Misbehavior Detection in Message Forwarding Networks

A dissertation submitted in partial fulfillment of the requirements for the completion of degree of the Master of Technology in
Computer Science

By
Shivram Latpate
(2005-2007)
Indian Statistical Institute
Kolkata.

Under The Supervision of

Dr. Aditya Bagchi
Computer and Statistical Services Center

INDIAN STATISTICAL INSTITUTE
203, Barrackpore Trunk Road
Calcutta – 700 035

Certificate

This is to certify that the dissertation entitled “Misbehavior Detection in Message Forward Networks” submitted by Shivram Latpate towards partial fulfillment for the degree of M.Tech in Computer Science at Indian Statistical Institute, Kolkata, embodies the work done under my supervision.

Dated: 23 July 2007

Signed:

(Dr. Aditya Bagchi)

Supervisor

External Expert

Acknowledgement

It is my privilege and pleasure to convey my gratitude to my guide **Dr. Aditya Bagchi** for his guidance, support and encouragement throughout the course of the dissertation without which it would have been impossible to complete this research endeavor. I express my gratitude to **Dr. Pianakpani Pal**, ECSU for his constructive criticism and suggestions throughout the year.

Shivram Latpate
(mtc0514)

Abstract

Social networks rely on the cooperation of the nodes participating in the network to forward packets to each other. However, it may be advantageous for individual nodes not to cooperate to save its resources, while still using the network to relay its own traffic. If too many nodes exhibit this misbehavior, network performance degrades and cooperating nodes may find themselves unfairly loaded. We propose a centralize protocol for social networks, making misbehavior unattractive for nodes. It aims at detecting and isolating misbehaving nodes, thus making it unattractive to deny co-operation.

Simulation experiments show that CONTROL node which controls the entire network, successfully detects misbehaving nodes by using the proposed protocol.

Key Words: Social networks, Misbehavior, Trust, Reputation, Success rate, Packets

Abstract

Social networks rely on the cooperation of the nodes participating in the network to forward packets to each other. However, it may be advantageous for individual nodes not to cooperate to save its resources, while still using the network to relay its own traffic. If too many nodes exhibit this misbehavior, network performance degrades and cooperating nodes may find themselves unfairly loaded. We propose a centralize protocol for social networks, making misbehavior unattractive for nodes. It aims at detecting and isolating misbehaving nodes, thus making it unattractive to deny co-operation.

Simulation experiments show that CONTROL node which controls the entire network, successfully detects misbehaving nodes by using the proposed protocol.

Introduction

1.Social Networks

A Social Network is the graph representation of a social community [1]. Here each member of a social community (people or other entities embedded in a social context) is considered as a node and communication (collaboration,interaction etc.) from one member of the community to another member is represented by a directed edge, forming a directed graph or digraph. A social network represents a network of acquaintances between people like, a club and its members, a city or village community, a research group communicating over Internet, a group of people communicating

with each other through e-mail messages for aspecific purpose etc. Recently, Web has played a major role in the formation of communities (cyber-communities) where the members are people from different parts of the globe who join the community for some common interest. The number of communities in the Web is increasing dramatically with time. Social networks represented as webgraphs follow IP protocol.

The Internet has been a vessel to expand social networks in many ways. Social networking services(SNSs) are one successful example of such a role. SNSs provide an online private space for individuals and tools for interacting with other people in the Internet. SNSs help people find others having common interest, establish a forum for discussion, exchange photos and
personal news and indulge many other cooperating activities [11].

This community formation is one of the most powerful socializing aspects of the Web and hence Web is also a social network. Out of the many social network models on the Web, most commonly used one is called a referral system. In such a system, each node in the social network provides a set of links to its acquaintances that in turn become member Nodes of the network. In the same way, these new nodes bring their acquaintances to the network again. This way the social network keeps on growing. So, the social network on the Web gives rise to an evolutionary graph. There are many commercial products like, LinkedIn.com, Ryze.com, Tribe.net[13][14][15] etc. that tend to view the web as a social network. These commercial products provide different types of services like, employment service, matrimonial service, e-learning service etc.

A graph representing social network may contain cycles and nested cycles. Cycles can be fused into a single node called as hyper node [1]. This fusion process converts original graph into a DAG [1].

This social network is maintained on a server or number of servers. This central control provides unique id to the member nodes. It also uses some node numbering scheme to assign numbers to the nodes for storing reachability information and also it assigns to each node its adjacency list. Adjacency list of a node contains list of those nodes which are directly reachable from that node. Each node also has its reachability information and routing table. This dissertation has used the terms network and social network synonymously.

2. Model and Problem Formulation

2.1 Motivation

Definitions:

Misbehavior: Misbehavior can be defined as non-cooperative behavior of a node which prevents some other node from getting proper service. There can be different types of misbehavior. One important misbehavior is to mislead other nodes. This dissertation has studied this misleading behavior.

Misleading: In this case a node may receive a packet but may not forward to next node, or it may receive but may not acknowledge after receiving, misleading other nodes. This may be interpreted as unsuccessful transfer of traffic through the network. It will also cause unnecessary retransmission of same packet again and again resucing the throughput of the network

considerably.

In a social network a node may be interested in sending some information or advertisement to a few eligible nodes. All such eligible destination nodes may not be directly reachable from a designated Source node. In that case source node can take help of intermediate nodes to forward the message to destination. By using routing table andReachability information Source node sends a packet to the intended node through shortest path. The protocol ensures that the intermediate nodes along the path forward the packet on behalf of the Source node. This message forwarding continues till the destination node is reached.

If a message is received by the destination node correctly, it sends an acknowledgment to the source. In this way, messages are sent to all destinations.

If an acknowledgment is not received from a destination then the source node times out and reports to the central authority.

There may be possibility of misbehavior in that path by a node. The present work is the outcome of an effort to model a protocol for misbehavior detection and isolating misbehaving nodes from the social network after getting enough evidences against it.

2.2 Related Works

Lot of research work has been done to detect misbehavior in Ad-hoc networks.

An ad hoc network is a group of wireless mobile computers (or nodes), in which

nodes cooperate by forwarding packets for each other to allow them to communicate beyond direct wireless transmission range.

Ad hoc networks require no centralized administration or fixed network infrastructure such as base stations or access points, and can be quickly and inexpensively set up as needed. They can be used in scenarios in which no infrastructure exists, or in which the existing infrastructure does not meet application requirements for reasons such as security or cost.

2.2.1 CONFIDANT PROTOCOL

S. Buchegger and Jean-Yves LeBoudec[MobiHOC-02] proposed CONFIDANT protocol for ad-hoc network. It detects misleading nodes by means of observation and more aggressively informs other nodes of this misbehavior through reports sent around the network.

CONFIDANT consists of four components. The Components are present is each node.

1. The Monitor

It is equivalent to neighborhood watch. A node locally looks for the deviating nodes. The nodes of the neighborhood watch can detect deviations by the next node on the source route by either listening to the transmission of the next node or by observing route protocol behavior. As soon as a given bad behavior occurs, the reputation system is called.

2. The Trust Manager

This component deals with incoming and outgoing ALARM messages. The source

of an ALARM has to be checked for trustworthiness before triggering a reaction. Thus there is a filtering of incoming ALARM messages according to the trust level of the reporting node. The trust manager consists of the following components.

a. An alarm table containing information about received alarms.

b. A trust table managing trust levels for nodes to determine the trustworthiness of an alarm.

c. A friends list containing all friends a node potentially sends alarms to.

3.The Reputation System

The reputation system in this protocol manages a table consisting of entries for nodes and their rating. The rating is changed only when there is sufficient evidence of malicious behavior that is significant for a node and that has occurred a number of times exceeding a threshold to rule out coincidences.

4.The Path Manager

The path manager performs the following functions:

a. Path re-ranking according to security metric, e.g. reputation of the nodes in the path.

b. Deletion of paths containing malicious nodes.

c. Action on receiving a request for a route from a malicious node.

d. Action on receiving request for a route containing a malicious node in the source route.

2.2.2 OCEAN PROTOCOL

Sorav Bansal and Mary Baker [4] proposed OCEAN protocol. OCEAN is a layer that resides between the network and MAC layers of the protocol stack, and it helps nodes make intelligent routing and forwarding decisions in ad-hoc networks. It avoids trust-management machinery and uses direct first-hand observations of other nodes behavior. OCEAN consist of five components

1.Neighbor-Watch

This module observes the behavior of the neighbors of a node. It relies on the omni directional nature of the antenna and assumes symmetric bi-directional links. In particular, it tracks misleading routing misbehavior. When forwarding a packet, the module buffers the packet checksum, and then monitors the wireless channel after sending the packet to its neighbor. If it does not hear the neighbor attempt to forward the packet within a pre-specified period (default 1ms), Neighbor-Watch registers a negative event against the neighbor node and removes the checksum from its buffer. On the other hand, over hearing a forwarding attempt by the neighbor, Neighbor-Watch compares the packet to the buffered checksum, and if it matches, it registers a positive event and removes the checksum from its buffer. If the checksum does not match, it treats the packet as not having

been forwarded. These events are communicated to the RouteRanker, which maintains ratings of the neighbor nodes.

2.RouteRanker

Every node maintains ratings for each of its neighboring nodes. The rating is initialized to Neutral and is incremented and decremented on receiving positive and negative events respectively from the Neighbor-Watch component.

Neutral Rating 0
Positive Step +1
Negative Step -2
Faulty Threshold -40

3.Rank-Based Routing

The Rank-Based Routing module applies the information from Neighbor-Watch in
the actual selection of routes. To make it possible to avoid routes containing nodes in the faulty list, a variable-length field is added to the DSR Route-Request Packet(RREQ) called the avoid-list. The avoid list is a list of nodes that the RREQ transmitter wishes to avoid in its future routes. On re-broadcasting an RREQ, a node appends its faulty list to the avoid list of the RREQ packet. Any node receiving an RREQ checks the RREQ avoid list. Depending upon the avoid list and the RREQ-route, a node decides whether to suppress the RREQ, or honor the RREQ by either re-broadcasting it or replying with a DSR Route-Reply.

4. Malicious Traffic Rejection

This module performs the straight-forward function of rejecting traffic from nodes it considers misleading. It employ the policy of rejecting all traffic from a misleading node so that a node is not able to relay its own traffic under the guise of forwarding it on somebody elses behalf.

5. Second Chance Mechanism

The Second Chance Mechanism is intended to allow nodes previously considered misleading to become useful again. A timeout based approach where a misleading node is removed from the faulty list after a fixed period of observed inactivity. Even though the node is removed from the faulty list, its rating is not increased to neutral, so that it can quickly be added back in the event of continued misbehavior. This timeout value is called the Faulty Timeout.

2.3 Problem Formulation

In my work I have tried to give a protocol for misbehavior detection and imposing punishment to misbehaving nodes in a social network.

Definition: In a social network represented by Directed Acyclic Graph (DAG), each node is represented as a member and directed edges represent relations or communication of one node to another node. While sending packets from a node to an intended destination node, there may be misbehaving intermediate node. To make this misbehavior unattractive, the proposed centralized misbehavior handling protocol detects and punishes misbehaving nodes.

Problem Statement: Given a social network represented by Directed Acyclic Graph (DAG) with N nodes and a CONTROL node that controls

entire social network, to design a strategy for misbehavior detection and action upon its confirmation.

CONTENTS

1. Introduction ……..6

2. Model and Problem Formulation …… 8

3.1 Motivation

3.2 Related Works

3.3 Problem Definition

3. Misbehavior Detection Protocol ……..14

4. Implementation Details ……..…18

4.1 Generating Random DAG

4.2 Assigning Reachability Information

4.3 Simulation Steps

4.4 Experiment Results

5. Algorithms ……34

7. Conclusion and Future Works……..48

8. References……..49

9. Programme Code----------60

Chapter 1

Introduction

1. Social Networks

A Social Network is the graph representation of a social community [1]. Here each member of a social community (people or other entities embedded in a social context) is considered as a node and communication (collaboration, interaction etc.) from one member of the community to another member is represented by a directed edge, forming a directed graph or digraph. A social network represents a network of acquaintances between people like, a club and its members, a city or village community, a research group communicating over Internet, a group of people communicating with each other through e-mail messages for a specific purpose etc. Recently, Web has played a major role in the formation of communities (cyber-communities) where the members are people from different parts of the globe who join the community for some common interest. The number of communities in the Web is increasing dramatically with time. Social networks represented as web graphs follow IP protocol. The Internet has been a vessel to expand social networks in many ways. Social networking services (SNSs) are one successful example of such a role. SNSs provide an online private space for individuals and tools for interacting with other people in the Internet. SNSs help people find others having common interest, establish a forum for discussion, exchange photos personal news and indulge many other cooperating activities.This community formation is one of the

most powerful socializing aspects of the Web and hence Web is also a social network. Out of the many social network models on the Web, most commonly used one is called a referral system. In such a system, each node in the social network provides a set of links to its acquaintances that in turn become member Nodes of the network. In the same way, these new nodes bring their acquaintances to the network social network on the Web gives rise to an evolutionary graph. There are many commercial products like, LinkedIn.com, Ryze.com, Tribe.net [10][11][12] etc. that tend to view the web as a social network. These commercial products provide different types of services like, employment service, matrimonial service, e-learning service etc.

A graph representing social network may contain cycles and nested cycles. Cycles can be fused into a single node called as hyper node [1]. This fusion process converts original graph into a DAG [1].

This social network is maintained on a server or number of servers. This central control provides unique id to the member nodes. It also uses some node numbering scheme to assign numbers to the nodes for storing reachability information and also it assigns to each node its adjacency list. Adjacency list of a node contains list of those nodes which are directly reachable from that node. Each node also has its reachability information and routing table. This dissertation has used the terms network and social network synonymously.

Chapter Two

Model and Problem Formulation

2.1 Motivation

Definitions:

Misbehavior: Misbehavior can be defined as non-cooperative behavior of a node which prevents some other node from getting proper service. There can be different types of misbehavior. One important misbehavior is to mislead other nodes. This dissertation has studied this misleading behavior.

Misleading: In this case a node may receive a packet but may not forward to next node, or it may receive but may not acknowledge after receiving, misleading other nodes. This may be interpreted as unsuccessful transfer of traffic through the network. It will also cause unnecessary retransmission of same packet again and again rescuing the throughput of the network considerably.

In a social network a node may be interested in sending some information or advertisement to a few eligible nodes. All such eligible destination nodes may not be directly reachable from a designated Source node. In that case source node can take help of intermediate nodes to forward the message to destination. By using routing table and Reachability information Source node sends a packet to the intended node through shortest path. The protocol ensures that the intermediate nodes along the path forward the packet on behalf of the Source node. This message forwarding continues till the destination node is reached. If a message is received by the destination node correctly, it sends an acknowledgment to the source. In this way, messages are sent to all destinations. If an acknowledgment is not received from a destination then the

source node times out and reports to the central authority. There may be possibility of misbehavior in that path by a node. The present work is the outcome of an effort to model a protocol for misbehavior detection and isolating misbehaving nodes from the social network after getting enough evidences against it.

2.2 Related Works

Work has been done to detect misbehavior in Ad-hoc networks. An ad hoc network is a group of wireless mobile computers (or nodes), in which nodes cooperate by forwarding packets for each other to allow them to communicate beyond direct wireless transmission range.

Ad hoc networks require no centralized administration or fixed network infrastructure such as base stations or access points, and can be quickly and inexpensively set up as needed. They can be used in scenarios in which no infrastructure exists, or in which the existing infrastructure does not meet application requirements for reasons such as security or cost.

2.2.1 CONFIDANT PROTOCOL

S. Buchegger and Jean-Yves Le Boudec[MobiHOC-02] proposed CONFIDANT protocol for ad-hoc network. It detects misleading nodes by means of observation and more aggressively informs other nodes of this misbehavior through reports sent around the network.

CONFIDANT consists of four components. The Component s are present is each node.

1. The Monitor

It is equivalent to neighborhood watch. A node locally looks for the deviating nodes. The nodes of the neighborhood watch can detect deviations by the next node on the source route by either listening to the transmission of the next node or by observing route protocol behavior. As soon as a given bad behavior occurs, the reputation system is called.

2. The Trust Manager

This component deals with incoming and outgoing ALARM messages. The source of an ALARM has to be checked for trustworthiness before triggering a reaction, thus there is a filtering of incoming ALARM messages according to the trust level of the reporting node. The trust manager consists of the following components.

a. An alarm table containing information about received alarms.

b. A trust table managing trust levels for nodes to determine the trustworthiness of an alarm.

c. A friends list containing all friends a node potentially sends alarms to.

3. The Reputation System

The reputation system in this protocol manages a table consisting of entries for nodes and their rating. The rating is changed only when there is sufficient evidence of malicious behavior that is significant for a node and that has occurred a number of times exceeding a threshold to rule out coincidences.

4. The Path Manager

The path manager performs the following functions:

a. Path re-ranking according to security metric, e.g. reputation of the nodes in the path.

b. Deletion of paths containing malicious nodes.

c. Action on receiving a request for a route from a malicious node.

d Action on receiving request for a route containing a malicious node in the source route.

2.2.2 OCEAN PROTOCOL

Sorav Bansal and Mary Baker [4] proposed OCEAN protocol. OCEAN is a layer that resides between the network and MAC layers of the protocol stack, and it helps nodes make intelligent routing and forwarding decisions in ad-hoc networks.It avoids trust-management machinery and uses direct first-hand observations of other nodes' behavior. OCEAN consist of five components

1. NeighborWatch

This module observes the behavior of the neighbors of a node. It relies on the omni directional nature of the antenna and assumes symmetric bi-directional links. In particular, it tracks misleading routing misbehavior. When forwarding a packet, the module buffers the packet checksum, and then monitors the wireless channel after sending the packet to its neighbor. If it does not hear the neighbor attempt to forward the packet within a timeout (default 1ms), NeighborWatch registers a negative event against the neighbor node and removes the checksum from its buffer. On the other hand, on over hearing a forwarding attempt by the neighbor, Neighbor Watch compares the packet to the buffered checksum, and if it matches, it registers a positive event and removes the checksum from its buffer. If the checksum does not match, it treats the packet as not having been forwarded. These events are communicated to the RouteRanker, which maintains ratings of the neighbor nodes.

2. RouteRanker

Every node maintains ratings for each of its neighboring nodes. The rating is initialized to Neutral and is incremented and decremented on receiving positive and negative events respectively from the NeighborWatch component.

3. Rank-Based Routing

The Rank-Based Routing module applies the information from NeighborWatch in the actual selection of routes. To make it possible to avoid routes containing nodes in the faulty list, we add a variable-length field to the DSR Route-Request Packet (RREQ) called the avoid-list. The avoid list is a list of nodes that the RREQ transmitter wishes to avoid in its future routes. On re-broadcasting an RREQ, a node appends its faulty list to the avoid list of the RREQ packet. Any node receiving an RREQ checks the RREQ avoid list. Depending upon the avoid list and the RREQ-route, a node decides whether to suppress the RREQ, or honor the RREQ by either re-broadcasting it or replying with a DSR Route-Reply.

4. Malicious Traffic Rejection

This module performs the straight-forward function of rejecting traffic from nodes it considers misleading. It employ the policy of rejecting all traffic from a misleading node so that a node is not able to relay its own traffic under the guise of forwarding it on somebody else's behalf.

5. Second Chance Mechanism

The Second Chance Mechanism is intended to allow nodes previously considered misleading to become useful again. A timeout based approach where a misleading node is removed from the faulty list after a fixed period of observed inactivity. Even though the node is removed from the faulty list, its rating is not increased to neutral, so that it can quickly be added back in the event of continued misbehavior. This timeout value is called the Faulty Timeout.

2.3 Problem Formulation

In my work I have tried to give a protocol for misbehavior detection and imposing punishment to misbehaving nodes in a social network.

Definition: In a social network represented by Directed Acyclic Graph (DAG), each node is represented as a member and directed edges represent relations or communication of one node to another node. While sending packets from a node to an intended destination node, there may be misbehaving intermediate node. To make this misbehavior unattractive, the proposed centralized misbehavior handling protocol detects and punishes misbehaving nodes.

Problem Statement: Given a social network represented by Directed Acyclic Graph (DAG) with N nodes and a CONTROL node that controls entire social network, to design a strategy for misbehavior detection and action upon its confirmation.

