PARTNER SELECTION TECHNIQUES FOR TIMESTAMPED ANTI-ENTROPY PROTOCOLS

By

PRAKASH JOHN THOMAS

Bachelor of Engineering

Bharathiar University

Coimbatore, India

1990

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
July 1993

OKLAHOMA STATE UNIVERSITY

PARTNER SELECTION TECHNIQUES FOR TIMESTAMPED ANTI-ENTROPY PROTOCOLS

Thesis Approved :

Mitchell L Neilsen
Thesis Adviser

Thomas C. Collins
Dean of the Graduate College

ACKNOWLEDGEMENTS

I wish to express my sincere appreciation and thanks to Dr. Mitchell Neilsen for his encouragement and advise throughout my thesis research. His valuable guidance and in depth knowledge of my area of interest helped speed up this research. I would like to thank Dr. K.M.George and Dr. Marvin Stone for serving on my graduate committee. Special thanks go to Dr. Mansur Samadzadeh for his suggestions in writing this thesis and Dr. Dan Storm for giving me an opportunity to improve my computer skills in the Agricultural Engineering Department.

I would like to thank my parents for their wholehearted, unflinching, and relentless perseverance in everything that I have done so far. I would also like to thank my brothers Pradeep and Dileep for their support. Last, but not least, I wish to thank Arti for her love, faith, and understanding in my abilities.

TABLE OF CONTENTS

Chapter Page

Chapter Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

CHAPTER I

INTRODUCTION

A distributed system is a collection of computers located at different sites connected by a network. Processes communicate by exchanging messages. Each data item can either be stored at exactly one site or replicated and stored at different sites (Figure 1). Copies of the data that are stored at different sites are referred to as replicas. For simplicity, we assume that all data items are replicated at each site.

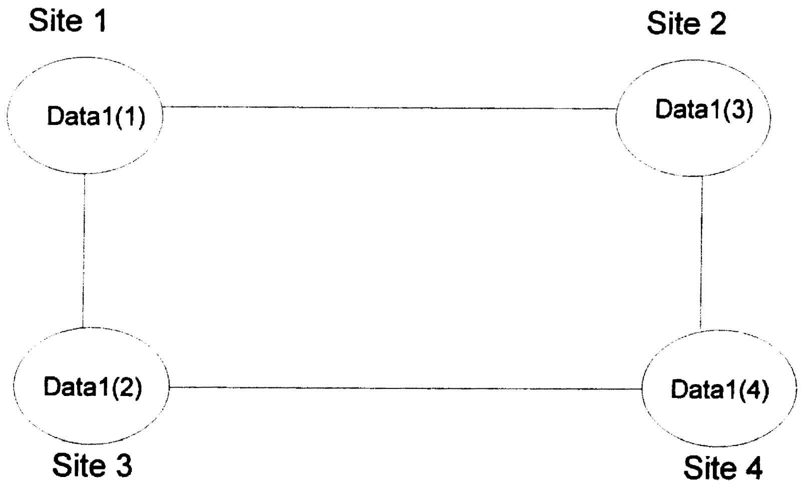

Figure 1. Replicated Data

These replicas must be *consistent.* Several different types of different types of consistency are possible.

1.1 Types of Consistency

Consistency protocols that require all copies of data objects to be consistent or identical at all times are called *strong consistency protocols* [Bernstein87]. Such a protocol is used when consistent information is essential. A common example is a database management system that maintains the balance of an account in a bank. Whenever a transaction is performed, changes in the account balance are immediately reflected at all sites. Thus, all updates are atomic. For many applications, atomic updates are unnecessary.

In order to ensure that updates originating at a given site are observed at all other sites, *weak consistency protocols* can be used. Weak consistency protocols only guarantee that the updates will eventually reach all sites. *Timestamped anti-entropy protocol*s are a particular class of weak consistency protocols[Demers88, Golding93a]. After an update occurs at any given site, that site selects a partner to enter into an anti-entropy session. During an anti-entropy session, sites exchange information. In this way, the update eventually reaches all other sites. The partner selection policy can greatly influence the performance and time required for an update to reach all sites.

1.2 Outline of Work

This thesis is to primarily investigate and compare the performance obtained using various partner selection policies. First, we propose new partner selection policies for propagating updates efficiently. Then, we evaluate these policies using a Monte Carlo simulator to solve the Markov model. Measures to be computed include distance traveled and communication latency. Finally, we derive a new measure of consistency. In particular, the following steps are carried out :

1. In Chapter II, weak consistency protocols are described in greater detail. Different types of consistency guarantees can be made by the communication protocol. These include guarantees on the reliability, order and time of message delivery. We are primarily interested in weak consistency guarantees that ensure reliable and eventual delivery of messages.

2. Secondly, different protocols can be used to communicate with a group of sites. These include Direct Mail, Rumor Mongery, and Anti-entropy. Our focus is on Anti-entropy Protocols.

3. In Chapter III, we refine our focus and discuss Timestamped Anti-entropy

Protocols (TSAE). The effectiveness of any such protocol depends on the partner selection policy used. We propose several new policies. In particular, a partner selection policy for reducing the network traffic is proposed. For example, using intercontinental links should be avoided as much as possible because of their high cost. In this case, anti-entropy sessions can be completed with near neighbors and then finally with replicas across intercontinental links.

4. In Chapter IV, a Monte Carlo simulation is used to simulate the propagation of update messages. The effect of different partner selection policies on system consistency is studied and analyzed. Existing partner selection policies and new selection policies are tested and compared. The total distance traveled by all update messages is found. The minimal distance traveled is computed using a spanning tree.

5. Finally, we summarize our results.

CHAPTER II

WEAK CONSISTENCY PROTOCOLS

There are many applications in which strong consistency is an unnecessary constraint. Multiple copies of data objects are not required to be consistent or identical at all times, but they are only required to eventually become consistent. This is called *weak consistency* [Golding93b]. Such an application is used when consistent information is not very critical or vital. A common example is Usenet News or weather report data. At any given time the news articles available at different sites may be different. However, all of the news will eventually become consistent, provided there are no updates.

