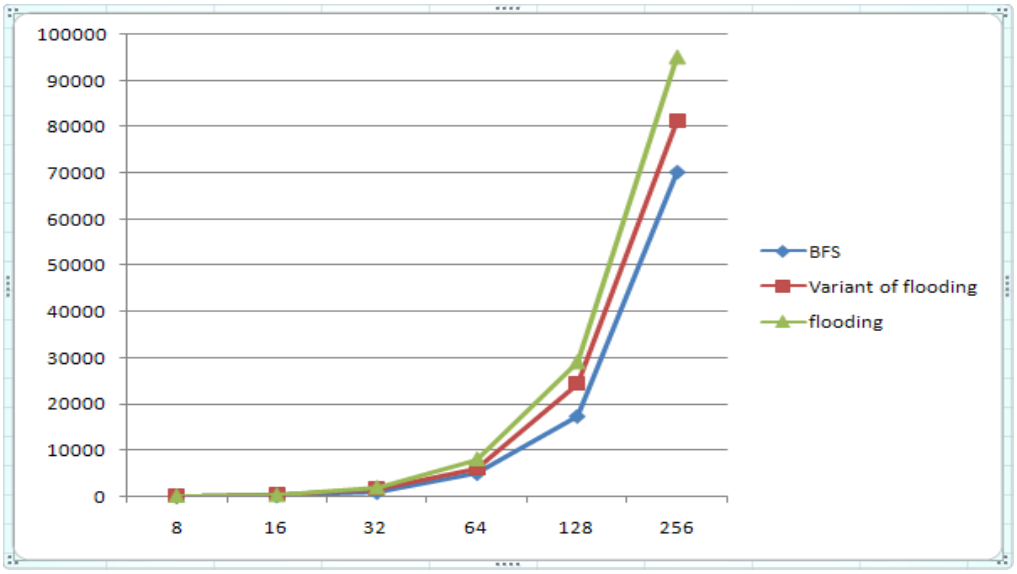

Figure 4.3: Number of Message Exchanged V/s Number of nodes

4.2 Observation:

From figure 4.1 it can be observed that as number of node increases maximum latency increases. BFS has lowest maximum latency among all the 3 algorithms. For lower number of nodes, all the 3 algorithms have approximately same latency. As number of node increases BFS performs better than all algorithms. Variant of flooding algorithm performs better than general flooding algorithm because it send message to a selected number of node. But flooding algorithm send to all neighboring node causing network congestion, loss of bandwidth. BFS creates tree for each node taking as root. So it can able to transmit message in correct direction. Similarly figure 4.2 shows that BFS performs better than all algorithms.

From figure 4.3 it can be observed that, number of message exchanged increases rapidly as number of node in network increases. Flooding algorithm send maximum number of messages, variant of flooding algorithm performs better than flooding by sending less number of messages than earlier. As BFS creates a tree for each node, it sends least number of messages.

CHAPTER 5

CONCLUSION

CONCLUSION

In our implementation all the nodes propagate their status information as quickly as possible to other nodes to allow it to effectively handle dynamic situations. All nodes can change state at the same time or a cascade of status changes can occur, while maintaining a notion of correctness at all times [3]. Our algorithms have been able to handle these behaviors effectively in not-completely- connected networks. The algorithms assumes the value faults in the nodes (i.e. either sending correct or erroneous message) to incorporate three different algorithms such as BFS Routing, Flooding, a variant of flooding out of which BFS results in shorter diagnostic latency and state holding time compared to other algorithms. With increasing complexity, the algorithms for supporting a wider range of faults such as partially working nodes and arbitrarily faulty nodes in dynamic fault environment for arbitrary network topologies can be developed. These algorithms can not be explicitly compared with previous algorithms as the definition of a testing round differs from algorithm to algorithm and due to the inability of previous algorithms to handle faults during fault recovery.

The next step of our work is to develop algorithm that provides coverage to a larger set of faults such as additional link failure between nodes. So on a progressive basis the final algorithm will achieve the capability of covering a complete range of faults from crash fault to faults of an unrestricted nature. Our simulation results are restricted up to 256 numbers of nodes due to processing requirement constraints. So on a high performance computer system our algorithm can be extended to simulate results for even more number of nodes.

REFERENCES:

[1] Arun Subbiah, and Douglas M. Blough, Distributed Diagnosis in Dynamic Fault Environments", IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS, VOL. 15, NO. 5, MAY 2004

[2] E.P. Duarte Jr., A. Brawerman, and L.C.P. Albini, "An Algorithm for Distributed Hierarchical Diagnosis ofDynamic Fault and Repair Events", Proc. IEEE ICPADS'OO, 2000.

[3] P.M.Khilar, and S.Mahapatra, "Distributed Diagnosis in Dynamic Fault Environment For Not-Completely Connected Network" , IEEE Indicon 2006 Conference, New Delhi, India, I I - 1 3 Dec. 2006

[4] Pabitra Mohan Khillar and Sudipta Mahapatra "Distributed Diagnosis in Dynamic Fault Environments for Arbitrary Network Topologies" , IEEE Indicon 2005 Conference, Chennai, India, I I - 1 3 Dec. 2005

[5] S. Rangarajan, A.T. Dahbura, and E. Ziegler, "A Distributed System-Level Diagnosis Algorithm for Arbitrary Network Topologies, "IEEE Trans. Computers, vol.44,pp.312-334, Feb. 1995.

[6] S. Even, Graph Algorithms, Computer Science Press, 1979.

[7] R. Farivar, S. G. Department of Computer Engineering, University of Technology Azadi Tehran, Iran." Directed Flooding: A Fault-Tolerant Routing Protocol " , Preceding of the 2005 Systems Communications ICW 05

[8] http://www.infosecwriters.com/text_resources/pdf/fault_tolerant.pdf

[9] http://ecow.engr.wisc.edu/cgi-bin/getbig/cse/bfs_fault.pdf

[10] http://sun.java.com/docs/network/sockets.jsp