Chapter Three

Misbehavior Detection Protocol

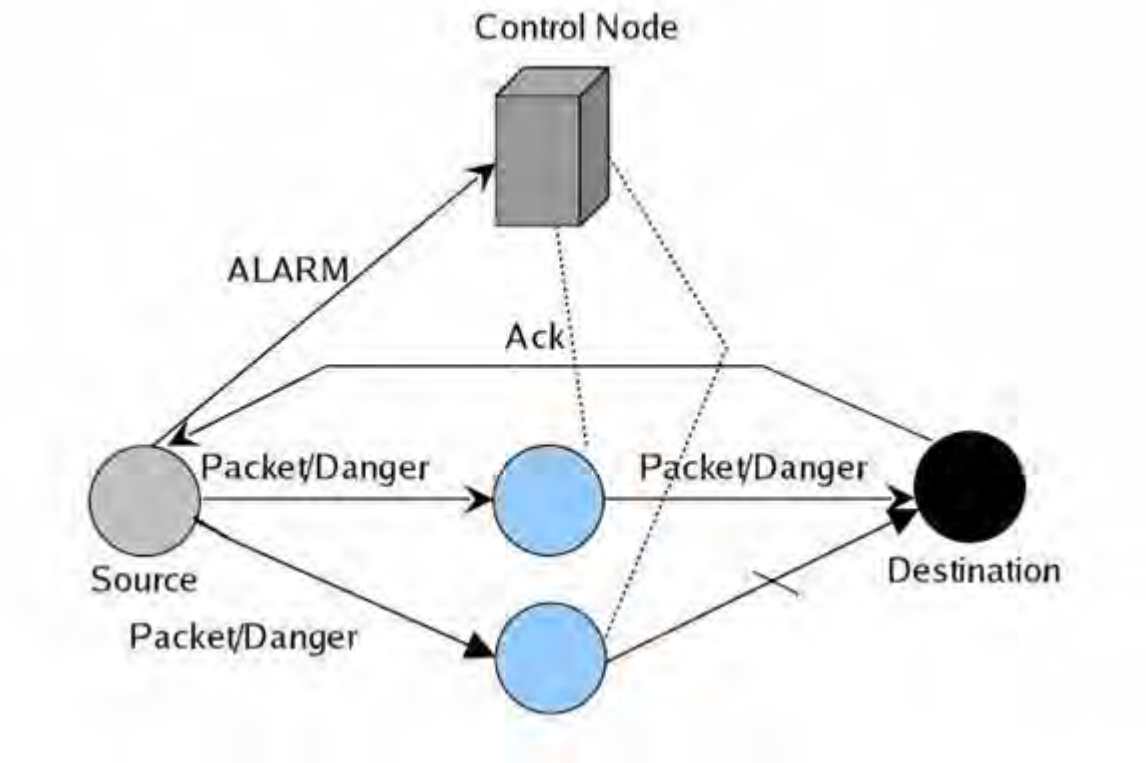


Fig 3.1 Misbehavior Detection System

3.1 PROTOCOL DISCRIPTION

There is a Control Node (CONTROL) that controls entire social network.

1. Each node has its adjacency list, Reachability information and routing table.
2. A node sends a packet to an intended destination through the shortest path.
3. In case of time out source sends an ALARM to CONTROL and SENDs DANGER SIGNAL to destination node of failed path to detect the Node which is failed to give acknowledgement.
5. CONTROL initiates VARIFICATION Process to detect misbehaving node.

3.2 DANGER SIGNAL

1. In case of DANGER SIGNAL a node sends a packet in a route to destination and expects ACK from each node on the way.
2. Before Destination D a ACK received from a Node X and not from Next Node y in a the route then problem is between node X and Y and CONTROL is informed accordingly.

3.3 VERIFICATION PROCESS

1. CONTROL asks the all source that can reach X and Y to send VERIFICATION Packet to these two nodes.
2. CONTROL routes the packets sent and ACK received for th4 both X and Y and determines SR (Success Rate) for each of them . as,

SR = (No. of ACK received) / (Packet sent).

SR is bounded between 0 and 1.

A. If SR is below a threshold then for either X or Y then Fault Count
FC is increases by 1.
B. If SR is bellow the threshold for both X and Y then Fault Count of Both X and Y is increased by 1.
C. If SR is above threshold then for both X and Y then either X or Y might have misbehaved or the concerned source node has made a deliberate false reporting. So CONTROL keeps X, Y and the concern source on ALERT. If the same situation occurs again then FC of involved nodes are increased by 1.

3.4 REPUTATION SYSTEM

1. Initially all the nodes are considered to have reputation value RV as **1**.
2. Each time a node receives and acknowledges a packet successfully its success Count is increased by 1. Its Reputation Value RV is updated as...

RV = RV + α *(FC / (FC + SC)).

3. Each time fault count FC of a node is increased by 1 then its Reputation Value RV is updated as...

RV = RV - α *(FC / (FC + SC)).

If the RV of the node is below threshold then it is put on the faulty List and removed from the network. Routing table for the affecting Nodes are updated accordingly.

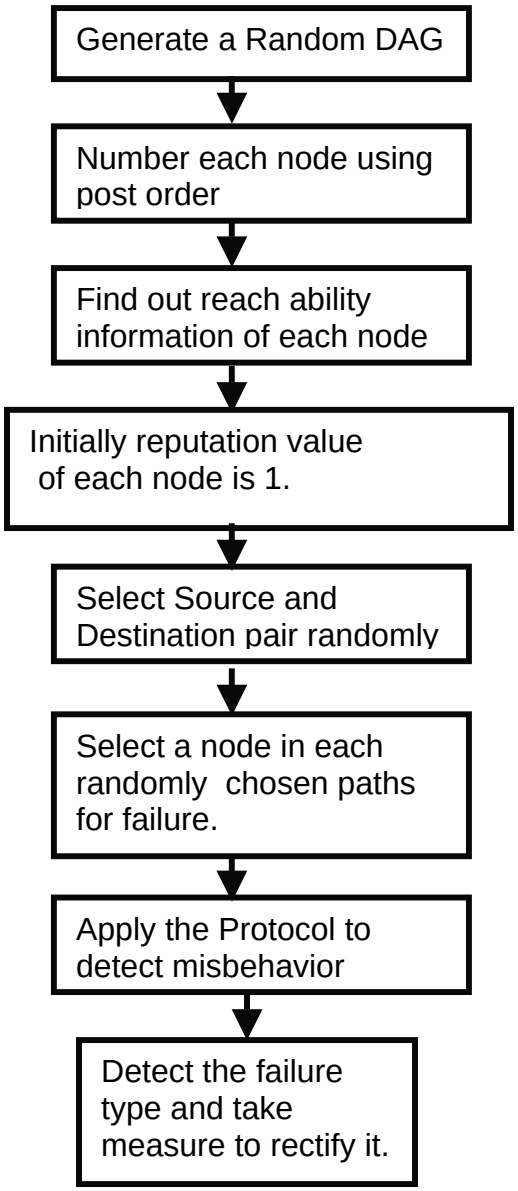
For misbehaving node higher value of α take to the faulty list faster even without giving it time to improve its RV, hence suitable Value is needed to chosen in order to take fair detection of Misbehavior.

Chapter Four
Implementation Details

Algorithm Steps:

1. Generate a DAG randomly.
2. Number each node using post order numbering.
3. Find out reachability information about each node.
4. Assign same Reputation value to each node, assuming all are faithful initially with value 1.
5. Select a source randomly.
6. Select destination reachable from the source, randomly.
7. Send a message from source to destination through shortest path.
8. Select a path randomly and select a misbehaving node randomly from that path.
9. Apply the Misbehavior detection protocol to detect failures in the network.
10. Take the measures according to failure type.
11. Repeat the steps 5 to 10 and study the success rate of packet transmission, Danger signal and verification overhead.

Flow Chart of the Algorithm



4.1 Generating Random DAG

In this chapter we generate a random DAG by fixing minimum percentage of source nodes and minimum percentage of sink nodes. Algorithm to generate random DAG is as follows

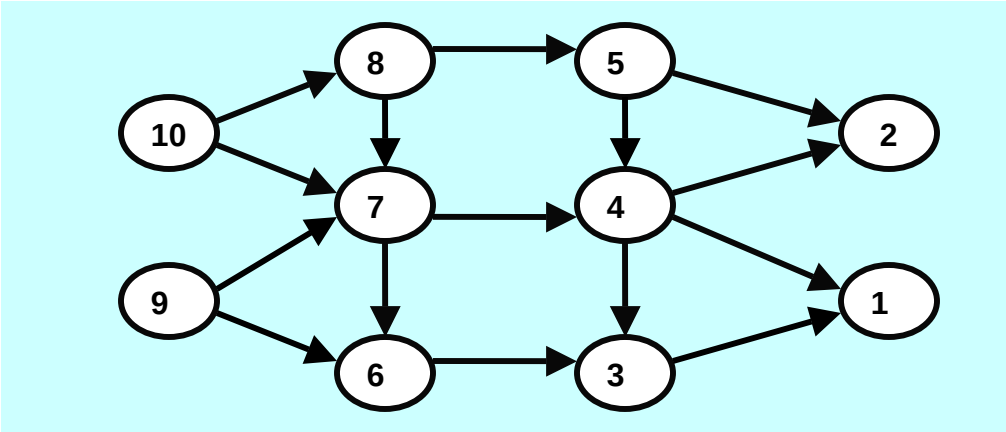


Fig 4.1

Node	Edge Information	Out Degree
1	0	0
2	0 0	0
3	1 0 0	1
4	1 1 1 0	3
5	0 1 0 1 0	2
6	0 0 1 0 0 0	1
7	0 0 0 1 0 1 0	2
8	0 0 0 0 1 0 1 0	2
9	0 0 0 0 0 1 1 0 0	2
10	0 0 0 0 0 0 1 1 0 0	2
In Degree	2 2 2 2 1 2 3 1 0 0	

Fig 4.2

4.2 Finding Reachability Information

In this chapter we compute the Reachability information for each node. The labeling scheme proposed by H V Jagadish [2] is adapted.

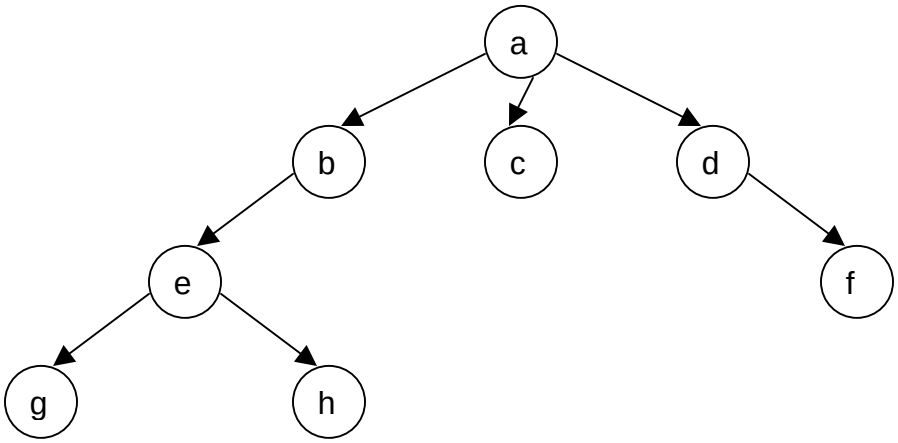
Definitions:

Successor List: The successor list of a node is the list of all nodes that are reachable from the specified node by traversing arcs of the graph, and immediate predecessor list is the list of all nodes to which the specified node has a direct arc.

Predecessor list: the predecessor list of a node is the list of all nodes that can reach this node by traversing arcs of the graph, and the *immediate predecessor* list of a node is the list of all nodes that have a direct arc to the specified node.

Post order Number: Number each node to reflect its relative position in a post order traversal of the tree. Let us call this number a *post-order number*.

Example:



Mapping Table for the DAG is as shown in Fig 4.3

Actual Node Value	a	b	c	d	e	f	g	h
Post-Order Number	8	4	5	7	3	6	1	2

Now, to each node in the tree assign an index consisting of the lowest post-order number among its descendents. For simplicity, we assume that every node can reach itself, so that the index associated with a leaf node is the same as the post-order number of the node.

From now onwards, all the processing is done with reference to this post order numbering of the nodes.

By generalizing the numbering scheme that was adopted for directed trees to the case of Directed Acyclic Graphs and assuming that the graph consists of only one connected component; note that disjoint components can be hooked together by creating a virtual root node [4]

4.2.1 Preprocessing

The compression scheme works as follows:

1. Find a spanning tree T for the given graph G. We call T the tree-cover of G.

2. Assign post-order numbers and indices to the nodes of T as discussed earlier in case of directed trees. Thus, at the end of this step, an interval [i, j] is associated with each node, such that j is the post-order number of the node and I is the lowest post-order number among its descendents.

3. Examine all the nodes of G in the reverse topological order. At each node p, do the following processing:

    - For every arc (p, q), add all the intervals associated with the node q to the intervals associated with the node p.

    - At the time of adding an interval to the interval set associated with a node, if one interval is subsumed by

another, discard the subsumed interval. That is, if two intervals $[i_1, i_2]$ and $[j_1, j_2]$ are such that $i_1 \leq j_1$ and $i_2 \geq j_2$, then discard $[j_1, j_2]$. If two intervals $[i_1, i_2]$ and $[j_1, j_2]$ are such that $j_1 = i_2 + 1$, then create one $[i_1, j_2]$ corresponding to these two intervals. Merge two intervals, $[i_1, i_2]$ and $[j_1, j_2]$ into $[i_1, j_2]$, if $i_1 \leq j_1 \leq i_2 \leq j_2$.

Example:

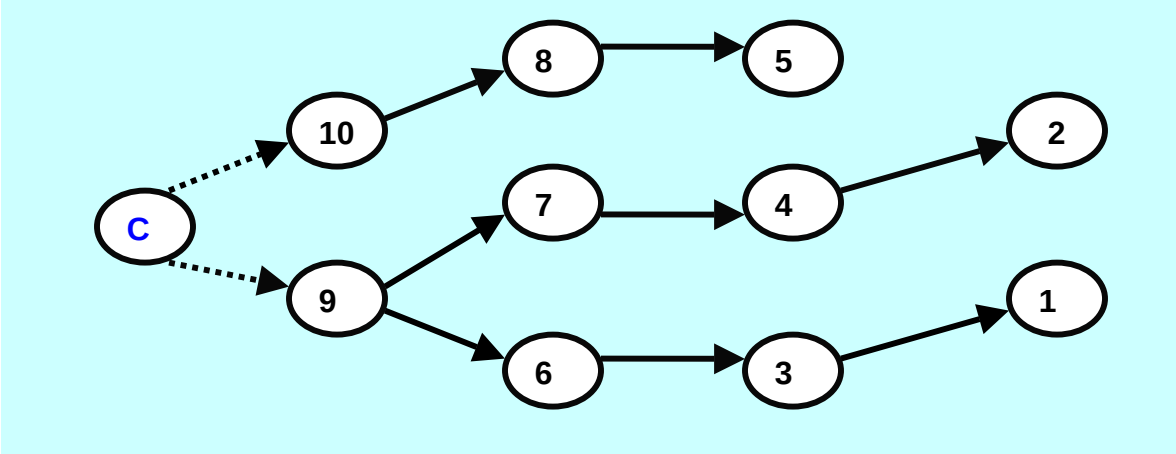


Fig. 4.4 Tree Cover of the DAG G with Original Numbering scheme

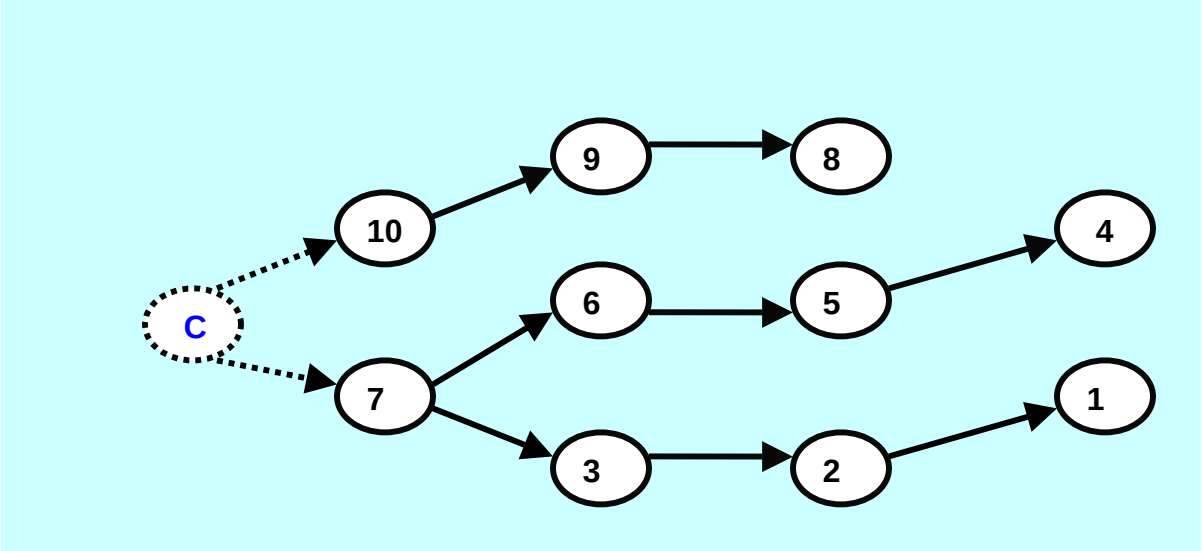
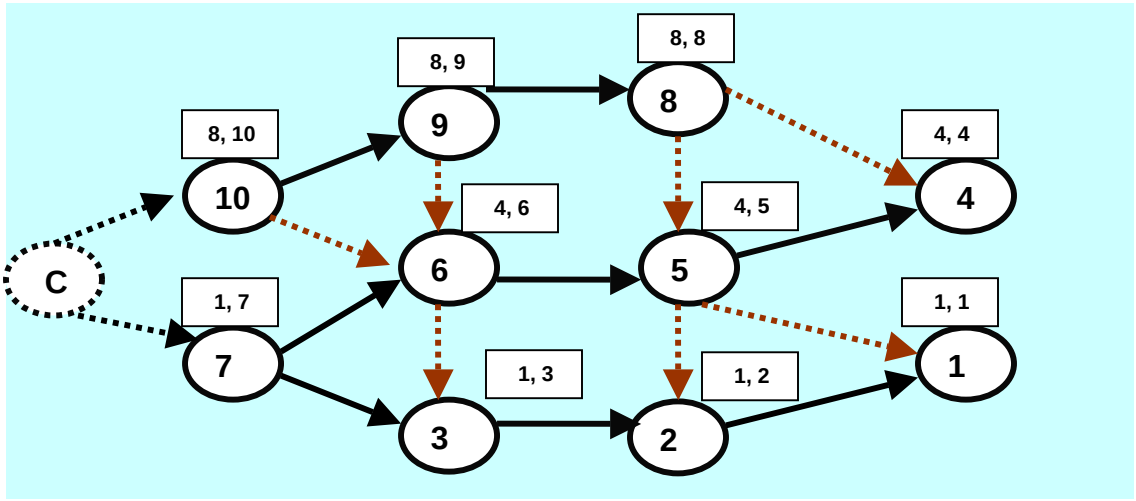


Fig 4.5 post order numbering assigned to Nodes

Initial Node Number	1	2	3	4	5	6	7	8	9	10
Post Order Node Numbering	1	4	2	5	8	3	6	9	7	10

Assigning Reachability Information to each node



Unique range intervals assigned to each node

Node	Reachable Range Values
1	[1,1]
2	[1,2]
3	[1,3]
4	[4,4]
5	[4,5][1,2]
6	[1,6]
7	[1,7]
8	[8,8][4,5][1,2]
9	[8,9][1,6]
10	[1,6][8,10]

Fig.4.6 each node is assigned with unique node interval.

4.3 Simulation Steps

1. Randomly pick up one Source and one destinations node where the destination is reachable from the concerned source.

2. Send Packet through the best routes.

3. Randomly consider an intermediate node to fail in randomly failed path for follow the above procedure.

4. Repeat steps 1 and 3 for different source destination pair gradually increase the probability of failure and study the DANGER SIGNAL and overhead of verification and success rate of packet sending. Here overhead is measured in terms of number of packets sent.

Experiment 1

Title: Studying Success Rate of packet delivery by gradually increasing probability of Node failure keeping packet rate constant.