Wide area networks use weak consistency protocols to improve their scalability and fault tolerance. Weak consistency protocols will be used in the future for mobile computing. To meet availability demand, data replication is used. Replication is dynamic; servers are added or removed depending on the demand. The system is asynchronous and the servers are independent . Synchronous cooperation between different sites is not required. This is a unique feature that distinguishes a weak consistency protocol from a strong consistency protocol. Strong consistency protocols require synchronous cooperation. Consequently, they only work well with small numbers of replicas; that is, they do not scale well. Furthermore, they are not suited for interactive applications because of their poor response time[Golding93b].

2.1 Types of Consistency

Levels of consistency in a replication protocol depend on consistency guarantees made by the communication protocol. Such guarantees include:

a. Message Delivery
b. Time of Delivery
c. Message Ordering

These consistency guarantees are explained in the following subsections.

2.1.1 Message Delivery

Delivery of messages can be done either *reliably* or with *best effort.* When a message is delivered reliably, its arrival is guaranteed. When a message is delivered with best effort, the system will make an attempt to deliver the message, but the delivery of the message is not guaranteed.

2.1.2 Time of Delivery

Delivery of messages can be done either *synchronously* or *eventually*. When a message is delivered synchronously, it is delivered within a finite amount of time. When a message is delivered eventually, it is done so within a finite, but unbounded time.

2.1.3 Message Ordering

Delivery of messages to processes can be done in any order, it can even be in an order totally different from the way in which they were actually received. Messages can be ordered in a total order, temporal order,causal order [Lamport 1978], FIFO, etc.

2.2 Group Communication Protocols

Since there are multiple copies of data objects stored at different sites, there must be a mechanism by which sites communicate. The sites communicate by using *object managers*. The object managers handle all accesses to the data objects. The managers can be tailored to meet specific demands. We assume that Group Communication Protocols are used. In a

Group Communication Protocol, the set of processes are grouped together. Suppose there are *n* sites. In [Demers88], three Group Communication Protocols are specified. The Group Communication Protocols are:

a. Direct Mail

b. Rumor Mongery

c. Anti-entropy

These protocols are discussed in greater detail in the following subsections.

2.2.1 Direct Mail

> Each and every new update is immediately mailed from its entry site to all other sites using a single unreliable multicast datagram. Direct mail generates *n* messages per update; each message traverses all the network links between its source and destination. So, ... the traffic is proportional to the number of sites times the average distance between sites[Demers88].

This protocol is reasonably efficient, but not entirely reliable. Individual sites cannot always know well in advance about all other sites. A partial solution to this problem is manual intervention by system administrators. This solution works well for a small number of sites, but as the number of sites grow, this solution becomes intangible. Datagrams are used to send messages, and so packets or in this case mail messages may be lost.

Queues are used to keep the messages at the server so that the sender is not delayed. These queues are maintained by the mail server, on stable storage, to prevent it from being affected by disk crashes. Messages may be discarded when the queues overflow or their destinations are inaccessible for a long time and so direct mail is not reliable [Demers88].

2.2.2 Rumor Mongery

In this protocol, all sites are 'ignorant' initially, but when a site receives a new update it suddenly becomes active; that is, it treats the new update as a 'hot rumor'. As long as a site is holding a hot rumor, another site is periodically chosen at random. The site holding the hot rumor makes sure that the site that has just been chosen observes the update. A site backs off from sharing a hot rumor when too many sites have already seen it. Now the site retains the update without any further propagation. The so called 'hot rumor' is no longer hot. In effect, only the most recent updates are sent from one replica to another replica again using unreliable datagrams. This protocol also suffers from the fact that datagrams may be lost. If the rumor cycles are too fast, there is a chance that some updates will not reach all sites [Demers88].

2.2.3 Anti-entropy

In this protocol updates originate at a single site or replica and are propagated to others. When a replica wishes to send a message, the message is timestamped with the current time which is derived from the current clock value at that site and the identity of the replica. Then, the replica writes the message to a log. The log is maintained on stable storage to survive temporary crashes or failures. Since the messages are timestamped, the term *timestamped anti-entropy* came into existence [Golding93a].

"From time to time, a replica will select another replica, and the two will exchange the contents of their message logs in an *anti-entropy session*" [Golding93a]. After the session is over, both replicas will have the same set of messages. There is no interruption when the two replicas are engaged in an anti-entropy session.

For example in one real-time environment where anti-entropy sessions have been implemented, there are a number of servers running on the Xerox Corporate Internet. Initially all updates messages are mailed and then anti-entropy sessions are run in the background, in case messages do not reach all sites. For a domain stored at 300 sites, almost 90,000 mail messages are introduced. This leads to heavy network traffic. To offset this network load, anti-entropy sessions can be used as the only device to propagate the messages without significantly increasing the load.

CHAPTER III

TIMESTAMPED ANTI-ENTROPY

3.1 Description

"The timestamped anti-entropy protocol provides reliable and eventual delivery of messages" [Golding93a]. Assume there exists a single process per site. The protocol is fault tolerant; that is, messages are delivered to all other operating process within the group, even if a process has failed.

Assumptions made include the following:

a. all sites are fully interconnected;

b. sites and processes have access to stable storage which is not corrupted if the system crashes;

c. sites have loosely synchronized clocks[Golding93a];

d. sites/processes fail by crashing; that is, they do not send invalid messages.

There are *transient failures* and *permanent failures*. When a transient failure occurs, the site goes down for a short period of time and then comes back up and joins the protocol. When a permanent failure occurs, the site is permanently removed from service. Permanent failures are fail-stop; that is, sites suffering permanent failures do not send spurious or malicious messages, but simply stop.

In order to make message exchange efficient, each replica maintains a *summary timestamp vector.* The summary timestamp vector is indexed by the identity of the replica and contains the largest timestamp the replica has received from the other replicas.

When a replica enters into an anti-entropy session with its partner, it can compare its summary timestamp vector with that of its partner. Thus, it can determine which updates have not been exchanged. Once this is determined, the replicas exchange updates using a reliable stream communication protocol. Then each partner updates its summary timestamp vector. At this point, both replicas have the same summary timestamp vector.

Consider a group of three replicas A, B and C. Replicas A and B decide to engage in an anti-entropy session. Their message logs before the anti-entropy session are shown in Figure 2 and their summary timestamp vectors are shown in Figure 3. The two replicas engage in an anti-entropy session shown in Figure 4. The summary timestamp vectors after the anti-entropy session are shown in Figure 5.

The message logs can become very large when there are a large number of replicas engaging in anti-entropy sessions. So, how to safely purge the message logs is an important issue. Each and every replica needs information on the messages that other replicas have received in order to truncate their message log safely. For replicas to know about every other replica, each replica has to explicitly acknowledge every message. This can be avoided by maintaining an *acknowledgement time vector.* Each replica maintains an acknowledgement timestamp vector. This vector is usually the same as the summary timestamp vector. This acknowledgement timestamp vector is also exchanged by a replica with its partner during the anti-entropy session. Any message in the log whose timestamp is smaller than every timestamp in the acknowledgement timestamp vector has been received and acknowledged by every replica in the group, so it can be purged.

The timestamped anti-entropy protocol can be used with best-effort multicast for efficient performance. The rationale for using best-effort multicast is that when a replica wishes to send an update message to all other replicas within the group, it can first send a multicast message. Chances are that some of the replicas will receive the update message and some will not because message delivery is not guaranteed with best-effort multicast. Now, replicas can enter into anti-entropy sessions with other replicas and propagate this update. Thus, all replicas will receive the update eventually.

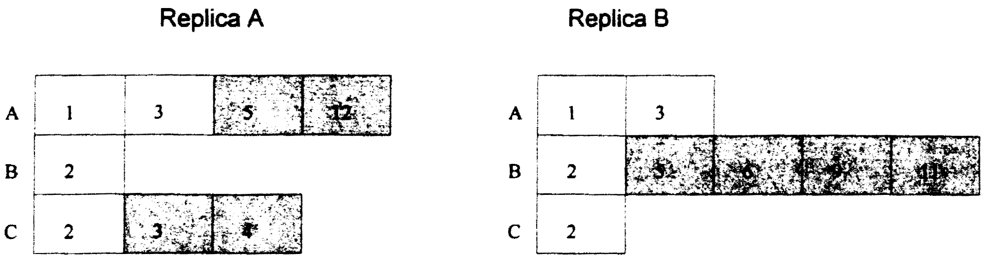

Figure 2. Message Logs for Replicas

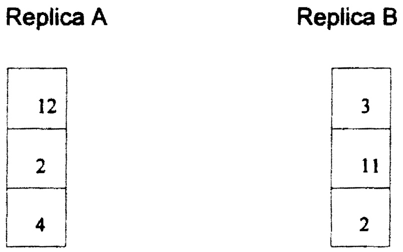

Figure 3. Summary Vectors for Replicas

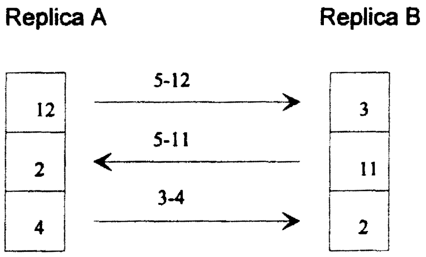

Figure 4. Anti-entropy Session

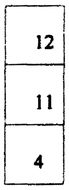

Figure 5. Summary Vector after Anti-entropy Session

3.2 Partner Selection Policies

A replica can use any one of several partner selection policies to choose a replica from the group and enter into an anti-entropy session. Partner selection is extremely important. It can affect the time required for message delivery, the degree of consistency, and the amount of network traffic caused by the protocol.

Golding classified partner selection policies into Random, Deterministic and Topological Policies[Golding93a]. We extend this classification and divide the policies into the following five classes:

a. Random Policies

b. Deterministic Policies

c. Topological Policies

d. Hierarchical Policies

e. Combination Policies

The various Partner Selection Policies are explained in the following subsections.

3.2.1 Random Policies

Each replica has a probability assigned to it. Each replica randomly selects a partner for an anti-entropy session [Golding93a].

3.2.1.a *Uniform Policy*. Each and every replica has an equal probability of being randomly selected. Once the partner replica is chosen, anti-entropy is carried out to exchange the database contents. This could lead to overloaded network links [Golding93a].

3.2.1.b *Distance Biased Policy*. Replicas that are closer have a higher probability of being selected. This policy discriminates against distant replicas. Demers et al. found that by distance biasing partner selection, network traffic on critical intercontinental links can be reduced[Demers88]. Selection policies can also consider the cost of communication or monetary costs of using a communication link.

3.2.1.c *Oldest Biased Policy*. "Replicas are selected proportional to their age in the summary timestamp vector" [Golding93a]. The replicas that are older have a lower probability of being selected for engaging in anti-entropy sessions. Update messages to any replica is propagated to another replica that has been recently updated.

3.2.2 Deterministic Policies

These policies use some fixed rule for a replica to select its partner. "State information such as a sequence counter could be used as well" [Golding93a].

3.2.2.a *Oldest First Policy*. Replicas that have not been updated for the longest time will be selected. This is determined by the oldest value in the summary timestamp vector. If there are ties, they can be broken by taking distance into consideration [Golding93b].

3.2.2.b *Latin Squares Policy*. Alon et al. proposed a technique in which a truncated Latin Square of size *n-1* x *n* is used, where every row and column has every entry just once [Alon87]. Anti-entropy sessions are divided into rounds. This policy guarantees messages to be received by all replicas in O(log *n*) time.

3.2.3 Topological Policies

These policies assign the replicas to nodes in a graph such as a ring or a mesh. Then the messages are propagated along the edges of the graph.

3.2.3.a *Ring Policy*. Replicas are organized into a ring [Golding93a]. Messages are propagated along the edges of the ring. The performance when compared to other policies is poor because at least half of the ring structure has to be traversed for an update to reach all sites.

3.2.3.b *Binary Tree Policy*. Replicas are assigned to nodes in a binary tree, and messages are propagated along the edges of the tree [Golding93a]. It takes O(log *n*) rounds for update message to reach all nodes in the tree.