Experimental Details:

Number of Nodes in the Network: 2000
Number of Source Nodes: 200
Number of Sink Nodes: 400
Number of Packets sent each time: 2000

Success Rate (SR) = (SPD)/ (NP)

SPD = No of Packets successfully delivered to destination
NP = Total Number of Packets sent (Here NP = 2000)

Table 8

Probability of Failure	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40
No of Packets Delivered successfully	1517	1384	1287	1197	1115	1030	916	854
Total Number of packets	2000	2000	2000	2000	2000	2000	2000	2000
Success Rate(SR)	0.7585	0.6920	0.6436	0.5985	0.5575	0.5150	0.4580	0.4270

Graphical Interpretation of results:

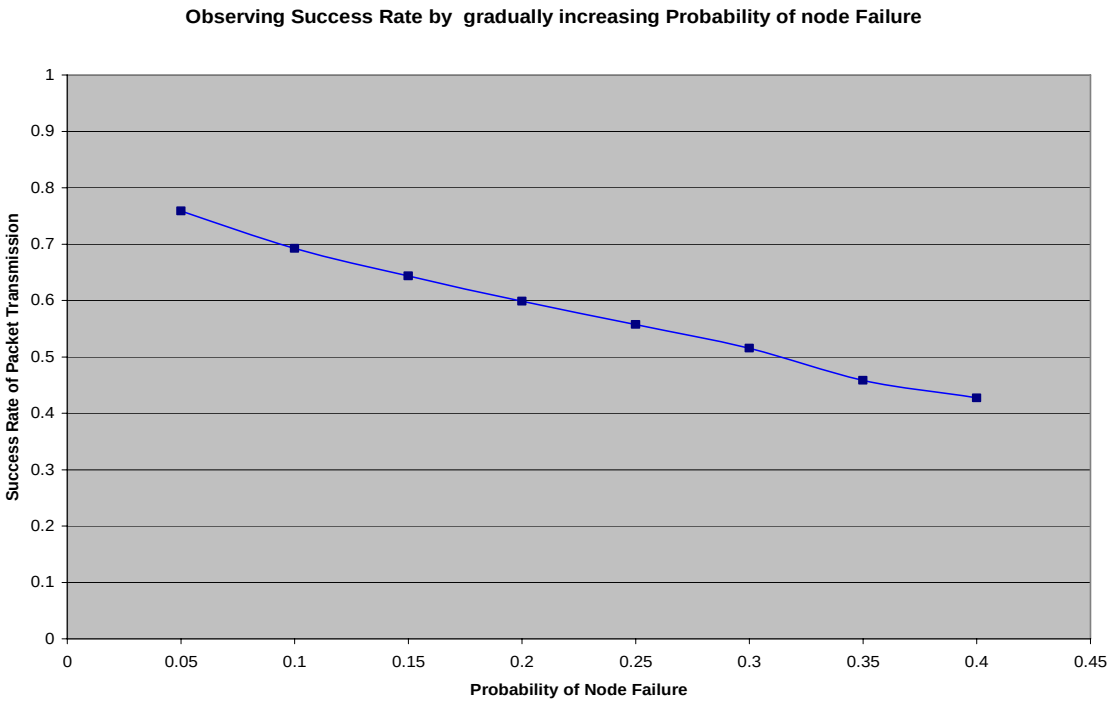


Fig. Study of Probability of node failure Vs Success Rate

From the above results it is clear that with gradual increase in the Probability of node failure success rate (SR) of packet transmission decreases.

Experiment 2

Experiment Description:

We fix the following parameters throughout the experiment. We vary number of failed node in the network and study the efficiency of the protocol along with Overhead of verification w. r. t. increase in failure rate of the nodes. Here overhead is measured in terms of number of packets sent in verification process.

Table 1

Fixed Parameters

Total number of nodes	1000
Number of Source Nodes	200
Number of Sink nodes	200
No. Of destinations selected to send packets	100
Constant α	0.20
Threshold	0.40

Results Obtained:

Table 2

Number of node failed randomly	5
Overhead of verification process	796 packets
Number of nodes isolated from network(Put in Faulty List)	3
Number of nodes put on Alert list	2
Number of nodes escaped from both list	0

Table 3

Number of node failed randomly	10
Overhead of verification process	1194 packets
Number of nodes isolated from network	6
Number of nodes put on Alert list	2
Number of nodes escaped from both list	2

Table 4

Number of node failed randomly	15
Overhead of verification process	2581 packets
Number of nodes isolated from network(Put in Faulty List)	12
Number of nodes put on Alert list	2
Number of nodes escaped from both list	1

Table 5

Number of node failed randomly	20
Overhead of verification process	4172 packets
Number of nodes isolated from network(Put in Faulty List)	12
Number of nodes put on Alert list	8
Number of nodes escaped from both list	0

Table 6

Number of node failed randomly	25
Overhead of verification process	4975 packets
Number of nodes isolated from network (Put in Faulty List)	15
Number of nodes put on Alert list	7
Number of nodes escaped from both list	3

Table 7

Number of node failed randomly	30
Overhead of verification process	5529 packets
Number of nodes isolated from network(Put in Faulty List)	23
Number of nodes put on Alert list	4
Number of nodes escaped from both list	3

Graphical plotting of Results

1. Number of Failure nodes Vs Verification Overhead

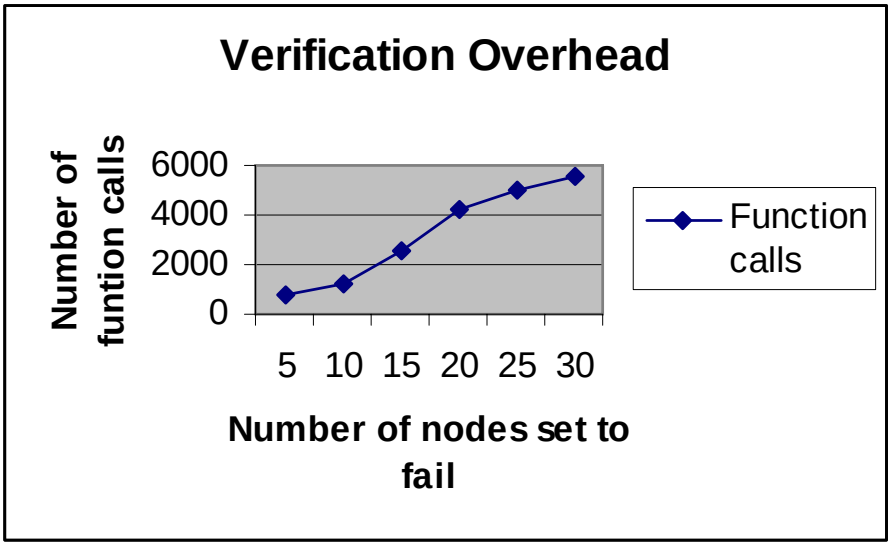


Fig 4.7

2. Number of randomly failed nodes Vs No of Nodes isolated from the Network among failed nodes**.**

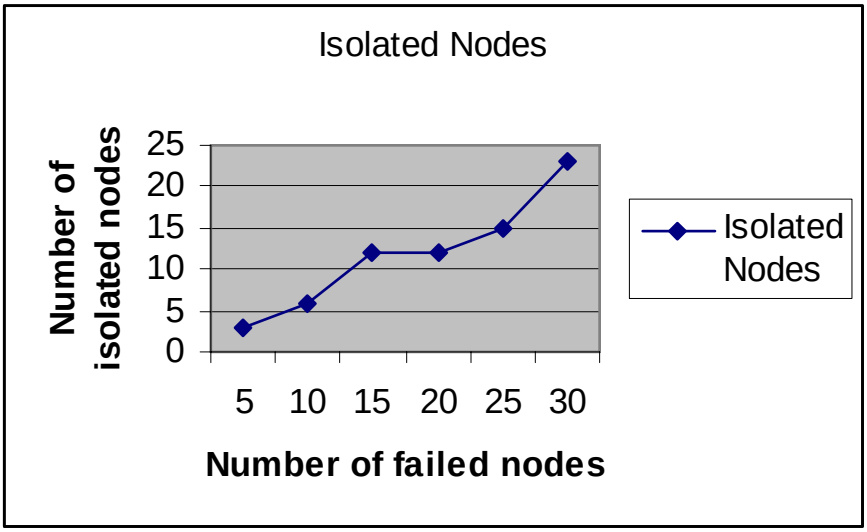


Fig 4.8

3. Number of randomly failed nodes Vs No of Nodes put on Alert List among failed nodes

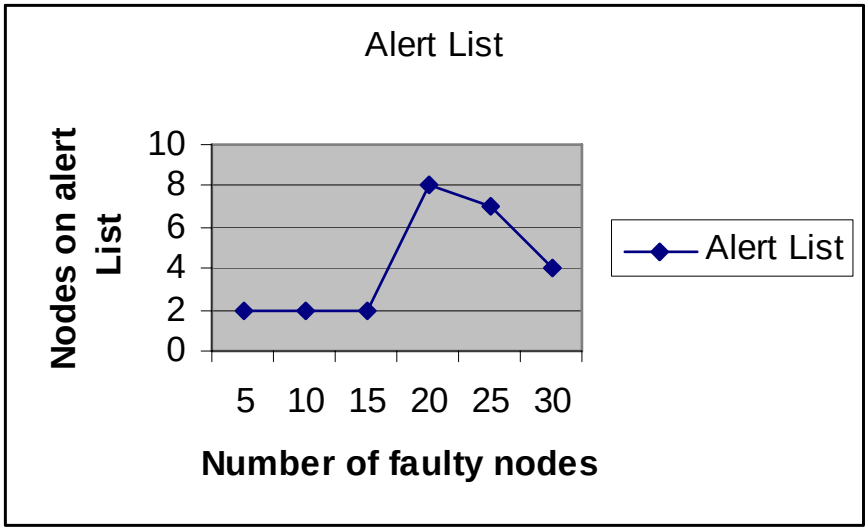


Fig 4.9

4. Combined Graph

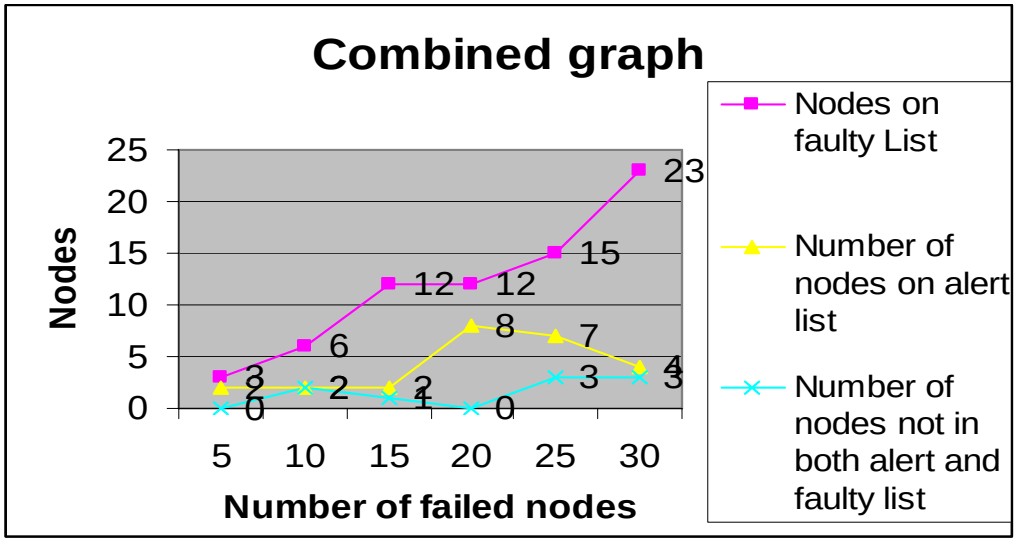


Fig 4.10

Chapter Five

ALGORITHMS

Algorithm 1 Generating a random DAG

Algorithm: **generateRandomDAG**(N,mSrcP,mSnkP)

Input: N: Number of nodes in DAG G.
mSrcP: minimum percentage of source node
mSnkP: Minimum percentage of sink nodes

Output: a random DAG G of N nodes where nodes are numbered using
Reverse topological ordering
DAG is represented by a continuous binary string of zeros and ones.

```
Begin
    NoOfSrc = (mSrcP * N) /100 //Fixed minimum source nodes
    NoOfSnk = (mSnkP * N ) / 100 //fixed minimum sink nodes

    TotalBits = N*(N+1) /2    /*Total bits require to store the DAG*/
    TotalBytes = TotalBits /8 + 1

    Create an array adjM of Bytes whose size is equal to TotalBytes
    For (i = 1 to TotalBytes) do
        Generate a random number in the range [0,255],
        Let it be Rnum.
        adjM[i] =  Rnum
    End
    /*Set all diagonal elements to zero */
    For (i = 1 to N) do
```

```
            setBit(i,i,0,adjM)
          End
         /*Set all indegrees of source node to zero */
         For (i = N-1 to (N-NoOfSrc+1) ) do
               For (j = ((i+1) to N) do
                   setBit(i,j,0,adjM)
               End
        End

       /*Set all out degrees of all sink node to zeros*/
       For (i = 2 to NoOfSnk )
          For (j = 1 to i-1)
             setBit(i,j,1,adjM)
          End
       End
End
```

Algorithm 2 to add an edge from a node to other node

Algorithm: **set Bit** (row,col,bit,adjM)

Input: row: row number in DAG G represented by adjM
col: Column number in DAG G represented by adjM
bit: either 0 0r 1
adjMat: adjacency matrix of DAG G

Output: Set the value of given position in adjacency matrix to given bit value. In other words add or delete an edge between one node to other.

Begin

1. Find the starting position of the given row in the adjacency matrix adjM.
2. Add offset col to the starting position
3. Set the value at that position to as bit /* bit = 0 or 1*/

End

Algorithm 3 to retrieve an edge Information from a node to other

---

3. Algorithm: **getBit**(row,col,adjM)

Input:

row: row number in DAG G represented by adjM
col: Column number in DAG G represented by adjM
adjMat: adjacency matrix of DAG G

Output:
Return the value at the given position in adjacency matrix.
in other words check whether there is an edge from node row to node col.

Begin
1. Find the starting position of the given row in the adjacency matrix adjM.
2. Add offset col to the starting position
3. Read the value at that position /* value can be either 0 or 1*/
4. Return that value

End

Algorithm 4 Algorithm for assigning node Id to each node in the network

---

Algorithm: **assignId**(N,adjM)

Input: N: Number of nodes in DAG G represented by adjM
adjM: adjacency matrix representation of DAG G

Output: each node is assigned with Id which is equal to value of lowest post order number among all its descendents.

```
Begin
    Define an array Id of size equal to N
    For (i = 1 to N ) Do
        if (i is a sink node )   then
          Id[i] = i
        Else
          Find a node immediate successor j of i such that
          Id[j] is lowest among all immediate successor of i.
          Id[i] = Id[j]
        End
    End
End
```

Explanation: Start from lowest post order number to highest post order number. Here for every node which is not sink we need to examine only list of its immediate successors to get Id.

Algorithm 5 Algorithm for generating non tree edge cover

5. Algorithm: **genNonTreeEdgeCover**(N,adjM,Tcov)

Input: N: number of nodes in DAG G.
adjM: adjacency matrix representation of DAG G whose nodes are numbered using
post order number method .
Tcov: Tree Cover of DAG G.

Output: nTCov : non tree edge cover of DAG G.

```
Begin
    For i = 1 to N do
        For j = 1 to N-1 do
            p = getBit(i,j,AdjM)
            q=getBit(i,j,Tcov)
            bit = p XOR q.
            setBit(i,j,bit,nTcov)
        End
    End

End
```

Algorithm 6 Algorithm for assigning Reachability information to a node

6. Algorithm: **generateRangevalues**(N,IdM,adjM)

Input: N: Number of nodes in DAG G.
IdM: Matrix which stores Ids of node N
adjM: adjacency matrix representation of DAG G.

Output: Each node is assigned a initial range value [i,j] where j is its post order number of that node and i is Id assigned to that node.

```
Begin
    Initialize a adjacency list Rlist for each node.
    Initially Rlist is empty.
    For i = 1 to N do
        Add range (Id[i], i) in Rlist[i]
    End
End
```

Algorithm 7 Algorithm for inheriting node range intervals

Algorithm: **inheritNodeRange**(N,adjM,nTcov,Rlist)

Input: N: Number of nodes in DAG G.
adjM: Adjacency matrix representation of DAG G.
nTcov: Non tree edge cover of DAG G.
Rlist: Range list array. initially only one range is associated with each node.

Output: each node is associated with an interval or set of intervals that encapsulate reachability information for the nodes which are reachable by this node.

Begin

1. Examine all the nodes in ascending postorder numbers.
2. At each node p do the following processing

2.1 For every arc(p,q), add all the intervals associated with the node q to The interval associated with node p.

2.2 At the time of adding an interval to the interval set associated with node, If the two intervals [i,j], [k,l] are such that i <= k and l > j then discard [i,j].

If the two intervals [i,j],[k,l] such that k = j+1 then create on [i,l] Corresponding to the two intervals.

if two intervals [i,j],[k,l] such that i <= j <= k <= l then merge the two Intervals into [i,l].

End

Algorithm 8 Algorithm to generate Reachability matrix

Algorithm: **generateReachabilityMatrix**(N,adjM,Rlist)

Input:
N: Number of nodes in DAG G.
adjM: Adjacency matrix representation of DAG G.
Rlist: Range list of all the nodes of DAG G.

Output: Reachability matrix R such that
RMat(i,j) = 1, if node i is reachable from node j.
= 0, otherwise

```
Begin
    For (i = 1 to N) do
      For (each interval [p,q] associated with node i)  do
        For (k = p to q) do
            setBit(i,k,1,RMat)
        End
      End
    End
End
```

Algorithm 9 for assigning node Id to each node in the network

9. Algorithm: **isReachable**(N1,N2,RMat)

Input: N1, N2: Nodes of DAG G.
RMat: Reachability Matrix.

Output: If N1 is reachable to N2 then return 1else return 0.

```
Begin
        Reach = getBit (N1, N2, RMat)
        If (reach) then
          Return (1)
        Else
          Return (0)

End
```

Algorithm 10 Algorithm for selecting source and destination randomly

Algorithm: **selectRandomSrcDest(N, SrcA)**

Input: N: Number of Nodes
srcA: array of Source Nodes

OutPut : A randomly selected source node and r randomly selected destinations.
(r is generated randomly) stored in array Dest

Begin

1. Generate a random number between 1 to NoOfSources let is be Rn.
   randSrc = srcA[Rn];
   Initially array Dest is empty.
2. Generate a random number r between 1 to (N - NoOfSources);
   For i = 1 to r do
   Generate a random number between 1 to N let it be p.

   if (p is not already selected and not a source node then)
   Add p into Dest Array.
   End if
   End for

End

Algorithm 11 Algorithm for partitioning list of destinations into reachable and unreachable.

Algorithm: partitionReachUnReach(Rs, Dest)

Input: Rs: Randomly selected Source Node.
Dest: Randomly selected r Destinations.
RMat: Reachability Matrix of Graph G.

Output: Two lists Reachable and Unreachable First contains List of destination which are
Reachable from source and second contains list of unreachable Destinations.

Begin

Examine each node p in Dest.
If (isReachable (Rs,p,RMat)) then
Add p in Reachable Matrix.
Else
Add p in Unreachable Matrix.

End If.

End.