3.2.3.c *Minimum Spanning Tree Policy*. We propose the following new policy. Replicas are arranged to form the nodes of a graph. Anti-entropy sessions are denoted as edges in a minimal spanning tree. The length of an edge denotes the cost associated with using the edge. This cost may be delay, time, distance, etc. The minimal spanning tree is constructed by using PRIM's algorithm[Tremblay91]. Along with propagating the update messages, while engaging in anti-entropy sessions, the minimal cost is calculated. Cost is dependant on the parameter that is to be optimized.

3.2.3.d *Mesh Policy*. Replicas are organized into a two-dimensional rectangular mesh [Golding93a]. Since this is a two-dimensional mesh all sites/replicas are fully interconnected. There is more than one path from one replica to another. Based on the criteria that is to be optimized, links can be chosen appropriately. This policy is more fault tolerant because there are redundant paths between replicas.

3.2.3.e *Hypercube Policy*. We propose the following new policy. Replicas are arranged to form the nodes of a hypercube. The nodes of a hypercube are adjacent to each other and hence the vertices can engage in anti-entropy sessions. A typical example is a hypercube with 6 sides and 8 vertices. In 3 rounds all replicas receive an update. In general O(log *n*) rounds are needed to update all replicas.

3.2.4 Hierarchical Policies

We propose the following hierarchical policy. The set of sites/replicas are divided into a hierarchy. Approaching the hierarchy from the top down, the sites can be divided into different levels. Sites at level 1 are at the top of the hierarchy, and then sites at level 2 are next in the hierarchy and so on. The sites at level 1 are connected to the most expensive links. The heuristic is to avoid using the most expensive links as much as possible, but at the same time updates must reach all sites.

An update can occur at sites belonging to either level 1 or level 2. If an update occurs at a site belonging to level 1, sites belonging to level 1 are selected first for anti-entropy sessions. Once all sites at level 1 have received the update, sites at level 2 are chosen for the update to be propagated. Sites at level 2 can propagate the updates using relatively inexpensive links. If an update occurs at a level 2 site, it must first select a site at level 1 and then propagate the update to sites at level 2. Sites at level 1 use the protocol described above to send the updates to other sites. The idea here is that level 1 sites have a higher priority over sites at level 2, and so, they must be chosen for the update to be sent over longer distances. This kind of partner selection scheme is extremely useful for reducing the load on intercontinental links.

An example to illustrate the above selection scheme is to have 3 sites at level 1 say, one in U.S.A. another in Europe, and a third one in Asia. Updates occurring at the site in U.S.A. belonging to level 1 selects another site at level 1, either from Europe or Asia. Then the site in U.S.A. can propagate the update message to sites locally within the sites at level 2. The sites in Europe and Asia also update messages to sites locally within their sites at level 2. If the update occurred at a site in Houston belonging to level 2 of the hierarchy, then this site sends its update message to a designated level 1 site within U.S.A. and that site sends the update message to the level 1 sites in Europe and Asia.

3.2.5 Combination Policies

From the above mentioned policies, numerous combinations can be envisioned by combining two or more policies. For example, we propose a random policy combined with a distance biased policy. A random policy can be used to propagate replica updates initially. A threshold value can be set which when triggered can switch the policy to a distance biased policy. The threshold value can be implemented as a counter. So, for example, this policy starts by using the random policy for propagating the update message. A counter contained in the update, is decremented. The counter is decremented every time the random policy is used. Once the counter reaches zero, policies are switched and the distance biased policy takes over.

This could give an improvement in performance, since randomly choosing sites could improve the probability that the furthest replicas get the updates quickly and then by distance biasing, the replicas that are closest in terms of distance can propagate the updates locally. In most cases replicas have a locality of reference and so it helps to use distance biasing.

CHAPTER IV

SIMULATION MODEL

4.1 Description

In order to study how quickly the updates occurring at any one site are propagated to all other sites, a Monte Carlo simulator is used [Golding93b]. The code for the Monte Carlo simulator is shown in the Appendix.

The partner selection policy used to explain the simulation is the Random Policy. The number of replicas is fixed, say n . The system starts with an update occurring at any one of the n replicas. This update has to be propagated to all other n-1 replicas using anti-entropy sessions.

Anti-entropy sessions are modelled using a Markov model. Anti-entropy sessions are considered to be a Poisson process with arrival rate λa and permanent site failures are a Poisson process with rate λf. Let f be the total number of available replicas and m be the number of available replicas that have observed a message update. The term *available* means the replica has not failed. The number of successful anti-entropy sessions is a function of m

and *f*. It also depends on the particular partner selection policy. If *f* replicas initiate anti-entropy sessions, each of these *f* replicas choose a partner based on the partner selection policy. The probability a replica that has observed an update contacts a replica that has not yet observed the update is given by (f - m)/(f - 1). The rate of useful anti-entropy is given by f((f - m)/(f -1)) λa [Golding93b]. For example, in Figure 7, the Markov model is at state <1,3>. Here m = 1 (one replica has observed an update) and f = 3 (three replicas are available). The probability that the replica that has observed the update selects a replica that has not observed the update, is given by the formula (f - m)/(f - 1) = (3 - 1)/(3 - 1) = 2/2, that is, there is a 100% probability of choosing a replica that has not received the update. The selection could be affected by the failure of the replica chosen or the failure of the replica initiating the anti-entropy session. The rate of useful anti-entropy is given by , considering the above mentioned example, 3*(2/2)*λa. This means that there is a higher probability of useful anti-entropy sessions compared with the failure rate λf.

Referring to Figure 6 or 7, each state is labelled <m,f >. A random transition probability is computed for each outgoing link. The links labeled λf are failed links. A state transition is made by selecting a transition based on a random number. This process is repeated until the system enters a terminal state. At this point, total system time is computed and checked to see if propagation of the update has succeeded or failed.