Algorithm 12 Algorithm for checking whether a node is reachable from other

---

12. Algorithm: **isReachable**(p,q,RMat)

```
Input:  p: Node which would be checked whether reachable to q or not.
        q : Node to check for reachability.
         RMat : reachabilbity Matrix calculated from reachability Information.

Output:  If P is reachable to q then return 1
          if p is not reachable to q then return 0

Begin
      reach = getBit(p,q,RMat)
      if (reach) then
        Return 1
      Else
        Return 0
     End if
End
```

Algorithm 13 Algorithm for sending message to all destinations

```
13. Algorithm: sendMessageToAllDest
(Rsrc,DestA,NoOfDest,RMat,AdjMat,RNf )

Input: Rsrc :- Randomly chosen source node to send messages.
        NoOfDest : Number of Destination nodes
        DestA : Destination Nodes array.
        RMat : reachability Matrix.
        AdjMat : Adjacency Matrix of Graph G.


Output: ackMat : an acknowlegment matrix which stores acknowledgments
                 received from Destinations.

Begin
        Select a random Destination for failure let it be fNode.
          fType= getRandomFType();

          /* If  type = 1 then its misbehavior */
          if (fType == misB)
             mType = getRndMisBehaviour()

          /*also get choose w r t whom it is misbehaving */

        mBwrt = getFailurewrt (src, neighboring Node)
         For (All reachable destination nodes d from Source Rsrc )  do
              ack = sendMessage(src, d, RMat, AdjMat,fNode);
              ackMat[d] = ack ;
```

```
        End For

    ONE: If (there is any unrachable node from source) then
            Select a source src node among unreachable destinations.

           For (each reachable destination d from src) do
                ack = sendMessage(src,d,RMat,AdjMat,fNode).
                ackMat[d] = ack
           End for

          If (Still there is any destination remained ) then
             goto ONE.
          end if
        end if
End
```

Algorithm 14 Algorithm sending message from a source to destinantion

14. Algorithm **sendMessage**(src, dest,RMat, AdjMat,fDest)

```
Input: src : Source node
       dest : Destination node
       AdjMat : Graph represented by an adjacency Matrix
       RMat : Reachability Matrix.
       fNode : randomly chosen destination to fail.

Output: It sends ack = 1 if message is successfully sent to destination else gives
        ack = 0 for failure.

Begin
      If dest != fdest then do the following
      While (1) do
            If dest is directly reachable from src then
               Return (1).
            Else
                 Find its immediate successor which is reachable to dest and
take a hop
                 To that node.
             End if
       End while

     If (dest is of randomly failed path) then
          /* select randomly node to fail only once */
          /* in later case this node will be refer a failed node*/
         /*we do not know path length as we take only one hop at a time*/
```

```
          /*so we follow some fair policy to select a node randomly*/

          rFailure = testRandomFailure();

          if (rFailure ) then      /*Current node is selected as failed node */
              fNode = currentNode;
             Return (0)
         Else
             /*If no node is selected randomly as failure node then sink node get
failed*/
               if (current node is sink ) then
                  fNode = currentNode.
                   return(0)
         end
      end
end
```

Algorithm 15 Algorithm for selection failure type of node randomly

. Algorithm: **getRandomFType** ()

Input: None

Output: 1 = misbehavior
0 = Physical failure

```
Begin
          Generate a randomly 100 binary bits
          Counts number of one's in that
           If number of one's > 50 then
              Return (1) //Misbehavior
            Else
              Return (0) //Physical failure

End
```

Algorithm 16 Algorithm for selecting an intermediate node to fail.

Algorithm: **testRandomFailure** ()

Input: None

Output: 1: if a node in randomly failed path is selected as failed node
0: Otherwise

```
Begin
        Generate a random number N in the range of 1 to 100.
        Generate a binary string of length n randomly initialized with 0's and 1's.
        Generate two numbers n1 and n2 in the range of 1 to 100 and n1 < n2.

        if (No of 1's in the string are in the range (n1,n2] ) then
          Return (1)
        Else
          Return (0)
        End
End
```

Explanation: here while reaching to destination we do not know path length. A node can take a hop by using information associated with it. So this can be one of the policies to generate the randomness criteria.

Algorithm 17 Algorithm for deciding misbehavior randomly

17. Algorithm: **typeOfMisbehaviour** (Node)

Input: Misbehaving Node whose malicious act is to be deciding randomly.

Output: 1: if misbehaving nodes does not forward
0: if misbehaving node does not acknowledge.

```
Begin
        If (node is sink) then   /*A sink can not forward any way*/
           Return (0)

       Else
           Generate a number in the range 0 to 1000
           If (number is greater than 500) then
              Return (1)
           Else
               Return (0)
          End if
       End if
End
```

Algorithm 4 Algorithm for assigning node Id to each node in the network

19. Algorithm: **sendDangerSignal**(S, D, AdjM, RMat)

Input: S is source node
D is destination node
AdjM is graph G represented by Adjacency matrix
RMat is Reachability matrix

Output: When danger signal passed by a path, after receiving the signal each normally
Behaving node will acknowledge individually.
It will trace the node which does not acknowledge.

```
Begin

    ack = 1
    currentNode = S

    While ( ack ) do
        prevNode = currentNode
        /*Take a hop to nextnode */
        currentNode = nextHop(current, D, AdjM,RMat  )

        /*Collect an ack from Current Node */
        /*If Nodes gives ack then ack = 1 else ack = 0 */
```

ack = getAcknowledgement(currentNode)
End

Return (currentNode)

End

Explanation: If Danger signal would be passed through a path which fails. The node which fails to acknowledge would be traced. Also the Node which was last acknowledged may be tested for its behavior.

Algorithm 4 Algorithm for assigning node Id to each node in the network

---

20. Algorithm: **nextHop**(C,D,AdjM, RMat)

Input: C - current Node
D- Destination
AdjM - Adjacency Matrix
RMat - Reachability Matrix

Output:
Current node C takes a hope to ***nextNode*** in the path towards destination.
by using Adjacency and Reachability information attached with it.

Begin
Examine **AdjM** associated with **C** and find first eligible immediate Successor which is reachable to Destination let it be ***nextNode***.

Return (***nextNode***)

End

Conclusion and Future Work

Study of other misbehaviors is needed like message tempering, selfish behavior, means a node not co-operating but relying its own traffic in the networks.

As verification overhead is increasing with increase in no of misbehaving nodes, idea of distributed misbehavior detection protocol can be an issue of study.

Second chance mechanism can be implemented in which CONTROL can put back an isolated node in the network. In this case reputation value of the node will not be increased. Node has to increase its reputation by co-operating. This way a node may get second chance to prove it to be co operative**.**

Bibliography

[1] C.Bhanu Teja, S. Mitra , A Bagchi, and A. K. Bandyopadhyay, “Pre-processing and Path Normalization of a Web Graph used as a Social Network ”

[2] R. Agrawal , A. Borgida , H. V. Jagadish, “Efficient management of transitive relationships in large data and knowledge bases”, Proceedings of the 1989 ACM SIGMOD international conference on Management of data, p.253-262, June 1989, Portland, Oregon, United States262, June 1989, Portland, Oregon, United States.

[3] S. Buchegger and Jean-Yves Le Boudec. “Performance Analysis of the CONFIDANT Protocol; Cooperation of Nodes -Fairness in Dynamic Ad Hoc NeTworks”, In Proceedings of IEEE/ACM Symposium on Mobile Ad Hoc Networking and Computing (MobiHOC), Lausanne, CH, June 2002.

[4] Sorav Bansal, Mary Baker, “Observation-based Cooperation Enforcement in Ad hoc Networks”, Technical Report Computer Science Dept Stanford University.

[5] Jean-Yves Le Boudec and Slavisa Sarafijanovic “An Artificial Immune System Approach to Misbehavior Detection in Mobile Ad-Hoc Network” in Proceeding of

Bio-AIDT 2004.

[6] S Balchandran D DasGupta and L Wang "A Hybrid Approach for Misbehavior Detection in Wireless Ad-Hoc Networks"

[7] A. V. Aho, J. E. Hopcroft and J. D. Ullman, "Data Structure and Algorithms", Addison-Wesley, Reading, Mass.1983.

[8] Bhanu Teja C.(2005): Processing of path queries on Trees and DAGs.