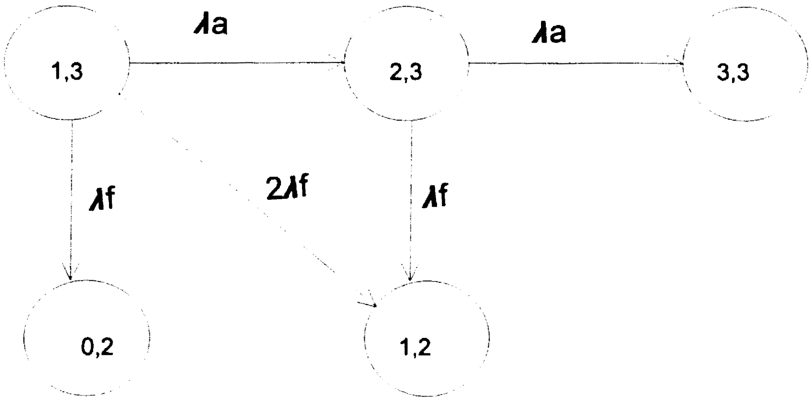

Figure 6. Rigid Policy

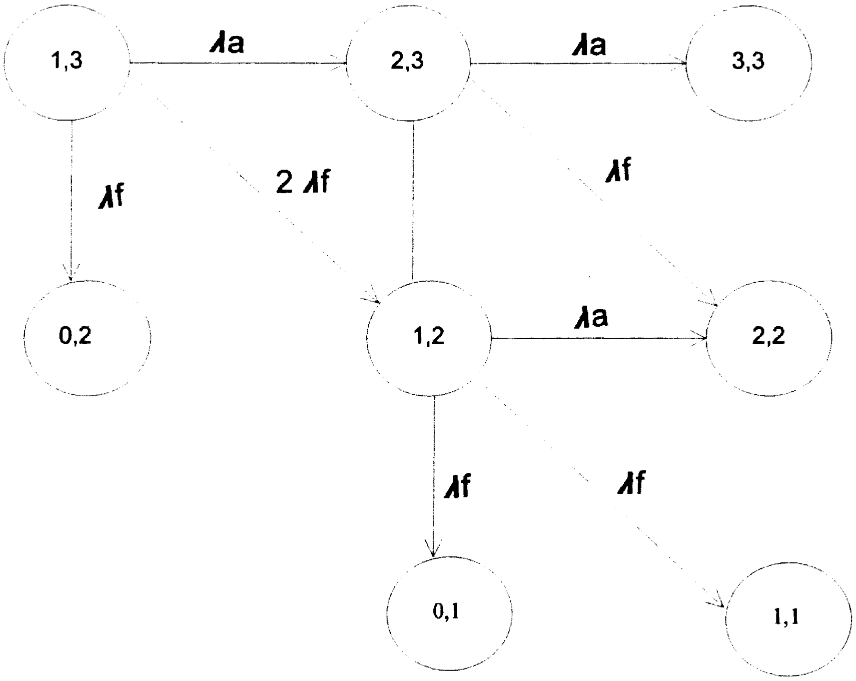

Figure 7. Flexible Policy

Referring to Figure 6, the state transitions using a Markov model for 3 replicas are shown. An update occurs at any one replica and so, 1 out of 3 replicas have the update. This is represented by the initial state <1,3>. A transition from the initial state can be made either to state <2,3> or <0,2>. If the transition is a success, then the state transition is to the state <2,3>. If the link or replica itself fails, then the update is lost and 0 out of 2 operational replicas have received the update. This is represented by the state <0,2>. If all replicas have the update, then it reaches a final state where 3 out of 3 replicas have received the update. This is called the Rigid Policy.

Referring to Figure 7, even if some of the *failed* states are reached, the update is propagated providing there are other operational replicas that have received the update. If the replica holding the update fails, then none of them observe the update (represented by the state <0,2>). At state <1,3> if the site that has been selected for an anti-entropy session fails then state <1,2> is reached. That is, the first replica can still propagate the update to the remaining operational site. Since one replica has failed,there are only two operational replicas. However, a final state can be reached even if some of the replicas have failed. If state <3,3> is reached, then all sites observe the update. This is called the Flexible Policy.