M.Tech.(Comp.Sc) Dissertation Report, Indian Statistical Institute

[9] http://www.wikipedia.com

[10] http://www.LinkedIn.com

[11]. http://www.Ryze.com

[12]. http://www.Tribe.net

```
#include <stdlib.h>
#include<math.h>
#include<stdio.h>
#include<time.h>
#include<string.h>

#define USAGE "%s: total-dags percentage-source percentage-sink"
#define IN   0
#define OUT  1



 typedef unsigned char BYTE;

 unsigned char ucMisBehType;

 unsigned long ulFailSource,
               ulTypeOfFail,
               ulFailedNode,
                 ulRandFailedPath,
                 ulMisBWrt,
                 *ulpIsoNodeArray,
               ulPrevNode;



/*****************************************************************************/
/* This Function is to assign a Range to Node which gives Reachability info */
/*****************************************************************************/


 struct sRange {
                 unsigned long ulNodeNo;
                 unsigned long ulStart, ulEnd;
                 struct sRange *next;
               };

/*****************************************************************************/

   BYTE isSinkNode(unsigned long,BYTE *);

/*****************************************************************************/
/* This Function gives Sum of natural number                                 */
/*****************************************************************************/

 unsigned long sumNatNum(unsigned long);
```

```
/*****************************************************************************
**/
/* This Function is to assign a Range to Node which gives reachability
information          */
/*****************************************************************************
*/

 void transformPostOrder(unsigned long,  unsigned long * ,BYTE
*ucpInputAdMat,
                    BYTE *ucpPostOrderAdMat);

/*****************************************************************************
*/

 void genNonTreeCover(unsigned long,
                    BYTE *,
                    BYTE *,
                    BYTE *);

/*****************************************************************************
**/

 void genNodeInterval(unsigned long  ,unsigned long * , struct sRange * );

/*****************************************************************************
**/

 void genTreeCover(unsigned long ,
                unsigned long *,
                unsigned long *,
                BYTE *,
                unsigned long,
                unsigned long *) ;

/*****************************************************************************
***/


 void inheritNodeRange(unsigned long, BYTE * ,BYTE *,  struct sRange *,
unsigned long *);

/*****************************************************************************
**/

 int subSumed(unsigned long , unsigned long, unsigned long, unsigned long);
```

```
/*****************************************************************************************
*/

 void addNodeRange(   unsigned long ,
                      unsigned long,
                      unsigned long,
                      unsigned long,
                      struct sRange * );

/*****************************************************************************************
*/

 void delNextNode(unsigned long ,struct sRange *,struct sRange * );

/*****************************************************************************************
*/

 void delRedundantRange( unsigned long ,
                         unsigned long ,
                         unsigned long,
                         struct sRange *,
                         struct sRange * );

/*****************************************************************************************
***/

 void mergeTreeNonTreeEdge(unsigned long, BYTE *, BYTE *, BYTE *);

/*****************************************************************************************
***/

/*****************************************************************************************
**/
 void genReachabilityMat(struct sRange *, BYTE *,unsigned long );

/*****************************************************************************************
**/


/*****************************************************************************************
**/
 void selectRandomDest(unsigned long,
                      unsigned long,
                      unsigned long *,
                      BYTE *);
```

```
/*****************************************************************************
**/

/*****************************************************************************
**/
 void classifyReachable(unsigned long,
                        unsigned long,
                        unsigned long,
                        unsigned long *,
                        unsigned long *,
                        unsigned long *,
                        unsigned long*,
                        unsigned long *,
                        BYTE *);

/*****************************************************************************
**/

/*****************************************************************************
**/

 BYTE sendMessage(unsigned long,
                  unsigned long,
                  BYTE * ,
                  BYTE *);

/*****************************************************************************
**/

/*****************************************************************************
***/

                  BYTE getTypeOfMisBehaviour(unsigned long ,unsigned long ,BYTE
*);

/*****************************************************************************
***/

/*****************************************************************************
**/
                      BYTE getFailureType();
```

```
/*****************************************************************************/


/*****************************************************************************/
                BYTE isRandNodeFailure();
/*****************************************************************************/


/*****************************************************************************/
  void  forwardMsgToAllDest(unsigned long,
                unsigned long,
                unsigned long *,
                unsigned long,
                unsigned long,
                unsigned long*,
                unsigned long *,
                unsigned long*,
                BYTE *,
                BYTE *,
                unsigned long *);
/*****************************************************************************/

/*****************************************************************************/
int AllCovered(unsigned long * ,unsigned long );
/*****************************************************************************/

/*****************************************************************************/
 unsigned long sendDangerSignal(  unsigned long,
                                  unsigned long,
                                          unsigned long *,
                                          BYTE *,
                                          BYTE *
                                 );
/*****************************************************************************/

/*****************************************************************************/
    unsigned long  takeNextHop(unsigned long,BYTE *,BYTE *,unsigned long);

/*****************************************************************************
***/

/*****************************************************************************/

   BYTE getAcknowledgment(unsigned long,
                          unsigned long,
                                unsigned long,
```

```
                        unsigned long);

/*****************************************************************************/

/*****************************************************************************/
          void failureDetection(
                  unsigned long ,
                      unsigned long,
                  unsigned long,
                      unsigned long *,
                  unsigned long *,
                      unsigned long ,
                      unsigned long ,
                      BYTE * ,
                      BYTE *
                     );
/*****************************************************************************/

/*****************************************************************************/
    unsigned long  misBehaviorWrt();
/*****************************************************************************/



/*****************************************************************************/

void takeActMisBehWrtNbr(
                      unsigned long ,
                      unsigned long ,
                    unsigned long ,
                      unsigned long *,
                      BYTE * ,
                      BYTE *
                    ) ;



/*****************************************************************************/

void takeActMisBehWrtSrc(
                unsigned long ,
                unsigned long ,
                    unsigned long ,
                    unsigned long *,
```

```
                    BYTE *,
                    BYTE *);

/*****************************************************************************/


/*****************************************************************************/
   BYTE   checkMisWrtNbr(unsigned long ,
                         unsigned long ,
                             unsigned long ,
                                 unsigned long  ,
                                 unsigned long ,
                                   unsigned long ,
                                   unsigned long *,
                                   unsigned long *,
                                 BYTE *,
                                 BYTE *);



/*****************************************************************************************
***********/

   void reduceTrust(unsigned long ,
                    unsigned long *);

/*****************************************************************************************
***********/

BYTE checkPhysicalFailure(unsigned long ,
                    unsigned long ,
                    unsigned long ,
                        unsigned long ,
                          unsigned long ,
                        BYTE *,
                          BYTE *);



/*****************************************************************************/

    void takePhyFailAct(unsigned long ,
            unsigned long ,
            unsigned long ,
              unsigned long,
            BYTE *,
```

```
            BYTE *);

/*****************************************************************************/

BYTE checkMisBehWrtSrc(unsigned long ,
                       unsigned long ,
                       unsigned long ,
                            unsigned long ,
                            unsigned long ,
                            unsigned long *,
                            BYTE *,
                            BYTE *
                      );


/*************************************************************************/
     void isolateNode(unsigned long,unsigned long,BYTE *);
/*************************************************************************/

/*************************************************************************/
/*           FUNCTION DEFINATIONS                                        */
/*************************************************************************/



BYTE  getBitVal(unsigned long ucRow,unsigned long ucPos,BYTE *ucptAmat)
{
  /*****************************************************/
  /*   To Get the Bit in a row;BitPosition->01234567    */
  /*****************************************************/
  unsigned long int uiOffset,ucDiv,ucTmp  ;
  int i;

  BYTE ucRmndr,ucBitval;

  ucBitval=128;
  uiOffset=sumNatNum(ucRow-1);
  ucDiv=(uiOffset+ucPos)/8;
  ucRmndr=(uiOffset+ucPos)%8;

  if(ucRmndr){
     ucBitval = (ucBitval >> (ucRmndr-1));
     ucTmp=ucptAmat[ucDiv];
  }
```

```
  else{
    ucBitval=1;
    ucTmp=ucptAmat[ucDiv-1];

  }

  ucTmp= (ucTmp & ucBitval);

  if(ucTmp)
     return(1);
  else
     return(0);

}

/*****************************************************************/
/* This function set the bit Argmts: 1.Row no  2.position of bit  */
/* 3.value to set 0/1 and 4.adj matrix                            */
/*****************************************************************/

void setBitVal(unsigned long ucRow,unsigned long  ucPos,unsigned long  ucBit,
BYTE *ucptAmat)
{
  unsigned long int uiOffset,ucDiv,ucTmp;
  int i;
  BYTE ucRmndr,ucBitval;
  ucBitval=128;
  uiOffset=sumNatNum(ucRow-1);
  ucDiv=(uiOffset+ucPos)/8;
  ucRmndr=(uiOffset+ucPos)%8;

  if(ucRmndr)
    ucBitval = (ucBitval >> (ucRmndr-1));
  else
    ucBitval=1;

  if(ucBit ){
      switch(ucRmndr)
      {
        case 0:
          ucptAmat[ucDiv-1]=(ucptAmat[ucDiv-1]|ucBitval);
          break;
        default:
           ucptAmat[ucDiv]=(ucptAmat[ucDiv]|ucBitval);
       }
       /*printf("\n Cheking...change BYTE : %d \n ",ucptAmat[ucDiv]);*/
```

```
  }
  else
{
     switch(ucRmndr)
      {
        case 0:
             ucptAmat[ucDiv-1] = ucptAmat[ucDiv-1] & (~ucBitval);
          break;
        default:
           ucptAmat[ucDiv]= (ucptAmat[ucDiv] & (~ucBitval));
       }
      /* printf("\n Cheking...change BYTE : %d \n ",ucptAmat[ucDiv-1]);*/
}


}



BYTE setBit0(BYTE ucPosition)
{

/*  ********************** */
/*  bitPosition -> 01234567 */
/*  ********************** */
   BYTE ucByte = 1;
   ucByte = ucByte << (7-ucPosition);
   ucByte = ~ucByte;
   return ucByte;

}

BYTE setBit1(BYTE ucPosition)
{
/*  ********************** */
/*  bitPosition -> 01234567 */
/*  ********************** */
   BYTE ucByte = 1;
   ucByte = ucByte << (7-ucPosition);
   return ucByte;
}


BYTE getBit(BYTE ucByte, BYTE ucPosition)
{
/*  ********************** */
```

```
/*  bitPosition -> 01234567 */
/*  ********************** */
    ucByte = ucByte << ucPosition;
    ucByte = ucByte >> 7;
    return ucByte;
}


unsigned long sumNatNum(unsigned long ulNatNum)
{
    return ulNatNum * (ulNatNum + 1L) / 2L;
}

BYTE alreadySet(unsigned long *ulpNode,
                unsigned long ulNo,
                unsigned long ulTheNode)
{
    unsigned long ulCount;

    for (ulCount = 0; ulCount < ulNo; ulCount ++)
        if (ulpNode[ulCount] == ulTheNode)
            return 1;
    return 0;
}


void assignNodeId(unsigned long ulNodes,
                  BYTE * ucpPostAdMat,
                  unsigned long * ulpNodeId )
  {

      unsigned long ulTmpVal,
                    ulTmpId,
                    ulRow,
                    ulCol;

      for(ulRow = 1; ulRow <= ulNodes ; ulRow++  )
      {
          //if its sink Node then assign node number as node id
          if ( isSinkNode(ulRow,ucpPostAdMat) )
             ulpNodeId[ulRow - 1] = ulRow ;
          //If not sink Node then examine all its descendent and their ids ans take
smallest
          else
          {
              ulTmpId =  ulNodes + 10;  //initialize Some larger number
```

```
            for(ulCol = 1; ulCol < ulRow ; ulCol++)
            {
              if( ( getBitVal( ulRow,ulCol,ucpPostAdMat ) ) == 1 )
              {

                 ulTmpVal = ulpNodeId[ulCol - 1];
                 if(ulTmpVal < ulTmpId )
                   ulTmpId = ulTmpVal;
              }
            }
            ulpNodeId[ulRow - 1]= ulTmpId;

         }  /* End Else*/

      }  /* End For  */

   }  /* End Function */


/*****************************************************************************************
**/
/* Main Function Accepts from command line :: No Of Nodes, % of Source Node,
% of Sink node  */
/*****************************************************************************************
*/

int main(int  argc,
         char *argv[])
{
    //flushall(void);
    char **cppCheck = NULL;

    BYTE *ucpAdMat,
         *ucpTmpAdMat,
         *ucpPostAdMat,
         *ucpNonTreeAdMat,
         *ucpBuffAdMat,
         *ucpPostOrgAdMat,
           *ucpTmpPostOrgAdMat,
         *ucpReachAdMat;

    unsigned int tmp1,tmp2, uiflag, uiTmpNode;

    unsigned long ulSink,
                  ulCount,
                  ulSource,
```

```
                ulNoOfSink              = 0L,
                ulRowCount,
                    *ulpDestAckMat,
                ulColCount,
                ulNoOfNodes,
                ulArraySize,
                    *ulpUnReachDest,
                ulNoOfSource            = 0L,
                *ulpInDegree,
                *ulpSinkNode,
                *ulpOutDegree,
                ulSinkInDegree          = 0L,
                *ulpSourceNode,
                ulTotalNoOfEdges        = 0L,
                ulSourceOutDegree       = 0L,
                ulNoOfAnchorNodes       = 0L,
                ulAnchorNodesInOutDegree = 0L,
                *ulpSinkArray,
                *ulpAnchorArray,
                 ulTmpAnchor,
                *ulpPostOrder,
                *ulpNodeId,
                 ulRandAnchor,
                 *ulpMsgSrcArr,
                 arrcntsrc,
                 ulNodeNo,
                 ulFlag,
                     *ulpDestArray,
                 ulTmpSource,
                 arrcntsink,
                  *ulpSourceArray,
                 *ulpNodeTrust,
                  ulNotAcknowledged,
                  ulNoOfDest,
                 *ulpReachDest,

                  ulReachCount,
                  ulUnReachCount,
                  ulFailedPathDetect,

                  ulRandSource ;

   long lAckFlag;
/*  struct sRange {
                  unsigned long ulNodeNo;
                  unsigned long ulStart, ulEnd;
```

```
                struct sRange *next;
              };
*/

  struct sRange *sdaRangeMat;
  struct sRange *sdTmp;

  if (argc !=4)
  {
     printf(USAGE,argv[0]);
     exit(-1);
  }

  ulNoOfNodes = strtoul(argv[1],cppCheck,10);
  ulSource    = strtoul(argv[2],cppCheck,10);
  ulSource    = (ulNoOfNodes * ulSource) / 100;
  ulSink      = strtoul(argv[3],cppCheck,10);
  ulSink      = (ulNoOfNodes * ulSink) / 100;
  ulArraySize = sumNatNum(ulNoOfNodes) / 8 + 1;



  if ((ulpNodeTrust = (unsigned long *) calloc(ulNoOfNodes,
                                    sizeof(unsigned long))) == NULL)
  {
     printf("Unable to allocate memory %ld for SourceNode",ulSource);
     exit (-1);
  }


  if ((ucpAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
  {
     printf("Unable to allocate memory %ld",ulArraySize);
     exit (-1);

  }


   if ((ucpBuffAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
  {
     printf("Unable to allocate memory %ld",ulArraySize);
     exit (-1);

  }

    if ((ucpNonTreeAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
```

```
    {
        printf("Unable to allocate memory %ld",ulArraySize);
        exit (-1);

    }



    if ((ucpPostAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
    {
        printf("Unable to allocate memory %ld",ulArraySize);
        exit (-1);
    }

    if ((ulpPostOrder = (unsigned long *) calloc(ulNoOfNodes + 1,
                                        sizeof(unsigned long))) == NULL)
    {
        printf("Unable to allocate memory %ld for SourceNode",ulSource);
        exit (-1);
    }

    if((ulpAnchorArray = (unsigned long *)calloc(ulNoOfNodes, sizeof(unsigned
long))) == NULL)
    {
        printf("Unable to allocate memory %ld for Anchor Nodes", ulNoOfNodes);
        exit(-1);
    }

    if ((ulpInDegree = (unsigned long *) calloc(ulNoOfNodes + 1,
                                        sizeof(unsigned long))) == NULL)
    {
        printf("Unable to allocate memory %ld for InDegree",ulNoOfNodes);
        exit (-1);
    }



    if ((ulpOutDegree = (unsigned long *) calloc(ulNoOfNodes + 1,
                                        sizeof(unsigned long))) == NULL)
    {
        printf("Unable to allocate memory %ld for OutDegree",ulNoOfNodes);
        exit (-1);
    }
```

```
   if ((ulpSinkNode = (unsigned long *) calloc(ulSink + 1,
                                    sizeof(unsigned long))) == NULL)
   {
      printf("Unable to allocate memory %ld for SourceNode",ulSource);
      exit (-1);
   }

   if ((ulpSourceNode = (unsigned long *) calloc(ulSource + 1,
                                    sizeof(unsigned long))) == NULL)
   {
      printf("Unable to allocate memory %ld for SourceNode",ulSource);
      exit (-1);
   }


   if ((ulpNodeId = (unsigned long *) calloc(ulNoOfNodes + 1,
                                    sizeof(unsigned long))) == NULL)
   {
      printf("Unable to allocate memory %ld for SourceNode",ulNoOfNodes);
      exit (-1);
   }




      srand(rand());

/*  ************************************************************************ */
/*  Generate Graph  Randomly if no specific graph manually assigned         */
/*  ************************************************************************ */

   for (ulCount =0; ulCount < ulArraySize; ulCount ++)
   {    srand(time(NULL));
        ucpAdMat[ulCount] = rand() % 256;
   }




/*****************************************************************************************
**/
/* graph can be generated randomly but here I am giving specific graph as input
for testing  */
```

```
/*****************************************************************************************
*/

  /*  ucpAdMat[0]=0;
    ucpAdMat[1]=24;
    ucpAdMat[2]=132;
    ucpAdMat[3]=32;
    ucpAdMat[4]=226;
    ucpAdMat[5]=129;
    ucpAdMat[6]=132;
    ucpAdMat[7]=6;
    ucpAdMat[8]=0;
    ucpAdMat[9]=96;
    ucpAdMat[10]=135;
    ucpAdMat[11]=0;
    ucpAdMat[12]=6;
    ucpAdMat[13]=64;
    ucpAdMat[14]=4;
    ucpAdMat[15]=0;
    ucpAdMat[16]=88;
 */


/*  ********************************************************************** */
/*  Set Source and Sink, calculate InDegree, OutDegree                     */
/*  ********************************************************************** */


   for(ulCount =0; ulCount < ulArraySize; ulCount ++)
    {
            ucpPostAdMat[ulCount] = 0;
    }

     for(ulCount =0; ulCount < ulArraySize; ulCount ++)
    {
            ucpBuffAdMat[ulCount] = 0;
    }

     for(ulCount =0; ulCount < ulArraySize; ulCount ++)
    {
            ucpNonTreeAdMat[ulCount] = 0;
    }


    for (ulCount = 0, ulRowCount =1; ulRowCount <= ulNoOfNodes; ulRowCount
++)
```

```
   {
      ulpInDegree[ulRowCount]  = 0;
      ulpOutDegree[ulRowCount] = 0;
   }

    for(ulRowCount = 1; ulRowCount <= ulNoOfNodes; ulRowCount++)
    {

         if(ulRowCount <= ulSink)
         {
              for(ulColCount = 1; ulColCount <= ulRowCount ; ulColCount++ )
              {    setBitVal(ulRowCount,ulColCount,0,ucpAdMat);}


         }

         setBitVal(ulRowCount,ulRowCount,0,ucpAdMat);

         if(ulRowCount  > (ulNoOfNodes - ulSource) )
         {
           for(ulColCount = ulRowCount; ulColCount <= ulNoOfNodes; ulColCount
++  )
              setBitVal(ulColCount,ulRowCount,0,ucpAdMat);
         }


    }


    for(ulRowCount = 1 ; ulRowCount <= ulNoOfNodes; ulRowCount++ )
    {
      for(ulColCount = 1; ulColCount < ulRowCount ; ulColCount++)
        ulpOutDegree[ulRowCount] +=
                  getBitVal(ulRowCount,ulColCount,ucpAdMat);

    }

   for(ulRowCount=1; ulRowCount <= ulNoOfNodes; ulRowCount ++ )
   {
      for(ulColCount = ulRowCount; ulColCount <= ulNoOfNodes; ulColCount++)
         ulpInDegree[ulRowCount] +=
                   getBitVal(ulColCount,ulRowCount,ucpAdMat);

   }
```

```
for (ulCount                = 1L,
     ulNoOfSink             = 0L,
     ulNoOfSource           = 0L,
     ulSinkInDegree         = 0L,
     ulNoOfAnchorNodes      = 0L,
     ulSourceOutDegree      = 0L,
     ulAnchorNodesInOutDegree = 0L; ulCount <= ulNoOfNodes; ulCount ++)
{
    if (ulpInDegree[ulCount] == 0)
    {
        if (ulCount > ulSource)
            ulpSourceNode[ulNoOfSource] = ulCount;
        ulNoOfSource ++;
        ulSourceOutDegree += ulpOutDegree[ulCount];
        continue;
    }

    if (ulpOutDegree[ulCount] == 0)
    {
        if (ulCount < ulNoOfNodes - ulSink)
            ulpSinkNode[ulNoOfSink] = ulCount;
        ulNoOfSink ++;
        ulSinkInDegree += ulpInDegree[ulCount];
        continue;
    }

    if(ulpOutDegree[ulCount] > 1 || ulpInDegree[ulCount] > 1)
    {
         ulpAnchorArray[ulNoOfAnchorNodes] = ulCount;
         ulNoOfAnchorNodes ++;
           ulAnchorNodesInOutDegree += ulpInDegree[ulCount] +
                                ulpOutDegree[ulCount];
    }
}



  if ((ulpSinkArray=(unsigned long *) calloc(ulNoOfSink + 1,
                                     sizeof(unsigned long))) == NULL)
{
    printf("Unable to allocate memory %ld for SinkNode Array",ulSource);
    exit (-1);
}
```

```
/*  ********************************************************************** */
/*  Print Adjacency Matrix                                                 */
/*  ********************************************************************** */
  /*  for (ulCount = 0, ulRowCount =1; ulRowCount <= ulNoOfNodes; ulRowCount
++)
    {
       printf("\n %4d ",ulRowCount);
       for (ulColCount =1; ulColCount <= ulRowCount; ulColCount ++, ulCount++)
             printf("%3d",getBit(ucpAdMat[ulCount / 8],ulCount%8));
       for (; ulColCount <= ulNoOfNodes; ulColCount ++)
          printf("   ");
       printf(" %d ",ulpOutDegree[ulRowCount]);
    }
    printf("\n      ");

    for (ulColCount =1; ulColCount <= ulNoOfNodes; ulColCount ++)
       printf("%3d",ulpInDegree[ulColCount]);
    printf("\nInitial");
    printf("\n Required Source = %ld \tExisting Source = %ld\n Required Sink =
%ld \tExisting Sink = %ld",ulSource,ulNoOfSource,ulSink,ulNoOfSink);
    printf("\n AnchorNodes = %ld ",ulNoOfAnchorNodes);
    printf("\n OutDegree of Source = %ld\n In Degree of Sink %ld",
                                  ulSourceOutDegree,ulSinkInDegree);
    printf("\n In and Out Degree of Anchor Nodes %ld",
                                  ulAnchorNodesInOutDegree);
*/
    if ((ulpSourceArray=(unsigned long *) calloc(ulNoOfSource + 1,
                                     sizeof(unsigned long))) == NULL)
    {
       printf("Unable to allocate memory %ld for SinkNode Array",ulSource);
       exit (-1);
    }


    //printf("\n Total Edges = %ld", ulTotalNoOfEdges);


    if ((ucpTmpAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
    {
       printf("Unable to allocate memory %ld",ulArraySize);
       exit (-1);
    }

    memcpy(ucpTmpAdMat,ucpAdMat,ulArraySize);
```

```
 arrcntsrc=0;
 arrcntsink=0;
 tmp1=1;
 tmp2=0;

for(ulCount = 1 ; ulCount <= ulNoOfNodes ; ulCount++){

      /*Sink Search*/
   for(tmp1=1; tmp1 <= ulCount ; tmp1++)
   {
      if((getBitVal(ulCount,tmp1,ucpTmpAdMat))==1)
      {
         tmp2=100;

      }

   }

   if(tmp2 != 100){

     ulpSinkArray[arrcntsink] = ulCount;
     arrcntsink++;

   }
   else
     tmp2=0;

   tmp1=0;
   tmp2=0;
   for(tmp1 = ulCount; tmp1 <= ulNoOfNodes; tmp1++)
   {
      if((getBitVal(tmp1,ulCount,ucpTmpAdMat)) == 1)
      {
         /*printf("\n Node No= %d \n  is not source\n",ulCount); */
         tmp2=100;

      }

   }
```

```
      if(tmp2 != 100){

        ulpSourceArray[arrcntsrc] = (ulCount);
        arrcntsrc++;

      }
      else
        tmp2 = 0;

    }



     if((ulpDestArray  = (unsigned long *)calloc((ulNoOfNodes - ulNoOfSource),
                                sizeof(unsigned long))) == NULL)
    {
        printf("Unable to allocate memory %ld for Anchor Nodes", ulNoOfNodes);
        exit(-1);
    }
/*****************************************************************************************/
/*This Function will Create Tree Cover for DAG(initially DAG is not in Postorder
numbered) */
/*****************************************************************************************/



 genTreeCover(ulNoOfNodes,ulpSourceArray,ulpSinkArray,ucpTmpAdMat,
             ulNoOfSource,ulpPostOrder);


/*****************************************************************************/
/*      Take NonTree Nodes in Buffer Matrix                                  */
/*****************************************************************************/

  genNonTreeCover(ulNoOfNodes,ucpAdMat,ucpTmpAdMat,ucpBuffAdMat);




/*****************************************************************************/
/*  Following Code Transforms Given NonTree edge matrix into New Node
Number      */
/*****************************************************************************/
```

```
transformPostOrder(ulNoOfNodes,ulpPostOrder,ucpBuffAdMat,ucpNonTreeAdM
at);



/*****************************************************************************/
/*   This Code will transform Tree Cover into PostOrder Numbered form        */
/*****************************************************************************/



transformPostOrder(ulNoOfNodes,ulpPostOrder,ucpTmpAdMat,ucpPostAdMat);

   for(ulCount = 0 ; ulCount < ulNoOfSink ; ulCount++)
   {
      tmp1 = ulpSinkArray[ulCount];

      ulpSinkArray[ulCount] = ulpPostOrder[tmp1 - 1];


   }



/*****************************************************************************************
*/
/*  This will Merge Post Order tree Cover woth Post order Non Tree Cover
*/
/*****************************************************************************************
*/
    if ((ucpPostOrgAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
   {
      printf("Unable to allocate memory %ld",ulArraySize);
      exit (-1);
   }



mergeTreeNonTreeEdge(ulNoOfNodes,ucpPostAdMat,ucpNonTreeAdMat,ucpP
ostOrgAdMat);
```

```
   /*****************************************************/
   /*      Recalculate  Sources and sink                */
   /*****************************************************/
  for(ulCount = 0; ulCount < ulNoOfSource ; ulCount++)
    ulpSourceNode[ulCount] = 0;

   for(ulCount = 0; ulCount < ulNoOfSource ; ulCount++)
   {
      tmp1 = ulpSourceArray[ulCount];
      ulpSourceNode[ulCount] = ulpPostOrder[tmp1 - 1];

   }



/*****************************************************************************************
***/
/*  This function will assign to a node lowest post order number among all its
descendent     */
/*****************************************************************************************
***/


     assignNodeId(ulNoOfNodes,ucpPostAdMat,ulpNodeId);

/*****************************************************************************************
**/
     if ((sdaRangeMat = (struct sRange *) calloc(ulNoOfNodes ,
                                    sizeof(struct sRange))) == NULL)
   {
      printf("Unable to allocate memory %ld for InDegree",ulNoOfNodes);
      exit (-1);
   }
  /*************************************************************************
This will generate initial NOde intervals                        */
  /*************************************************************************/


   genNodeInterval(ulNoOfNodes,ulpNodeId,sdaRangeMat);

  /*************************************************************************/



  /*********************************************************************/
  /*     This Function Inherite Intervals from Non tree Nodes          */
```

```
  /*************************************************************************/


inheritNodeRange(ulNoOfNodes,ucpPostOrgAdMat,ucpPostAdMat,sdaRangeMa
t,ulpNodeId);

  /*************************************************************************/




 printf("\n *************** Reachability List ************************  \n ");

   for (ulRowCount = 1 ; ulRowCount <= ulNoOfNodes; ulRowCount++)
   {
      sdTmp = &sdaRangeMat[ulRowCount - 1];

      printf("\n Node :: %d  " ,ulRowCount );
      while( sdTmp  !=  NULL )
      {
         printf("[%ld,%ld] ",sdTmp->ulStart, sdTmp->ulEnd);
         sdTmp = sdTmp->next;
      }



   }

   printf("\n  \n");


/*************************************************************************************
*/
  /*            Generate A Reachbility Matrix                                      */

/*************************************************************************************
*/

     if ((ucpReachAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
     {
       printf("Unable to allocate memory %ld",ulArraySize);
       exit (-1);

     }
```

```
    for (ulCount =0; ulCount < ulArraySize; ulCount ++)
    {
        ucpReachAdMat[ulCount] = 0;
    }


   genReachabilityMat(sdaRangeMat,ucpReachAdMat,ulNoOfNodes);




/*************************************************************************************************/
  /*                  Assigning Trust Values to each Node                                          */
/*************************************************************************************************/
  for(ulCount = 1; ulCount <= ulNoOfNodes ; ulCount++)
  {
    ulpNodeTrust[ulNoOfNodes - 1] = 100;
  }


/*************************************************************************************************/

     /*********************************************/
     /*  Generate Random Destination              */
    /*********************************************/



     if ((ulpUnReachDest = (unsigned long *) calloc(ulNoOfDest + 1,
                                    sizeof(unsigned long))) == NULL)
    {
       printf("Unable to allocate memory %ld for SinkNode Array",ulSource);
       exit (-1);
    }
```

```
    srand(time(NULL));

    ulNoOfDest = 1 + (unsigned long ) rand() % (ulNoOfNodes - ulNoOfSource);


    tmp1  = ( rand() % ulNoOfSource);
    ulRandSource = ulpSourceNode[tmp1];

    //printf("\n index = %d  Random Source = %d \n ", tmp1,ulRandSource );

    if( ulNoOfDest >= ((3*ulNoOfNodes)/4))
            ulNoOfDest = ulNoOfDest - (ulNoOfSource + (unsigned
long)(ulNoOfNodes /4) );

    if ((ulpReachDest = (unsigned long *) calloc(ulNoOfDest + 1,
                                    sizeof(unsigned long))) == NULL)
   {
     printf("Unable to allocate memory %ld for SinkNode Array",ulSource);
     exit (-1);
   }

   // printf("\n No Of Destinations selected is :: %d \n", ulNoOfDest);

          ulMisBWrt = 10; //Initiaaly some value other than 0 and 1
            ulTypeOfFail = 10;  //Global value declaration
            ucMisBehType = 10;

    selectRandomDest(ulNoOfDest,
                ulNoOfNodes,
                ulpDestArray,
                ucpPostOrgAdMat);
printf("\n Destinations \n");
  for(ulCount = 0; ulCount < ulNoOfDest ; ulCount++)
  {
    printf(" %d ", ulpDestArray[ulCount]);
  }



  if ((ulpDestAckMat = (unsigned long *) calloc(ulNoOfDest + 1,
                                    sizeof(unsigned long))) == NULL)
   {
     printf("Unable to allocate memory %ld for SinkNode Array",ulNoOfDest);
     exit (-1);
   }
```

```
    ulReachCount = 0;
    ulUnReachCount = 0 ;

    /*****************************************************************************/
    /* Classify the nodes based on reachability and Non Reachability From
Source Node  */
    /*****************************************************************************/

    classifyReachable(ulNoOfDest,
                ulNoOfNodes,
                ulRandSource,
                ulpDestArray,
                &ulReachCount,
                &ulUnReachCount,
                ulpReachDest,
                ulpUnReachDest,
                ucpReachAdMat);


/*******************************************************************************************/




 /*******************************************************************************************/
 /*    Now Send Message to All The nodes                                                    */
 /*******************************************************************************************/

   if ((ulpMsgSrcArr = (unsigned long *) calloc(ulNoOfDest + 1,
                                sizeof(unsigned long))) == NULL)
   {
      printf("Unable to allocate memory %ld for SinkNode Array",ulNoOfDest);
      exit (-1);
   }



   forwardMsgToAllDest(ulRandSource,
                ulNoOfDest,
                ulpDestArray,
                ulReachCount,
                ulUnReachCount,
                ulpReachDest,
                ulpUnReachDest,
```

```
                    ulpDestAckMat,
                    ucpReachAdMat,
                    ucpPostOrgAdMat,
                    ulpMsgSrcArr );



   /*Check acknowledgments to confirm message delivery*/

   lAckFlag = -1;



  /* printf("\n Destinations \n");
   for(ulCount = 0; ulCount < ulNoOfDest ; ulCount++)
   {
      printf(" %d ", ulpDestArray[ulCount]);
   }
  */


   for(ulCount = 0; ulCount < ulReachCount ; ulCount++)
   {
      if( ulpDestAckMat[ulCount] != 1)
      {
            lAckFlag = ulCount;
            break;
      }
   }

   if(lAckFlag != -1 )
   {
     ulFailedPathDetect = ulpReachDest[ulCount];
     printf("\nAcknowledgment did not received from Destination Index = %d
Node =  %d  \n",ulCount, ulFailedPathDetect );
   }

   /*This condition must checked as source may not be reachable to failed path
Dest*/
   if((ulFailedPathDetect < ulRandSource) &&
               (getBitVal(ulRandSource,ulFailedPathDetect,ucpReachAdMat) == 1
))
            ulNotAcknowledged = sendDangerSignal(ulRandSource,
                                       ulFailedPathDetect,
                                         ulpSourceArray,
                                                ucpPostOrgAdMat,
```

```
                                                ucpReachAdMat );
   else
   {
     for(ulCount = 0; ulCount < ulUnReachCount; ulCount++ )
        if(ulpUnReachDest[ulCount] == ulFailedPathDetect )
        {
          ulTmpSource = ulpMsgSrcArr[ulCount];
        }
         ulNotAcknowledged = sendDangerSignal(ulTmpSource,
                                      ulFailedPathDetect,
                                      ulpSourceArray,
                                      ucpPostOrgAdMat,
                                      ucpReachAdMat );


   }

  printf("\n D.S. Detected that Node : %d did not ack \n", ulNotAcknowledged );


printf("\n==================================================
=========\n");

 if((ulpIsoNodeArray  = (unsigned long *)calloc((ulNoOfNodes + 1 ),
                                sizeof(unsigned long))) == NULL)
   {
      printf("Unable to allocate memory %ld for Anchor Nodes", ulNoOfNodes);
      exit(-1);
   }

 for(ulCount = 0 ; ulCount < ulNoOfNodes ; ulCount ++)
 {
    ulpIsoNodeArray[ulCount] = 0;
 }

 /*************************************************************************/
 /*                Failure Detection and Measures                         */
 /*************************************************************************/

  /*************************************************************/
  /***************     Copying Grpah             *************** */
  /*************************************************************/

   if ((ucpTmpPostOrgAdMat = (BYTE *) malloc(ulArraySize)) == NULL)
     {
       printf("Unable to allocate memory %ld",ulArraySize);
```

```
        exit (-1);

      }
    memcpy(ucpTmpPostOrgAdMat,ucpPostOrgAdMat,ulArraySize);


    failureDetection(  ulRandSource,
                    ulNoOfSource,
                    ulNoOfNodes,
                    ulpSourceArray,
                        ulpNodeTrust,
                    ulNotAcknowledged,
                    ulPrevNode,
                    ucpTmpPostOrgAdMat,
                    ucpReachAdMat
                  );



    /*****************************************************************************/
   /*                          FREE ALLOCATED MEMORY                            */
   /*****************************************************************************/


}          /*************End Main*********************************/


/*****************************************************************************/
/*                        End Of the Main Funtion !!        */
/*****************************************************************************/


void genTreeCover(unsigned long ulNodes,
                unsigned long *ulpSourceArray,
                unsigned long *ulpSinkArray,
                BYTE *ucpTmpAdMat,
                unsigned long ulNoOfSource,
                unsigned long *ulpPostOrder  )

{
        unsigned long  long  ldPathEdgeCnt;
        int iCntr,
            iCurSrc,
            iAnchor,
            iCol,
            iAncol,
```

```
    iNodeFlag,
    iTmp;

int   iCurrent,iPrev, iC,jC;
unsigned long *ulpStack,
              ulUnVisit,
              ulPostCount;

long int ulTop,ulZero;
BYTE ucOverflow;
unsigned short int *usipVisited;

 if ((ulpStack = (unsigned long *) calloc(ulNodes + 1,
                                  sizeof(unsigned long))) == NULL)
 {
   printf("Unable to allocate memory %ld for Stack",ulNodes);
   exit (-1);
 }




 if ((usipVisited = (unsigned short int *) calloc(ulNodes + 1,
                                  sizeof(unsigned short int))) == NULL)
 {
   printf("Unable to allocate memory %ld for Stack",ulNodes);
   exit (-1);
 }


 //ldPathEdgeCnt = 0;

ulPostCount = 1;
for(iC = 0 ; iC < ulNodes ; iC ++ )
     usipVisited[iC] = 0;
 // This flag is to know whether every Node is traversed at least once
 iNodeFlag = 0;


 for( iCntr = 0; iCntr < ulNoOfSource     ; iCntr++)
 {

    for(iC = 0; iC < ulNodes; iC++)
       ulpStack[iC] = 0;

    iCurrent = 1;
```

```
          iPrev = 1;
          ulZero=0;
          ulTop = 0 ;
        //Stack pointer incremented before pushing an element\

            ulpStack[ulTop] = ulpSourceArray[iCntr];
            iTmp = ulpSourceArray[iCntr];
            usipVisited[ iTmp - 1 ] = 2 ;


          while(ulTop >= ulZero )
          {

            iPrev = ulpStack[ulTop];

            while( (getBitVal(iPrev,iCurrent,ucpTmpAdMat) != 1) && (iCurrent <
iPrev))
            {
                  iCurrent++;
            }

           // printf("\n iCurrent = %d \n  ", iCurrent);
            if(iCurrent < iPrev)
            {
               ucOverflow = 0;

               ulpStack[++ulTop] = iCurrent;
               usipVisited[iCurrent - 1] = 1;

               /*  Now Delete all the incoming edges on that node*/
               for(iC = iCurrent  ; iC <= ulNodes  ; iC++ )
               {
                  if( iC != iPrev)
                  {
                     setBitVal(iC,iCurrent,0,ucpTmpAdMat);

                  }
               }
            }
            else
            {
              iCurrent = ulpStack[ulTop] + 1;
              iTmp = ulpStack[ulTop];
              ulpPostOrder[iTmp -1 ] = ulPostCount++;
              ulTop--;
              continue;
```

```
        }

        /*Check current node is Sink then Path is completed and print  */

        if( isSinkNode((unsigned long)iCurrent,ucpTmpAdMat) )
        {
          iTmp = ulpStack[ulTop];
          ulpPostOrder[iTmp -1 ] = ulPostCount++;
          ulTop--;
          iCurrent++;

        } /*End If */
        else
          iCurrent = 1;

      }  /* End while */

    } /*end for*/


   free(ulpStack);
   free(usipVisited);

}   /*End Function*/


BYTE isSinkNode(unsigned long ulNode ,
            unsigned char *ucpTmpAdMat)
{
   int iCC, iFlag;
   iFlag = 0 ;
   for( iCC = 1 ; iCC < ulNode ; iCC++ )
   {

     if(( getBitVal(ulNode,iCC,ucpTmpAdMat) == 1) )
       return(0);
   }
   return(1);

}   /*End Function */


void transformPostOrder(unsigned long ulNoOfNodes,
                  unsigned long * ulpPost,
                  BYTE  *ucpInputAdMat,
                  BYTE * ucpPostOrderAdMat)
```

```
{
     unsigned long tmp1, tmp2;
     unsigned long ulRowCount, ulColCount;

     for(ulRowCount = 1 ; ulRowCount <= ulNoOfNodes ; ulRowCount++)
    {
       for(ulColCount = 1; ulColCount < ulRowCount ;  ulColCount++ )
       {
         tmp1 = ulpPost[ulRowCount - 1];
         if(( getBitVal(ulRowCount,ulColCount,ucpInputAdMat)) == 1 )
         {
           tmp2 = ulpPost[ulColCount - 1];
           setBitVal(tmp1,tmp2,1,ucpPostOrderAdMat);
         }
       }

   }

}

void genNonTreeCover(unsigned long ulNoOfNodes,
                     BYTE * ucpOriginalAdMat,
                     BYTE * ucpCoverAdMat,
                     BYTE * ucpBuffAdMat )
{

   unsigned long ulRowCount, ulColCount,tmp1, tmp2, ulResult;

   for(ulRowCount = 1; ulRowCount <= ulNoOfNodes ; ulRowCount++ )
   {
     for(ulColCount = 1; ulColCount < ulRowCount; ulColCount++ )
     {
       tmp1 =  getBitVal(ulRowCount, ulColCount,ucpCoverAdMat);
       tmp2=   getBitVal(ulRowCount, ulColCount,ucpOriginalAdMat);
       ulResult = (tmp1 ^ tmp2) ;

       setBitVal(ulRowCount, ulColCount,ulResult ,ucpBuffAdMat);
     }
   }
}


void genNodeInterval(unsigned long  ulNoOfNodes,
                     unsigned long *ulpNodeId,
                     struct sRange *sdaRange)
{
```

```
   unsigned long ulRowCount;

   for(ulRowCount = 1 ; ulRowCount <= ulNoOfNodes; ulRowCount++)
   {
      sdaRange[ulRowCount -1 ].ulNodeNo =  ulRowCount;
      sdaRange[ulRowCount -1 ].ulStart  =  ulpNodeId[ulRowCount - 1];
      sdaRange[ulRowCount -1 ].ulEnd    =  ulRowCount ;
      sdaRange[ulRowCount -1 ].next     =  NULL;
   }


}

/*****************************************************************************/
BYTE isSourceNode(unsigned long ulNodeNo,
                  unsigned long ulNoOfNodes,
                      BYTE *ucpInputAdMat)
{
  int iCC, iFlag;
      iFlag = 0 ;
      for( iCC = ulNodeNo + 1  ; iCC < ulNoOfNodes ; iCC++ )
      {

        if(( getBitVal(iCC,ulNodeNo,ucpInputAdMat) == 1) )

            return(0);
      }
      return(1);


}

/*****************************************************************************/
  void inheritNodeRange(unsigned long ulNodes,
                      BYTE * ucpNonTreeAdMat,
                      BYTE *ucpAdMat,
                      struct sRange *sdaRMat,
                      unsigned long *ulpNodeId )
  {
       unsigned long ulRowCount,
                     ulColCount,
                     ulCount,
                     ulId1,
                     ulId2,
                     ulNode1,
                     ulSubSumFlag,
```

```
            ulNode2,
            ulTmpId1,
            ulTmpNode1;

 struct sRange *spDerived;
 struct sRange *spBase;
 struct sRange *spTmp,
             *spPrev,
             *spBuffNode;

for(ulRowCount = 1; ulRowCount <= ulNodes; ulRowCount++)
{
    if( !(isSinkNode(ulRowCount,ucpNonTreeAdMat)))
    {
       for( ulColCount = (ulRowCount - 1) ; ulColCount >= 1 ; ulColCount-- )
       {
          if ((getBitVal(ulRowCount, ulColCount,ucpNonTreeAdMat)) ==  1 )
          {
              spBase = &sdaRMat[ulColCount - 1];

              while(spBase != NULL)
              {
                  ulId1 = spBase->ulStart;
                  ulNode1 = spBase -> ulEnd;

                  spDerived = &sdaRMat[ulRowCount - 1 ];

                  ulSubSumFlag = 0;

                  while(spDerived != NULL)
                  {
                     ulId2 = spDerived->ulStart;
                     ulNode2 = spDerived->ulEnd;

                     if( subSumed(ulId1, ulNode1,ulId2, ulNode2))
                     {
                          ulSubSumFlag = 1;
                          break;
                     }
                     else if( (ulId1 <= ulId2) && (ulNode1 >= ulNode2) )
                     {
                         ulSubSumFlag = 1;
                         spDerived -> ulStart = ulId1;
                         spDerived -> ulEnd = ulNode1;

                         delRedundantRange(ulRowCount,
```

```
                                        ulId1,
                                        ulNode1,
                                        spDerived,
                                        sdaRMat  );


                                    break;
                                }
                                else if(( ulId1 < ulId2 ) && ( ulId2 == (ulNode1 +1)))
                                {
                                    ulSubSumFlag  = 1;
                                    spDerived -> ulStart = ulId1;
                                    delRedundantRange(ulRowCount,
                                        ulId1,
                                        ulNode2,
                                        spDerived,
                                        sdaRMat  );



                                    break;

                                }   /*End Else if*/
                                else if(( ulId1 <= ulId2) && (ulNode1 == ulId2) )
                                {
                                    ulSubSumFlag = 1;
                                    spDerived->ulStart = ulId1;
                                    delRedundantRange(ulRowCount,
                                        ulId1,
                                        ulNode2,
                                        spDerived,
                                        sdaRMat  );
                                     break;

                                }
                                else
                                    spDerived = spDerived ->next;
                            } /* End While */

                            if( ulSubSumFlag == 0)
                            {
                                addNodeRange(ulRowCount,ulColCount,ulId1,
ulNode1,sdaRMat);
                            }
                            spBase =  spBase -> next;
                        }
```

```
            }

          }

        } /* End Outer If */

    } /*End Outer For Loop*/

}  /* End Of The Function inheritNodeRange */


int subSumed(unsigned long ulId1,
             unsigned long ulNode1,
             unsigned long ulId2,
             unsigned long ulNode2)
{
    if((ulId2 <= ulId1) && (ulNode2 >=  ulNode1))
      return(1) ;
    else
      return(0);

}

void addNodeRange( unsigned long ulNode,
                   unsigned long ulInheritedNode,
                   unsigned long ulRange1,
                   unsigned long ulRange2,
                   struct  sRange *spRMat  )
{
     struct sRange *spNode;
     struct sRange  *spTmp;
     unsigned long ulRepeated;
     unsigned long ulId,ulTmpNode;


    spTmp = &spRMat[ulNode - 1] ;
    ulRepeated = 0 ;

    while(spTmp->next != NULL)
    {
         ulId = spTmp->ulStart;
```

```
      ulTmpNode = spTmp->ulEnd;

      if ((ulId == ulRange1 ) && (ulTmpNode == ulRange2) )
               ulRepeated = 1;
      else if(subSumed(ulRange1,ulRange2,ulId,ulTmpNode))
           ulRepeated =1;
      else if (subSumed(ulId,ulTmpNode,ulRange1,ulRange2))