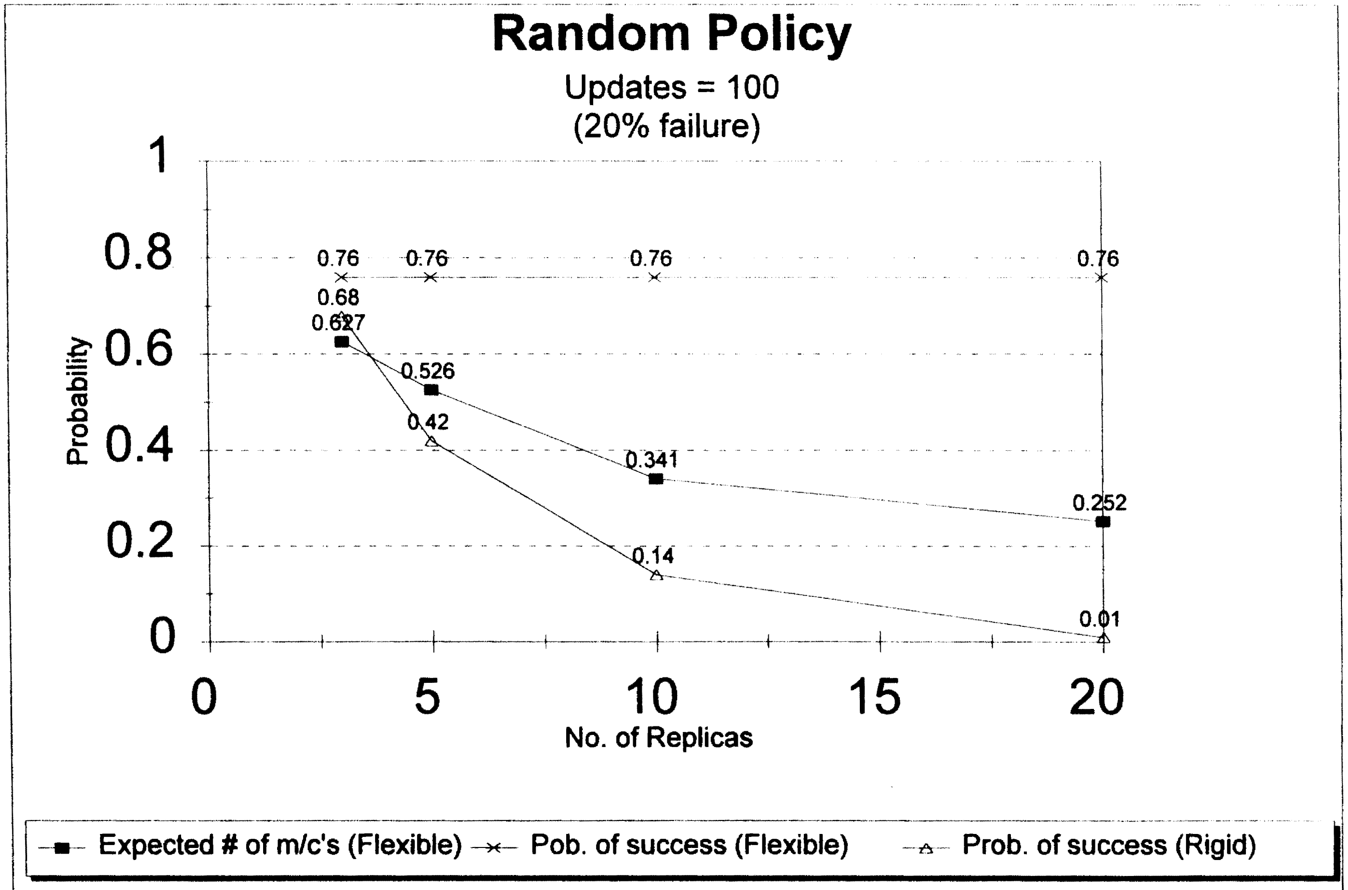

Figure 8. Random Policy (20% Failure)

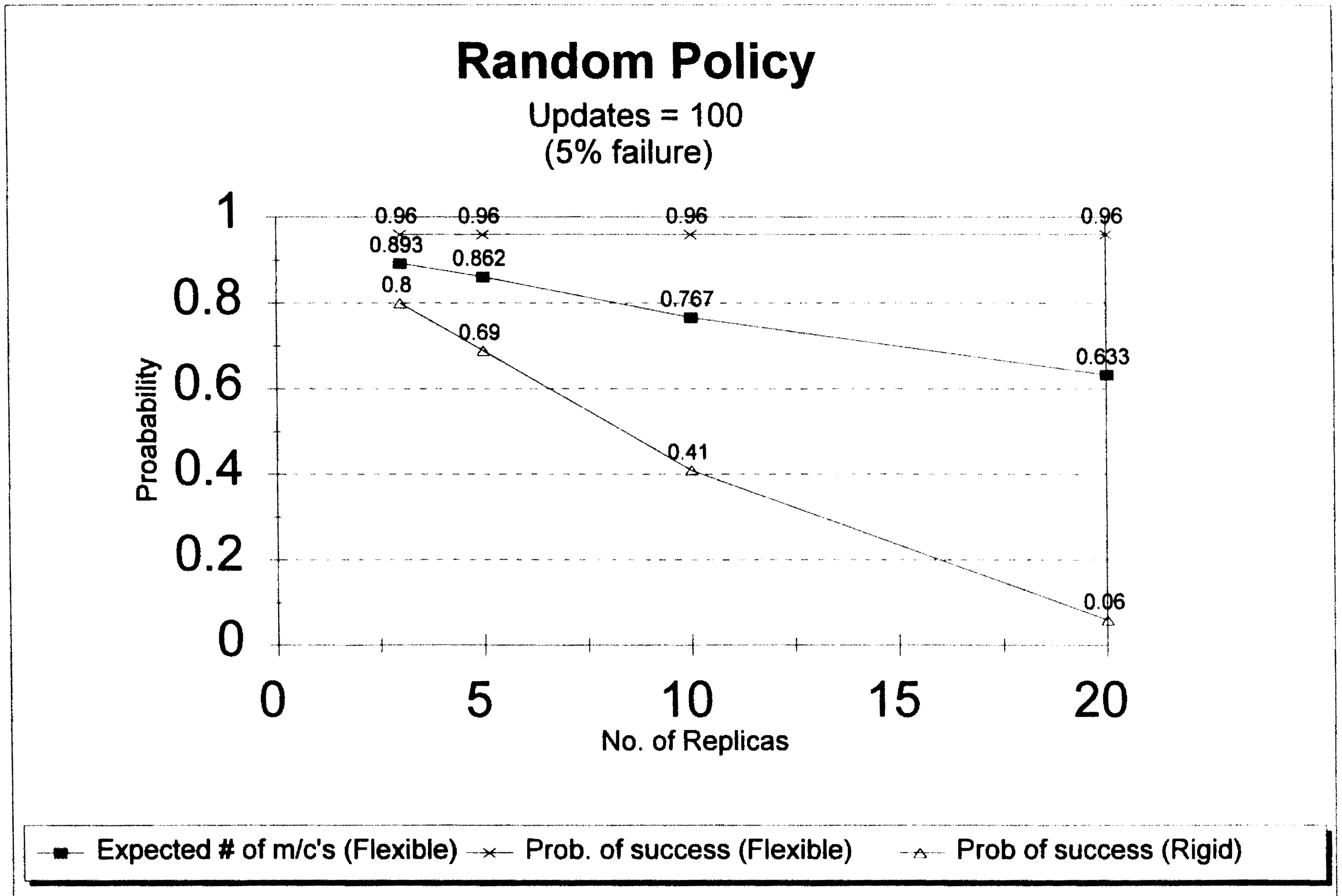

Figure 9. Random Policy (5% Failure)

Since failure is permanent, the state transition graph is acyclic. The number of states is $O(n^2)$. Thus, in an analytical solution, each state is associated with a differential equation. In general, analytical solutions are usually difficult to obtain, due to the large number of states.

4.2 Results

Success here is defined as an attempt for an update originating at any one site to be propagated to all operational sites. Figure 8 and Figure 9 show the probability of successful delivery for different numbers of replicas. Probability of success is calculated by dividing the number of successes by the total number of trials. Since the anti-entropy rate is mostly thousands of times higher than that of the permanent failure rate, the number of messages lost due to permanent failures is almost zero [Golding 93b].

Effects of temporary failure were evaluated in [Golding93b]. Temporary failures could be due to volatile storage, link failures,etc. Temporary failure is considered as a Poisson process with rate λt. Data is written to stable storage once every s time units. Probability that a failure occurs before a writeback is given by the following equation

$$P = \frac{-2e^{-s\lambda t} + s^2\lambda t^2 - 2s\lambda t + 2}{2s\lambda t^2}$$

From experimental values, for values s = 30 seconds and $1/\lambda t$ = 15 days, the probability of writeback failure is negligible. So, the expected failure rate of once every 15 days and a writeback within every 30 seconds is more than adequate to handle such failures[Golding93b]. Consequently, the analysis does not consider temporary failures.

4.2.1 Comparison and Analysis

A simulation is carried out to study the rate of message updates using the Random Selection Policy. Assumptions made are:

a. all sites are fully interconnected;

b. an update is defined as an attempt to propagate an update occurring at any one site to all other sites. An update can result in success or failure. Success means the update has been observed by all operational machines. Failure means that the update has been completely lost. The simulation was run for 100 updates. It was observed experimentally that for 100 updates steady state values were obtained;

c. in Figure 8, a 20% replica failure rate is assumed and in Figure 9, a 5% failure rate is assumed;

d. during anti-entropy sessions, failures do not occur; a site can only fail before or after an anti-entropy session;

e. results were noted for 5, 10, 20, 50, 80 and 100 replicas;

f. no temporary failures are considered;

g. distances are approximated by pinging sites on the Internet; if times between sites were unavailable, they were approximated using the Pythagorean Theorem;

h. even if only one machine does not receive the update, the update fails. This explains the disparity between the expected number of machines that are operational and the probability of success;

i. multiple updates are not considered.

Updates start at any one site as shown in the Markov model. A state change takes place if the next site or replica observes the update denoted by λa or a failure by λf.

A good random number generator was used. It returns a value between 0.0 and 1.0. For a 20% failure rate, $\lambda f = 0.20$ and $\lambda a = 0.80$. Any value between 0.0 and 0.8 indicates successful update of the replica by engaging in an anti-entropy session with that site. This is denoted by λa in the Markov model. If the random number generator returns a value between 0.8 and 1.0, it indicates a failure to update the next replica. This is represented by λf in the Markov model. Finally, transitions labelled $k.\lambda f$ represent failures of replicas which have not received the update. Using the above example, with k = 3, $k.\lambda f = 0.60$; thus, any value between 1.0 and 1.6 indicates that a replica which has not received the update has failed. Note that in this case, the number returned by the random number generator is scaled by a factor of 1.6 before being used to determine which transition to select.

In simulating the Rigid Policy, if the random number generator returns a value that falls in the *failure* interval, then the update does not reach the other replicas. Thus, the update is unsuccessful and is lost . This is represented in the Markov model in Figure 6. The probability of success versus the number of replicas ranging from 5, 10, 20, 50, 80 and 100 are plotted in Figure 8. The probability of success depreciates as the number of replicas increase. It is clear that with a small number of replicas the time to update is very short. Hence, there is a higher probability of success. As the number of replicas increases, the probability of success decreases. The expected number of machines that are still up and running at the end of the trials is the same as the probability of success.

In the Flexible Policy, shown in Figure 8, even if some of the sites fail, updates to other replicas can still take place. This is represented using the Markov model for 3 replicas shown in Figure 7. The graph includes the probability of success and the expected number of machines that are operational at the end of the trial. The expected number of machines is plotted as a percentage of the total number of machines.

In Figure 9, the probability of success is 95%; that is, it is the best possible case based on our assumption of a 5% failure rate. Some of the replicas lose the update due to machine failure. The expected number of machines that are operational is quite large, which clearly contributes to the successful delivery of message updates to the replicas. For example, for 5 replicas, 95% of the machines are expected to be up and running after the update has reached all sites. This means that on the average 4.75 out of 5 replicas receive the update.

Success means that all operational sites receive the update. Referring to Figure 9, the probability of success is quite high within our assumption limit of 5% failure rate. Increasing the number of replicas does not affect the probability of success because the updates might have been propagated by replicas to other operational sites. Increasing the number of replicas improves the chance of the update being propagated by operational sites. It is true that many sites might fail, but the time to failure rate is quite high and so it does not affect the probability of success. The expected number of sites that are operational is low because only the worst case failure is considered; that is, all failures are permanent. In reality, the expected number of sites that are operational might be quite high because many failures are transient[Golding93b].

Referring to Figure 10, for the Distance Biased Policy, the probability of success improves as the number of replicas increase until a steady value is reached. The expected number of machines drops steadily. The time taken for updates to reach all sites is quite small. This time taken is further reduced by the Minimum Spanning Tree Policy shown in Figure 18.

Referring to Figure 11, the Ring Policy gives one of the worst performances. The update message is retarded heavily by the ring structure. Any replica can only choose another replica which is adjacent in the ring. Even if only two replicas fail, the update may be lost. This contributes to the poor performance as shown by the graph. One half of the ring structure must be traversed to update all replicas. On the average at least half of the ring must be traversed for the update to reach all replicas. The time taken for the updates to propagate is extremely small because it is affected by the probability of success. Only if the various sites receive the update, is the time taken accumulated. This explains the very small time taken in Figure 18.

Referring to Figure 12, since replicas are arranged to form the nodes of a binary tree, updates are propagated along the edges of the graph. For fewer replicas, good performance is observed. As the number of replicas increase, the probability of success declines. Always, if a parent node fails, the update is not propagated further down the tree to the children. If one of the children fail then the other child node is chosen for anti-entropy sessions. Now, only if the other child also fails does the update not get propagated any further. In the end, many replicas are operational,but may not receive the update.

Referring to Figure 13, the Minimum Spanning Tree Policy is like the Distance Biased Policy, but with the added feature that the minimal path is always chosen for selecting a partner. This policy shows a reduction in time taken to update all operational sites (see Figure 18).

The Hierarchical Policy gives performance similar to the Distance Biased Policy in terms of updating various sites, but the time taken is slightly higher because long distance Intercontinental sites are chosen initially.

The Combination Policy (see Figure 15), has characteristics of the Random Policy. This is because the Random Policy is executed until the counter value reaches zero. The counter is initialized to $n/2$, where n is the number of sites.