      {
        spTmp->ulStart = ulRange1;
        spTmp->ulEnd  = ulRange2;
        ulRepeated = 1;
      }

     spTmp = spTmp->next;
}

/* Checking for Last Node*/
ulId = spTmp->ulStart;
ulTmpNode = spTmp->ulEnd;

if ((ulId == ulRange1 ) && (ulTmpNode == ulRange2) )
              ulRepeated = 1;

if(subSumed(ulId,ulTmpNode,ulRange1,ulRange2))
            ulRepeated =1;

if (subSumed(ulId,ulTmpNode,ulRange1,ulRange2))
      {
        spTmp->ulStart = ulRange1;
        spTmp->ulEnd  = ulRange2;

        delRedundantRange(ulNode,
                          ulRange1,
                          ulRange2,
                          &spRMat[ulNode -1],
                          spRMat);
        ulRepeated = 1;

      }

if(ulRepeated == 0)
{
     spNode = (struct sRange *) malloc(sizeof(struct sRange));

     spNode->ulNodeNo = ulInheritedNode;
     spNode->ulStart = ulRange1;
```

```
            spNode->ulEnd = ulRange2;
            spNode->next = NULL;

            spTmp->next = spNode;
        }

    }  /*End Of the Function*/

    void delNextNode(unsigned long ulNodes,struct sRange *spNode,struct
sRange *spRMat )
    {
        //printf("\n  #######  DELETE  Node Function called ######### \n ");
        struct sRange *spTmpNode;
        spTmpNode = spNode ->next;
        spNode->next = spTmpNode->next ;

    }


void delRedundantRange(  unsigned long ulRowCount,
                         unsigned long ulId,
                         unsigned long ulNode,
                         struct sRange *spNode,
                         struct sRange *sdaRRMat )
 {
        struct sRange *spTmp,
                      *spPrev;

        unsigned long  ulTmpId,
                      ulTmpNode;

        spTmp = sdaRRMat[ulRowCount -1].next;
        spPrev = spTmp;


     /*There may still be subsuming possible in next Nodes*/
        while(spTmp != NULL)
        {

            ulTmpId = spTmp->ulStart;
            ulTmpNode = spTmp->ulEnd;

           if(subSumed(ulTmpId, ulTmpNode,ulId,ulNode) && (spNode != spTmp) )
           {

                 if (spPrev == spTmp)
```

```
            {
               delNextNode(ulRowCount,spNode,sdaRRMat);
               spTmp = spTmp->next;
            }
            else
            {

               delNextNode(ulRowCount,spPrev,sdaRRMat);
               spPrev = spTmp;
               spTmp = spTmp->next;
            }

        }
        else if(subSumed(ulId,ulNode,ulTmpId,ulTmpNode) && (spNode !=
spTmp) )
        {
            spNode->ulNodeNo = spTmp->ulNodeNo;
            spNode->ulStart = spTmp->ulStart;
            spNode->ulEnd = spTmp->ulEnd;

            if (spPrev == spTmp)
            {
               delNextNode(ulRowCount,spNode,sdaRRMat);
               spTmp = spTmp->next;
            }
            else
            {

               delNextNode(ulRowCount,spPrev,sdaRRMat);
               spPrev = spTmp;
               spTmp = spTmp->next;
            }

        }
        else
        {
            spPrev= spTmp;
            spTmp = spTmp->next;

        } /*End else*/

    } /*End While*/

 }/* End Function*/
```

```
void mergeTreeNonTreeEdge(unsigned long ulNoOfNodes,
                          BYTE *ucpPostMat ,
                          BYTE *ucpNonTreeMat,
                          BYTE *ucpPostOrgAdMat)
 {
      unsigned long ulRowCount,
                    ulColCount;
      unsigned short int usiBit, usiTmp1, usiTmp2;

      for(ulRowCount = 1 ; ulRowCount <= ulNoOfNodes; ulRowCount++ )
      {
         for(ulColCount = 1 ; ulColCount <= ulRowCount ;ulColCount++)
         {
           usiTmp1 = getBitVal(ulRowCount, ulColCount, ucpPostMat);
           usiTmp2 = getBitVal(ulRowCount, ulColCount, ucpNonTreeMat);
           usiBit = (usiTmp1 | usiTmp2);
           setBitVal(ulRowCount, ulColCount,usiBit,ucpPostOrgAdMat);
         }
      }


 } /* End Function*/


void genReachabilityMat(struct sRange *sRangeMat, BYTE *ucpReach,unsigned
long ulNodes)
{
  unsigned long ulRowCount,
                ulColCount,
                ulTmp,
                uliC,
                uljC,
                ulFirst,
                ulLast;

  struct sRange *sdNodeRange;

  for(ulRowCount = 1 ; ulRowCount <= ulNodes ; ulRowCount++ )
  {
     sdNodeRange = &sRangeMat[ulRowCount - 1];
     while(sdNodeRange != NULL)
     {
          ulFirst = sdNodeRange ->ulStart;
          ulLast = sdNodeRange->ulEnd;
          for(uliC = ulFirst ; uliC <= ulLast; uliC ++)
```

```
            {
              setBitVal(ulRowCount , uliC , 1 , ucpReach);
            }
            sdNodeRange = sdNodeRange -> next;

        }  /*End While*/

    } /*End For*/

} /* End Function */


void selectRandomDest(unsigned long ulNoOfDest,
                      unsigned long ulNoOfNodes,
                      unsigned long *ulpDest,
                      BYTE *ucpInputGraph )
{

    unsigned long ulCount,
                  ulRandNum,
                  ulTime,
                  ulC,
                  ulTmp ;

    unsigned long *ulpNodeArr;

    if ((ulpNodeArr = (unsigned long *) calloc(ulNoOfNodes + 1  ,
                                        sizeof(unsigned long))) == NULL)
     {
         printf("Unable to allocate memory %ld for Node  Array",ulNoOfNodes);
         exit (-1);
     }

     for(ulCount = 0 ; ulCount < ulNoOfNodes ; ulCount++)
     {
        ulpNodeArr[ulCount] = 0;
     }

    srand(time(NULL));

    for(ulCount = 1; ulCount <= ulNoOfDest; ulCount++ )
    {
       // for(ulTmp = 0; ulTmp < 40 ; ulTmp++)


         srand(rand());
```

```
        ulRandNum = 1 + rand() %  ulNoOfDest;
          ulTmp = ulRandNum;



    if((!isSourceNode(ulRandNum,ulNoOfNodes,ucpInputGraph)) &&
(ulpNodeArr[ulRandNum - 1] == 0))
        {
          ulTmp = ulRandNum;
        }
        else
        {
          for(ulC = 1; ulC <= ulNoOfNodes ; ulC++)
          {
            if(! isSourceNode(ulC,ulNoOfNodes,ucpInputGraph) && ulpNodeArr[ulC -
1] == 0)
              ulRandNum = ulC ;
          }
        }


  if((!isSourceNode(ulRandNum,ulNoOfNodes,ucpInputGraph)) &&
(ulpNodeArr[ulRandNum - 1] == 0))
        {
          //  printf("\n Selected = %d  ",ulRandNum );
            ulpDest[ulCount - 1] = ulRandNum ;
            ulpNodeArr[ulRandNum - 1] = 1;
        }
        //else if
        else
          ulCount -- ;

  } /* End For */

} /* End Function */


/*****************************************************************************************
**********************/
/* Function Classify Reachable and UnReachable Destination nodes from source
Node                          */
/*****************************************************************************************
**********************/

void classifyReachable(unsigned long ulNoOfDest,
```

```
                    unsigned long ulNoOfNodes,
                    unsigned long ulSource,
                    unsigned long * ulpDest,
                    unsigned long *ulpReachCnt,
                    unsigned long *ulpUnReachCnt,
                    unsigned long * ulpReach,
                    unsigned long * ulpUnReach,
                    BYTE *ucpReachMat)
{

    unsigned long ulReachCnt,
                ulUnReachCnt,
                ulCount,
                ulTestNode,
                ReachFlag;

    *ulpReachCnt = 0 ;
    *ulpUnReachCnt = 0;
    ulReachCnt = 0;
    ulUnReachCnt =0;

    for(ulCount = 1; ulCount <= ulNoOfDest; ulCount++)
    {
        ulTestNode = ulpDest[ulCount - 1];
        if(( (ulSource > ulTestNode) &&
getBitVal(ulSource,ulTestNode,ucpReachMat) == 1))
            ulpReach[ulReachCnt++] = ulTestNode;


        else
            ulpUnReach[ulUnReachCnt++] = ulTestNode;

    }
    printf("\n Reachable  \n");
    for(ulCount = 0 ; ulCount < ulReachCnt  ; ulCount++)
    {
      printf("  %d  ", ulpReach[ulCount]);

    }

    printf("\n ");
    printf("\n Unreachable Nodes \n");

    for(ulCount = 0 ; ulCount < ulUnReachCnt ; ulCount++)
    {
      printf("  %d  ", ulpUnReach[ulCount]);
```

```
    }

    printf("\n ");

    *ulpUnReachCnt = ulUnReachCnt ;
    *ulpReachCnt = ulReachCnt  ;

}  /*End Function*/

/*****************************************************************************************/

/*********************************************************************/
/* Return Type is Acknowledgment = 1 if message sent successfully          */
/*********************************************************************/


BYTE sendMessage(unsigned long ulSource,
                 unsigned long ulDest,
                 BYTE * ucpReach,
                 BYTE *ucpAdjMat)
{
  unsigned long ulCurrNode,
                ulCount;


 ulCurrNode = ulSource;


 if( (ulDest < ulSource) && getBitVal(ulSource, ulDest,ucpReach) != 1)
 {
    printf("\nThere is SOmeee mistake Happened  \n ");
 }
 else if(ulRandFailedPath != ulDest )
{

  while(1)
  {
     if( (getBitVal(ulCurrNode,ulDest,ucpAdjMat) )== 1 )
        return(1);
     else
     {
       for(ulCount = ulDest + 1 ; ulCount < ulCurrNode ; ulCount++)
       {
          if(getBitVal(ulCurrNode,ulCount,ucpAdjMat) == 1)
          {
            if(getBitVal(ulCount,ulDest,ucpReach) == 1)
```

```
          {
             ulCurrNode = ulCount;
             break;
          }
         }
      } /*End For */

    } /*End Else*/

 } /*End While*/
  return(1);
} /*End If */

else
{
  while(1)
   {
      if( getBitVal(ulCurrNode,ulDest,ucpAdjMat) == 1 )
      {
           printf("\n ######### Failed Destintaion :: %d \n ", ulDest);
           ulFailedNode = ulDest;
           ulTypeOfFail = getFailureType();
             ulTypeOfFail = 1;
           if(ulTypeOfFail == 1 )           /* MisBehaviour */
           {
                ucMisBehType = getTypeOfMisBehaviour(ulDest,ulDest,ucpAdjMat);

                  if(ulCurrNode != ulSource)
                    ulMisBWrt = misBehaviorWrt();


                  else
                    ulMisBWrt = 0;
           }
            printf("\n FailureType of Node = %d , MIsBehaviour Type = %d  WRT =
%d  \n ",
                                      ulTypeOfFail,ucMisBehType, ulMisBWrt);

           return(0);
      }
      else
      {
        for(ulCount = ulDest + 1 ; ulCount < ulCurrNode ; ulCount++)
        {
          if(getBitVal(ulCurrNode,ulCount,ucpAdjMat) == 1)
          {
```

```
            if(getBitVal(ulCount,ulDest,ucpReach) == 1)
            {
               ulCurrNode = ulCount;
               if(isRandNodeFailure())
               {
                   ulFailedNode = ulCount;
                   printf("\n The Failed Node :: %d  \n", ulFailedNode );
                  // ulCurrNode = ulCount;

                   ulTypeOfFail = getFailureType();
                   if(ulTypeOfFail == 1  )           /* MisBehaviour */
                   {
                         ucMisBehType =
getTypeOfMisBehaviour(ulCurrNode,ulDest,ucpAdjMat);
                           ulMisBWrt = misBehaviorWrt();
                    }
                   printf("\n FailureType of Node = %d , MIsBehaviour Type = %d
WRT = %d\n ",
                                  ulTypeOfFail,ucMisBehType , ulMisBWrt);
                   return(0);
               }
            }
          }
        } /*End For */

      } /*End Else*/

   } /*End While*/

 } /*End Else*/

} /*End Function*/


/*****************************************/
/*  If Return =1 means failure occured      */
/*****************************************/

BYTE isRandNodeFailure()
{
    unsigned long ulLargeNum;
    srand(rand());
    ulLargeNum = rand()% 1000;
    if(ulLargeNum > 400 && ulLargeNum <= 900 )
      return(1);
   else
```

```
      return(0);

}

/*************************************************************/
/*  returning 0: Physical Failure  and   1: Misbehaviour    */
/*************************************************************/

BYTE getFailureType()
{
    unsigned int uiNum;
    srand(time(NULL));
    uiNum = 1 + rand() % 1000;
    if( uiNum > 500)
      return(1);
    else
      return(0);
}

/*********************************************************************/
/* Returning 0 means Not ack and returning 1 means Net forwarding  */
/*********************************************************************/

BYTE getTypeOfMisBehaviour(unsigned long ulNode,unsigned long
ulDest,BYTE *ucpAdjMat)
{
   /*A destination can only misbehave by not giving an ack*/
   if((isSinkNode(ulNode,ucpAdjMat)) || (ulNode == ulDest))
      return(0);
   else    /*Intermediate Node can do Two misbehaviours 1.Not forwarding 2.Not
ack */
   {
      if((rand()%1000) > 500 )
        return(1);
      else
        return(0);
   }

}


/*****************************************************************************/
 /* 0 => misbehavior w r t source   | 1 == misbehavior w r t neighbour   */

/*****************************************************************************/
```

```
unsigned long  misBehaviorWrt()
{
    int iNum;
    iNum = 1 + rand() % 10000;
    if(iNum >  ( 2000 ) && iNum <= ( 7000 ))
      return(1);
    else
      return(0);

}

/*****************************************************************************************
****/


void  forwardMsgToAllDest(unsigned long ulSource,
                    unsigned long ulNoOfDest,
                    unsigned long * ulpDestArray,
                    unsigned long ulReachCnt,
                    unsigned long ulUnReachCnt,
                    unsigned long *ulpReachArray,
                    unsigned long *ulpUnReachArray,
                    unsigned long *ulpAckMat,
                    BYTE *ucpReachAdMat,
                    BYTE *ucpAdjMat,
                    unsigned long *ulpMsgSrcA )
{

      unsigned long ulCount,
                ulDestination,
                ulIndex,
                ulTmpSource;

      unsigned long *ulpCovered;
      BYTE ucAck, ucSent ;

        for(ulCount = 0 ; ulCount < ulNoOfDest; ulCount ++)
      {
        ulpAckMat[ulCount] = 1;
      }


      ulTmpSource = ulSource ;

      ulpMsgSrcA[ulUnReachCnt] = ulTmpSource ;
      printf("\n Source at:  %d is  %d  \n",ulUnReachCnt, ulTmpSource);
```

```
        srand(time(NULL));
        ulIndex = rand() % ulReachCnt;

        /* Randomly select Failed Path*/


        ulRandFailedPath = ulpReachArray[ulIndex];
          printf("\n Destination choose for failure :: %d Index = %d \n ",
ulRandFailedPath, ulIndex);

        if(getBitVal(ulSource,ulRandFailedPath,ucpReachAdMat) == 1)
            ulFailSource = ulSource;



          /* Ack To the Source from Destination of Failed Path is Failed*/
        ulpAckMat[ulIndex] = 0;


        if((ulpCovered = (unsigned long * ) (calloc (ulUnReachCnt + 1,
                                        sizeof(unsigned long)))) == NULL)
          {
             printf("\n Unable to Allocate Memory for %d  ",ulUnReachCnt);
             exit(-1);

          }

        for(ulCount = 0 ; ulCount < ulReachCnt; ulCount++)
        {
            ulDestination = ulpReachArray[ulCount];
            printf("\n Message Sending to Destination = %d  from a Source = %d is",
                                              ulDestination,ulTmpSource);
              if(ulDestination > 0)
                 ucAck =  sendMessage(ulTmpSource,
                                ulDestination,
                                ucpReachAdMat,
                                ucpAdjMat);
            if(ucAck == 1 )
            {
              printf(" ==>  Successful \n " );
            }
            else
           {
```

```
            printf("======>>>>>  Failed \n ");
          }

      }

     /*************************************************************************/
     /*  Request one of the node from unreachable Node to Send message to
Rest        */
     /*************************************************************************/
       for(ulCount = 0; ulCount < ulUnReachCnt ; ulCount ++)
         {
           ulpCovered[ulCount] = 0;
         }



       /*Selected source is covered anyway*/

    while(!(AllCovered(ulpCovered,ulUnReachCnt)) )
    {
       /*Select First Uncovered */
         ulIndex = -1;
       for(ulCount = 0; ulCount < ulUnReachCnt  ; ulCount++ )
       {
            if(ulpCovered[ulCount] == 0)
            {
               ulTmpSource = ulpUnReachArray[ulCount];
               ulIndex = ulCount;
            }
         }

       for(ulCount = 0; (ulCount < ulUnReachCnt)  ; ulCount++)
       {
         if((ulIndex != ulCount) && (ulpCovered[ulCount] == 0)
              && (ulpUnReachArray[ulCount] > ulTmpSource) )
         {
                {
                  ulTmpSource = ulpUnReachArray[ulCount];
                  ulIndex = ulCount;
                  }

           }

       }  /*End For */

         /* Atleast There would be some one  */
```

```
        if(ulIndex != -1 ){
          ulpCovered[ulIndex] = 1;
         ulpMsgSrcA[ulIndex] = ulTmpSource;
      }
        else
        {
          printf("\n**************There is  Some Problem*******************\n");
          exit(-1);
        }


        for(ulCount = 0 ; ulCount < ulUnReachCnt; ulCount++)
      {
           ucSent =0;
           if((ulpCovered[ulCount] == 0 ) && (ulpUnReachArray[ulCount] !=
ulTmpSource) )
           {
           ulDestination = ulpUnReachArray[ulCount];

           ucAck = 0;

              /* Check for Reachability from Source */
              if( (getBitVal(ulTmpSource, ulDestination , ucpReachAdMat) == 1))
                {
                 printf("\n Message Sending to Destination = %d  from a Source =
%d is",
                                              ulDestination,ulTmpSource);
               ulpCovered[ulCount] = 1;

                 if( ulRandFailedPath == ulTmpSource )
                ulRandFailedPath = ulDestination;

                 if(ulDestination > 0) {
                  ulpMsgSrcA[ulCount] = ulTmpSource;
                  ucAck =  sendMessage(ulTmpSource,
                                ulDestination,
                                ucpReachAdMat,
                                ucpAdjMat);

            }
                  ucSent = 1;
             } /*End If */

           } /* End If */
```

```
                if(ucSent)
                 if(  ucAck )
                 {
                    printf(" ==>  Successful \n " );
                 }
                 else
                 {
                   printf("== * ====>>>>>  Failed \n  ");
                 }

        }  /*End For */

     } /* End While */


}/*End Function*/


/*****************************************************************************************/
/*        Function AllCovered  to check whether all Destibnations are covered
*/
/*****************************************************************************************/

int AllCovered(unsigned long * ulpCovered,unsigned long ulUnReachCnt)
{
  unsigned long ulCount;
  int iFlag;

    iFlag = 1;

  for(ulCount = 0 ; ulCount < ulUnReachCnt; ulCount++)
  {
      if(ulpCovered[ulCount] == 0 )
         iFlag = 0;
  }

  return(iFlag);

} /* End Function*/


/*****************************************************************************************/

 /*                    Funtion Send danger Signal                                  */

/*****************************************************************************************
```

```
*/

      unsigned long sendDangerSignal(  unsigned long  ulSource,
                                 unsigned long  ulFailDest,
                                          unsigned long *ulpSrcArray,
                                          BYTE * ucpAdjMat,
                                          BYTE *ucpRMat
                                          )
      {
                  unsigned long  ulCurrentNode;

                  BYTE ucAck;

                    printf("\n Danger Signal is Sent through Failed Path   \n");
                    ulCurrentNode = ulSource;
                    ucAck = 1;
                    while(ucAck)
                    {
                       ulPrevNode = ulCurrentNode;
                         ulCurrentNode =
takeNextHop(ulCurrentNode,ucpAdjMat,ucpRMat,ulFailDest);
                     ucAck =
getAcknowledgment(ulCurrentNode,ulPrevNode,ulSource,ulSource);
                          //printf(" \n ACK For %d  is %d \n", ulCurrentNode, ucAck);
                    }
                 printf("\n  Danger Signal Found that Node: %d does not
Acknowledge \n ", ulCurrentNode);
                 return(ulCurrentNode);

         }        /*End Function */



/****************** ***********************************************************/
/*                Function takeNextHop                                        */
/*****************************************************************************/

 unsigned long  takeNextHop(unsigned long ulCurrNode,
                       BYTE *ucpAdjMat,
                       BYTE * ucpRMat,
                       unsigned long ulFailDest)
 {

 unsigned long ulCount;

 if(  getBitVal( ulCurrNode, ulFailDest, ucpRMat) != 1  )
```

```
{
      printf("\n There is SOmeee mistake Happened \n ");
      exit(-1);
}
else
{
 while(1)
 {
  if( ulCurrNode > ulFailDest ) {
   if( (ulCurrNode != ulFailDest ) )
   {
    if( (getBitVal(ulCurrNode,ulFailDest,ucpAdjMat) ) == 1 )
    {
      // printf("\n %d is  Direct Reachable to  %d \n ", ulCurrNode, ulFailDest);
       ulCurrNode = ulFailDest;
       return(ulCurrNode);
    }
    else
    {
      for(ulCount = ulFailDest + 1 ; ulCount < ulCurrNode ; ulCount++)
      {
         if(getBitVal(ulCurrNode,ulCount,ucpAdjMat) == 1)
         {
           if(getBitVal(ulCount,ulFailDest,ucpRMat) == 1)
           {
              // printf("\n Taking Next hop to %d \n ", ulCount);
               ulCurrNode = ulCount;
               return(ulCurrNode);
           }
         }
      } /*End For */

    } /*End Else*/

   } /*End if */
  }
  else
    {
     printf("\n Source and Destination are same while Taking Hop \n ");
       break;
    }

  }  /* End While */

 } /* End else */
```

```
}  /* End function */

/*****************************************************************************/

BYTE getAcknowledgment(unsigned long ulNode,
                       unsigned long ulPrevious,
                            unsigned long ulSrcNode,
                            unsigned long ulFailedSrc )
{
   if( ulTypeOfFail == 0 )
   {
     if( ( ulFailedNode == ulNode ) )
          return(0);
   }
   else
   {
       if( ulMisBWrt == 0 )  /*MisBehavior  W R T source */
          {
            if( ucMisBehType == 0 ) /* NOt Ack */
          {
               if((ulFailedNode == ulNode)  && (ulSrcNode == ulFailedSrc ) )
                 return(0);
            }
            else
            {
               if(( ulNode != ulFailedNode ) && ( ulFailedNode == ulPrevious ) )
                 return(0);
            }
          }
          else
          {
            if(ucMisBehType == 0 ) /* NOt Ack */
          {
               if((ulFailedNode == ulNode) )
                 return(0);
            }
            else
            {
               if((ulNode != ulFailedNode) && (ulFailedNode == ulPrevious) )
                 return(0);
            }

          }  /*End Else*/

   }    /* End else */
```

```
    return(1);

}   /*End Function*/



/*****************************************************************************************
***********/

void failureDetection(unsigned long ulFailedPathSrc,
                      unsigned long ulNoOfSrc,
                      unsigned long ulNoOfNodes,
                      unsigned long *ulpSrcArr,
                           unsigned long *ulpTrustTab,
                           unsigned long ulNotAckNode,
                           unsigned long ulLastAckNode,
                           BYTE * ucpAdjMat,
                           BYTE * ucpRMat
                          )
{
      unsigned long ulCount,
                    ulTmpPrevNode,
                         ulTmpNode,
                         ucTypeOfFailure,
                         ucMBvrWrt,
                         ucMBvrType,
                    ulTmpCnt;

      BYTE ucReachable,
           ucPhyFail,
             ucMisWrtSrc,
             ucMisBWrtNbr,
           ucAck  ;

      ucReachable = 0;
      ucPhyFail = 10;
      ucMisWrtSrc = 10;
      ucMisBWrtNbr = 10;

      for(ulCount = 0 ; ulCount < ulNoOfSrc ; ulCount++)
      {
        if((ulFailedPathSrc != ulpSrcArr[ulCount]) )
          {

            if(ulpSrcArr[ulCount] > ulNotAckNode)
```

```
            {
            ucReachable = 0;
              ucReachable = getBitVal(ulpSrcArr[ulCount], ulNotAckNode,
ucpRMat );
                 if(ucReachable == 1)
                   break;
          }

          }

      } /* End For */

      if( ucReachable == 1 )
      {
        ucPhyFail = 10;
          ulTmpNode = ulpSrcArr[ulCount];

        ucPhyFail = checkPhysicalFailure(ulFailedPathSrc,
                                         ulTmpNode,
                                         ulNotAckNode,
                                             ulLastAckNode,
                                           ulNoOfNodes,
                                             ucpAdjMat,
                                           ucpRMat   );

        if(ucPhyFail == 1)
             printf("\n Physical Failure Occured at Node %d \n", ulNotAckNode)
        ;
          else if(ucPhyFail == 0)
        {
            printf("\n No Physical Failure Occured  \n");

        }

          ucTypeOfFailure = ulTypeOfFail;
        if( (ucPhyFail == 1) || (ucTypeOfFailure == 0 ))
        {
          takePhyFailAct(ulFailedPathSrc,
                               ulNotAckNode,
                       ulLastAckNode,
                               ulNoOfNodes,
                       ucpAdjMat,
                       ucpRMat);
```

```
        } /*end if*/

    }  /*End If*/
    else
    {
        ulTmpNode = 0;
        for(ulTmpCnt = ulNotAckNode + 1 ; ulTmpCnt <= ulNoOfNodes ;
ulTmpCnt++)
        {


           if(getBitVal(ulTmpCnt , ulNotAckNode, ucpRMat ) == 1)
             {
                  ulTmpNode = ulTmpCnt;
                       break;

             }
          }   /* End For */

                if( ulTmpNode > 0 )  {


                 ucPhyFail = checkPhysicalFailure(ulFailedPathSrc,
                                    ulTmpNode,
                                    ulNotAckNode,
                                        ulLastAckNode,
                                      ulNoOfNodes,
                                        ucpAdjMat,
                                      ucpRMat   );

            } /*End If */


              if(ucPhyFail == 1)
                printf("\n Physical Failure Occured\n")      ;

              else
                {

                     if( ucPhyFail == 0 )
                    printf("\n No Physical Failure Occured  \n");


                }
```

```
            if( ucPhyFail == 1 )
          {
            takePhyFailAct(ulFailedPathSrc,
                        ulNotAckNode,
                  ulLastAckNode,
                        ulNoOfNodes,
                  ucpAdjMat,
                  ucpRMat);


          } /*end if*/



    } /*End else */

/*****************************************************************************************
**************/
    if(ucPhyFail == 10 )
    {
      ucReachable = 0;
        ulTmpNode = 0;
      for(ulCount = ulNotAckNode + 1 ; ulCount < ulNoOfNodes ; ulCount++)
      {
        if((ulFailedPathSrc != ulCount) )
          {
           if(ulCount > ulNotAckNode)
             {

               ucReachable = getBitVal(ulCount, ulNotAckNode, ucpRMat );
                 if(ucReachable == 1)
                 {
                   ulTmpNode = ulCount;
                   break;
                 }
          }

          }

       }/* End For */

       if( (ulCount <  ulNoOfNodes) && (ucReachable == 1) )
         {

            ucAck = getAcknowledgment(ulNotAckNode,ulLastAckNode,
```

```
ulTmpNode, ulFailedPathSrc );
            if(ucAck == 1)
            {  printf("\n There is Physical Failure  Occured  \n");
               ucPhyFail = 1;

            }
            else
              ucPhyFail = 0;
         }



   }



/*****************************************************************************************
*********/

     if((((ucPhyFail == 0) && ( ulLastAckNode != ulFailedPathSrc )) && (ulMisBWrt
== 0)) || (ulTypeOfFail == 1)  )
     {
          if( (ucReachable == 1) && (ucPhyFail == 0) )
        {
          ucMisWrtSrc = 10;

          if(getBitVal(ulpSrcArr[ulCount], ulLastAckNode, ucpRMat ) == 1)

          {
                 ucMisWrtSrc = checkMisBehWrtSrc(
                                ulFailedPathSrc,
                              ulpSrcArr[ulCount],
                            ulNotAckNode,
                                ulLastAckNode,
                                ulNoOfNodes,
                                ulpTrustTab,
                                ucpAdjMat,
                                ucpRMat  ) ;

               if(ucMisWrtSrc == 0)
                    if((ulTypeOfFail == 1) && (ulMisBWrt = 0))
                      ucMisWrtSrc = 1;

          }
          else
          {
```

```
                ucReachable = 0;
                for(ulTmpCnt = 0; ulTmpCnt < ulNoOfSrc; ulTmpCnt++)
                {
                    if((ulFailedPathSrc != ulpSrcArr[ulCount]) )
                    {
                      if(ulpSrcArr[ulCount] > ulLastAckNode)
                      {
                        ucReachable = 0;
                        if((getBitVal(ulpSrcArr[ulTmpCnt], ulLastAckNode, ucpRMat) ==
1))
                     {
                          ucReachable = 1 ;
                          break;
                       }
                      }
                    }

                  }  /* End For */
                if((ucReachable == 1) && (ulTmpCnt < ulNoOfSrc) &&
(ulpSrcArr[ulTmpCnt] > ulNotAckNode) )
                {
                   ucMisWrtSrc = checkMisBehWrtSrc(
                                    ulFailedPathSrc,
                                  ulpSrcArr[ulTmpCnt],
                                ulNotAckNode,
                                    ulLastAckNode,
                                    ulNoOfNodes,
                                    ulpTrustTab,
                                    ucpAdjMat,
                                    ucpRMat  ) ;

                  if(ucMisWrtSrc == 0)
                       if((ulTypeOfFail == 1) && (ulMisBWrt = 0))
                         ucMisWrtSrc = 1;


                }

              } /* End  Else */
```

```
        }  /* End If */

    } /* End If  */
    else if((ucPhyFail == 0) && (ulLastAckNode == ulFailedPathSrc ))
    {
       ucReachable = 0;
       for(ulCount = ulLastAckNode + 1 ; ulCount < ulNoOfNodes ; ulCount++)
       {
         if((ulFailedPathSrc != ulCount) )
           {
            if(ulCount > ulNotAckNode)
              {

                 ucReachable = getBitVal(ulCount, ulNotAckNode, ucpRMat );
                   if(ucReachable == 1)
                   {
                     ulTmpNode = ulCount;
                     break;
                   }
            }

           }

       }/* End For */

       if( (ulCount <  ulNoOfNodes) && ucReachable == 1 )
         {

             ucAck = getAcknowledgment(ulNotAckNode,ulLastAckNode,
ulTmpNode, ulFailedPathSrc );
             if(ucAck)
             {
                printf("\n There is problem with respect to source   \n");
                ucMisWrtSrc = 1;
             }
         }



    }

    if((ucMisWrtSrc != 1 ) && (ucPhyFail == 0) && (ulMisBWrt == 1) )
    {

              printf("\n Misbehavior W r t neighbouring Node Occured!! ");
```

```
            ucMisBWrtNbr =  checkMisWrtNbr(         ulFailedPathSrc,
                                ulFailedPathSrc,
                                    ulNotAckNode,
                                        ulLastAckNode,
                                        ulNoOfNodes,
                                          ulNoOfSrc,
                                          ulpSrcArr,
                                          ulpTrustTab,
                                        ucpAdjMat,
                                        ucpRMat  );
    }

  // printf("\n Checked:: Came out of for Loop  \n ");

}  /*End function*/
/*****************************************************************************************
********/



/*****************************************************************************************
******/
BYTE checkPhysicalFailure(unsigned long ulFailedSrc,
                    unsigned long ulSrcNode,
                    unsigned long ulNotAckNode,
                        unsigned long ulLastAckNode,
                          unsigned long ulNoOfNodes,
                        BYTE *ucpAdjMat,
                          BYTE *ucpRMat )
{
            unsigned long ulCurrent,
                        ulCount,
                            ulTmpDest,
                        ulPrevious;

              BYTE ucAck;
              ucAck = 1;
              ulCurrent = ulSrcNode;
              ulTmpDest = 0;

                for(ulCount = ulNotAckNode + 1 ;ulCount < ulSrcNode ; ulCount++)
                  {
                  if(getBitVal(ulSrcNode, ulNotAckNode, ucpAdjMat) == 1)
                  {  ulTmpDest= ulSrcNode;
                     break;
                  }
```

```
                    if(( getBitVal(ulCount, ulNotAckNode, ucpAdjMat) ) == 1)
                    {
                      if((getBitVal(ulSrcNode, ulCount, ucpRMat)) == 1)
                        {   ulTmpDest = ulCount ;  break; }
                        else
                          continue;

                    }

                 }

            printf("\n Temp Dest is : %d  \n ", ulTmpDest);
              printf("\nUlCurrent Node is in PhyFail %d   \n", ulCurrent);

              if(ulTmpDest)
              {
                while( ulCurrent != ulTmpDest )
                {

                   ulPrevious = ulCurrent ;
                     ulCurrent = takeNextHop(ulCurrent, ucpAdjMat,ucpRMat,
ulTmpDest );
                     if(ulCurrent == ulPrevious)
                      break;

                }
                  if(ulCurrent == ulTmpDest)
                  ucAck = getAcknowledgment(
                                    ulNotAckNode,
                                          ulTmpDest,
                                          ulSrcNode,
                                          ulFailedSrc
                                    );
              if((ucAck) || (ulTypeOfFail == 1) )
                    return(0); //No Physical Failure
                  else
                    return(1); //Physical Failure


            }
              else
                printf("\n Could not Found one Alternative for Cheking physical
Failure  \n ");
```

```
} /*End Function*/

/*****************************************************************************/
BYTE checkMisBehWrtSrc(unsigned long ulFailedPathSrc,
                       unsigned long ulSrcNode,
                            unsigned long ulNotAckNode,
                       unsigned long ulLastAckNode,
                            unsigned long ulNoOfNodes,
                            unsigned long *ulpTrustTab,
                            BYTE * ucpAdjMat,
                            BYTE * ucpRMat
                      )

{
                  unsigned long  ulTmpNode,
                                 ulTmpCnt,
                                      ulCount,
                                      ulPrevious,
                                      ulCurrNode;


                   BYTE ucAck;



                     ulCurrNode = ulSrcNode;

                     ucAck = 1;

                     while( ucAck && (ulCurrNode != ulLastAckNode))
                     {

                           ulPrevious = ulCurrNode;
                           ulCurrNode = takeNextHop(ulCurrNode, ucpAdjMat, ucpRMat,
ulNotAckNode);
                                 if(ulCurrNode == ulPrevious)
                                 {
                                    printf("\n Hop Can no be taken Further !! \n");
                                      break;
                                 }

                     }
                            ucAck = getAcknowledgment(ulNotAckNode,
                                          ulLastAckNode,
                                                ulSrcNode,
```

```
                                        ulFailedPathSrc);

                if(((ucAck == 1) && (ulCurrNode == ulLastAckNode)) ||
                                ((ulTypeOfFail == 1) && (ulMisBWrt == 0 ) ))
                {


                        printf("\nMisBehavior occured W R T Source Now Detecting
Who is Culprit Among %d and % d \n ", ulLastAckNode, ulNotAckNode );

                         /*Step 1 : Find a Another Destination Which is Reachable
to Not Ack Node by excluding
                                        Last Ack Node   */
                      if( ulLastAckNode == ulFailedPathSrc ) //If path contains just two
nodes
                         {   printf("\n The Node %d  is Culprit and will be Punished
Now \n", ulNotAckNode);
                            return(1);
                         }
                         else
                         {
                          /* Here Both the Nodes needs to be checked separately */
                            ulTmpNode = 0;
                            for( ulCount = ulNotAckNode + 1 ; ulCount <=
ulNoOfNodes ; ulCount++ )
                            {
                               if(getBitVal(ulCount, ulNotAckNode, ucpAdjMat) == 1)


                                    if(getBitVal(ulSrcNode, ulCount, ucpRMat) == 1)
                                    {
                                       ulTmpNode = ulCount;
                                       break;
                                    }
                             } /*end For */

                             ucAck = 0;
                                 if( ulTmpNode > 0)

                                 {
                                     ucAck = getAcknowledgment(ulNotAckNode,
                                        ulTmpNode,
                                                ulSrcNode,
                                                ulFailedPathSrc);
```

```
                                        if(ucAck == 1)
                                        {
                                          printf("\n Node %d is Misbehaving W r t Source
%d  Hence Punishmened \n ",ulNotAckNode, ulFailedPathSrc);

                                            takeActMisBehWrtSrc(ulNotAckNode,
                                        ulFailedPathSrc,
                                              ulNoOfNodes,
                                        ulpTrustTab,
                                              ucpAdjMat,
                                              ucpRMat);
                                              printf("\nAction Taken Againest %d Node \n",
ulNotAckNode);

                                           return(1);

                                        }    /*End if */


                                    } /*End if */

                                      /*Test For the Second node */

                                       ulTmpNode = 0;
                                     for( ulCount = 1 ; ulCount < ulLastAckNode ;
ulCount++ )
                                     {
                                       if((getBitVal( ulLastAckNode,ulCount, ucpAdjMat) ==
1) && (ulCount != ulNotAckNode) )
                                    {

                                              ulTmpNode = ulCount;
                                              break;
                                         }


                                     } /*end For */
                                       if(ulTmpNode > 0)
                                       {
                                    ucAck = 0;
                                         ucAck = getAcknowledgment(ulLastAckNode,
                                                        ulTmpNode,
                                                              ulSrcNode,
                                                              ulFailedPathSrc);
                                 if(ucAck == 1)
                                        {
```

```
                                    printf("\n Node %d is Misbehaving W r t Source
%d  Hence Punishmened \n ",ulLastAckNode, ulFailedPathSrc);

                                       takeActMisBehWrtSrc(ulLastAckNode,
                                    ulFailedPathSrc,
                                          ulNoOfNodes,
                                    ulpTrustTab,
                                          ucpAdjMat,
                                          ucpRMat);
                                       printf("\nAction Taken Againest %d Node \n",
ulLastAckNode);

                                      return(1);

                                    }    /*End if */



                                 }

                              }  /*End Else */



                           /*Now the Task is to Finc Who is misbehaving W R T Source
Either Last Ack node or Not ack node */



                     }
                          else
                            return(0);

  } /*End Fuinction */


/*****************************************************************************************
****************/


/*****************************************************************************************
****************/
```

```
                        BYTE   checkMisWrtNbr(unsigned long ulFailedPathSrc,
                            unsigned long ulAlternateSrc,
                                unsigned long ulNotAckNode,
                                    unsigned long  ulLastAckNode,
                                    unsigned long ulNoOfNodes,
                                      unsigned long ulNoOfSrc,
                                      unsigned long *ulpSrcArr,
                                      unsigned long *ulpTrustTab,
                                    BYTE *ucpAdjMat,
                                    BYTE *ucpRMat  )

{
                    BYTE ucAck;

                        unsigned long ulCount,
                                    ulTmpNode;

                           ucAck = 10;
                        if(ulLastAckNode == ulFailedPathSrc)
                        {
                          printf("\n CulPrit Node is %d Misbehaving W R T NBR
Source %d  \n",

ulNotAckNode,ulFailedPathSrc );
                        }
                      else if((ulMisBWrt == 1) && (ulTypeOfFail == 1)  )

                      {
                            printf("\nMisbehavior w r t Neighbour is occured \n");
                              printf("\nDetecting Culprit Node .....\n");

                              for(ulCount = ulNotAckNode + 1; ulCount <
ulNoOfNodes ; ulCount++)
                              {
                                 if( (getBitVal(ulCount, ulNotAckNode,ucpAdjMat)
== 1) && (ulCount != ulLastAckNode))
                                   {
                                     ucAck = getAcknowledgment(
                                      ulNotAckNode,
                                      ulCount,
                                      ulFailedPathSrc,
                                      ulFailedPathSrc );
                                      break;
                                   }
                              } /* End for */
```

```
                                    if(ucAck == 1)
                                    {
                                       takeActMisBehWrtNbr(
                                   ulNotAckNode,
                                        ulLastAckNode,
                                        ulNoOfNodes,
                                          ulpTrustTab,
                                        ucpAdjMat,
                                        ucpRMat  );

                                          return(1);
                                  }
                                    else
                                    {
                                         for(ulCount = ulLastAckNode +1 ; ulCount <
ulNoOfNodes ; ulCount++)
                                        {
                                          if(
(getBitVal(ulLastAckNode,ulCount,ucpAdjMat) == 1) && (ulCount !=
ulNotAckNode))
                                             {
                                                 ucAck = 10;
                                                 ucAck = getAcknowledgment(
                                                 ulCount,
                                                 ulLastAckNode,
                                                 ulFailedPathSrc,
                                                 ulFailedPathSrc );
                                                 break;
                                             }

                                          } /*End For */
                                       if(ucAck == 1)
                                       {
                                        takeActMisBehWrtNbr(
                                   ulNotAckNode,
                                        ulLastAckNode,
                                        ulNoOfNodes,
                                          ulpTrustTab,
                                        ucpAdjMat,
                                        ucpRMat  );

                                          return(1);
                                   }


                                    }
```

```
                                printf("\n Action Taken Against Misbehavior w r t
neighbour  \n");

                        }  /*End Else */



 } /*End Function */



 /*****************************************************************************/
 void takeActMisBehWrtNbr(
                        unsigned long ulNotAckNode,
                        unsigned long ulLastAckNode,
                       unsigned long ulNoOfNodes,
                        unsigned long *ulpTrustTab,
                        BYTE * ucpAdjMat,
                        BYTE * ucpRMat
                       )

{
    unsigned long ulCount;
    BYTE ucBit;

    reduceTrust(ulNotAckNode, ulpTrustTab);
    reduceTrust(ulLastAckNode, ulpTrustTab);

    if(ulpTrustTab[ulNotAckNode - 1] < 40 )
      { isolateNode(ulNotAckNode, ulNoOfNodes, ucpAdjMat);}

    else if(ulpTrustTab[ulLastAckNode - 1] < 40)
    {   isolateNode(ulNotAckNode, ulNoOfNodes, ucpAdjMat);}

    else
    {
      for(ulCount = ulLastAckNode + 1 ; ulCount <= ulNoOfNodes; ulCount++)
      {
          if( getBitVal(ulCount, ulLastAckNode, ucpAdjMat) == 1)
            {
                setBitVal(ulCount, ulNotAckNode, 1, ucpAdjMat);
            }
      }

      setBitVal(ulLastAckNode, ulLastAckNode, 0,ucpAdjMat);
```

```
    } /* end Else */

}  /*  end function */


/*****************************************************************************/
void reduceTrust(unsigned long ulNodeNo,
                 unsigned long * trustTable
               )
{
      BYTE ucPunishment;

      ucPunishment = 5;

      trustTable[ulNodeNo - 1] -= ucPunishment;

}




void takePhyFailAct (
                unsigned long ulFailPSrc,
                unsigned long ulNotAckNode,
                unsigned long ulNoOfNodes,
                unsigned long ulLastAckNode,
                BYTE *ucpAdjMat,
                BYTE *ucpRMat )
{
       unsigned long ulCount, ulTmpCnt;


          /* Isolate the node */
          for(ulCount = ulNotAckNode + 1;  ulCount <= ulNoOfNodes; ulCount++)
          {
            if( (getBitVal(ulCount,ulNotAckNode, ucpAdjMat) == 1))
            {
              for(ulTmpCnt = 1; ulTmpCnt < ulNotAckNode; ulTmpCnt ++ )
              {
                if((getBitVal(ulNotAckNode, ulTmpCnt, ucpAdjMat)) == 1)
                     setBitVal(ulCount, ulTmpCnt, 1, ucpAdjMat);

              }
              setBitVal(ulCount, ulNotAckNode, 0,ucpAdjMat);
            }
          }
```

```
        ulpIsoNodeArray[ulNotAckNode - 1] = 1;

        printf("\n Action W R T Physical failure is Taken !! \n ");
}  /* End Function */

/*****************************************************************************************/



void takeActMisBehWrtSrc(unsigned long ulAccusedNode,
                unsigned long ulSrcNode,
                    unsigned long ulNoOfNodes,
                    unsigned long *ulpTrustTable,
                    BYTE *ucpAdjMat,
                    BYTE *ucpRMat)
{

       BYTE ucBit;
       unsigned long ulCount, ulTmp;

       reduceTrust(ulAccusedNode, ulpTrustTable);

       if(ulpTrustTable[ulAccusedNode - 1] < 40 )
           isolateNode(ulAccusedNode, ulNoOfNodes, ucpAdjMat);
         else
         {
           for (ulCount = ulAccusedNode + 1;
                ulCount < ulNoOfNodes; ulCount++  )
         {
                ucBit = 0;
                ucBit = getBitVal(ulCount, ulAccusedNode, ucpAdjMat);
                  if(ucBit)
                  {
                    for(ulTmp = 0 ; ulTmp < ulAccusedNode ;ulTmp ++ )
                    {
                      if( getBitVal(ulAccusedNode, ulTmp, ucpAdjMat) == 1 )
                          setBitVal(ulCount, ulTmp, 1, ucpAdjMat);
                    }
                    setBitVal(ulCount, ulAccusedNode, 0, ucpAdjMat );
                  }
           }
         }
}  /* END Function */

/*****************************************************************************************
```

****/

```
void isolateNode(unsigned long ulAccusedNode,
                 unsigned long ulNoOfNodes,
                     BYTE * ucpAdjMat)
{
  unsigned long ulCount;
  unsigned long ulTmpCnt;

  for(ulCount = ulAccusedNode + 1;
      ulCount <= ulNoOfNodes ;
        ulCount++)
  {
    if( getBitVal(ulCount, ulAccusedNode, ucpAdjMat) == 1)
      {
        for(ulTmpCnt = 1; ulTmpCnt < ulAccusedNode ;ulTmpCnt++)
        {
          if(getBitVal(ulAccusedNode, ulTmpCnt, ucpAdjMat) == 1)
            setBitVal(ulCount, ulTmpCnt,1,ucpAdjMat);
        }
      }

  }  /* End For */

  ulpIsoNodeArray[ulAccusedNode - 1] = 1;

}  /*End Funtion */
```