Referring to Figure 16, a sharp dip in the graph is noticed for the probability of success. Here the assumption is that a site chooses a partner site in the same row of the two dimensional grid. So, in the same dimension the update is propagated in a linear fashion. There may be failures and so the update may not be propagated to many operational sites. But if there are a large number of replicas, then the update can be propagated to sites in the second dimension. Hence, the graph shows a steady performance as the number of sites are increased. The expected number of machines that are operational drops as there could be many sites failing.

The Hypercube Policy is hard to model. Here the assumption made is that sites have to fit the nodes in a hypercube. So, the simulation was run for

2, 4 and 8 replicas. The probability of success is extremely high. This policy shows the best performance as the number of replicas increase. Each site has more than one choice of propagating an update to another operational site. This way, even if one of the partner sites chosen for engaging in an anti-entropy session fails, there exists alternate paths for the update to be propagated to operational sites.

Referring to Figure 20, the time taken for the few number of replicas are comparable. Time taken is calculated for all attempts to propagate an update (this includes failures and successes). The Ring Policy also shows good performance for few replicas. The Random Policy takes a longer time because of the criteria used in selecting partner replicas. Depending on the random number generator, the replicas may choose expensive links and result in a marked increase in time taken to update the various replicas.

Figure 19, shows the expected number of machines that are operational after an update has been propagated to all operational sites. As expected, the Ring Policy gives poor performance. The Binary Tree Policy gives a slightly better performance than the Ring Policy. The Mesh Policy shows better performance compared to the Ring and Binary Tree Policy for fewer sites. As the number of sites increase, it gives a steady performance. The Hypercube Policy has the maximum number of operational sites.

TABLE I

COMPARISON OF VARIOUS PARTNER SELECTION POLICIES

Policy	Number receiving update	Probability of success	Time Taken
Random	Average	Constant	Average
Distance Biased	High	High	Low
Ring	Lowest	Lowest	Lowest
Minspan. Tree	High	High	Least
Hierarchical	High	High	Highest
Combination	Average	Constant	High
Mesh	Low	Average	Low
Hypercube	High	Highest	Low

4.2.2 Measures

4.2.2.a Consistency. The notion of consistency is based on the fact that any update originating at any single site is received eventually by all other sites using anti-entropy sessions. In the absence of further updates, the probability that the information has not converged decreases exponentially with time. After updating all replicas, the number of operational replicas is a good indicator of the level of consistency achieved.

4.2.2.b Total Distance. For a distance biased policy, the total distance is calculated. This is a good measure to optimize when communication links are expensive. The total distance traversed can be minimized while achieving good propagation times using a minimal spanning tree policy.

4.2.2.c Communication Latency. This measure is tied in with the total distance. Distance from link to link between nodes affects the communication latency. Communication latency is the time required for the update to be propagated to all operational sites. If long distance links are used while updating,the time to update may become longer and consequently an increase in communication latency results.

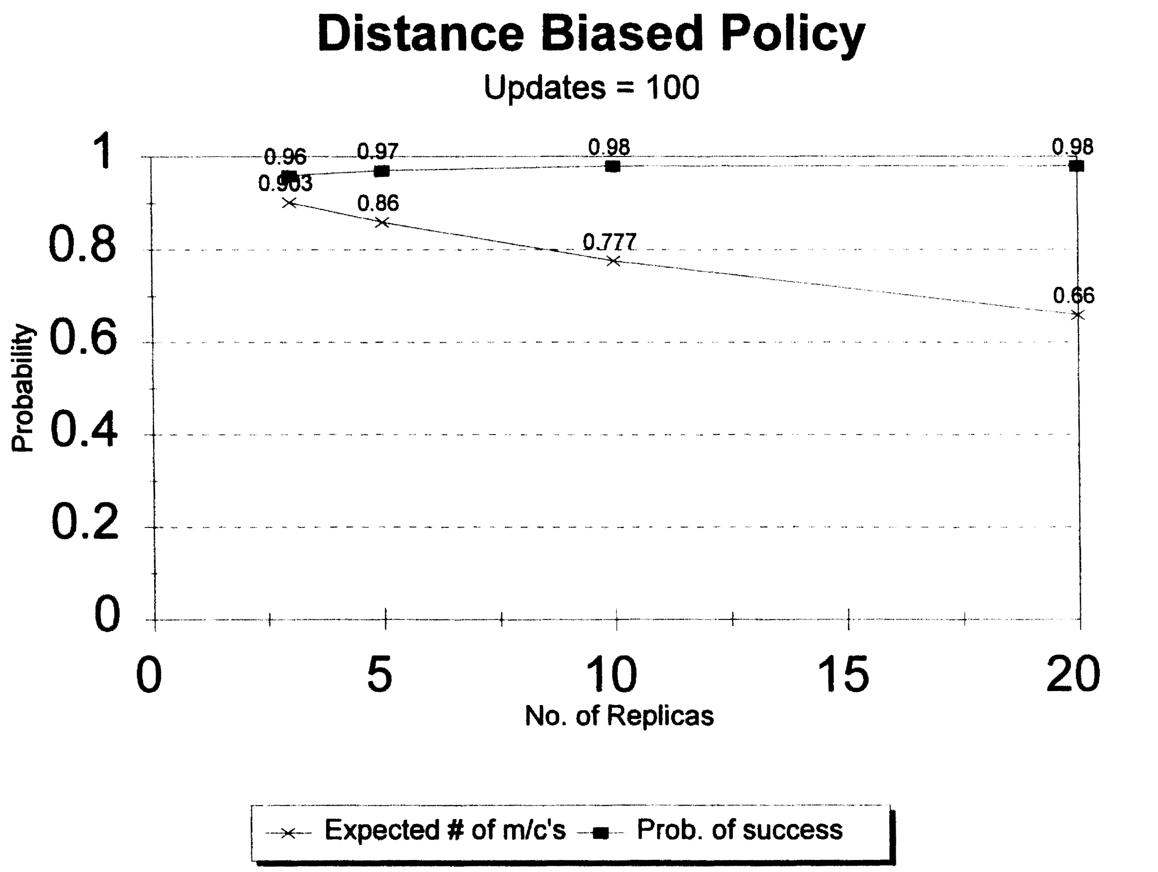

Figure 10. Distance Biased Policy

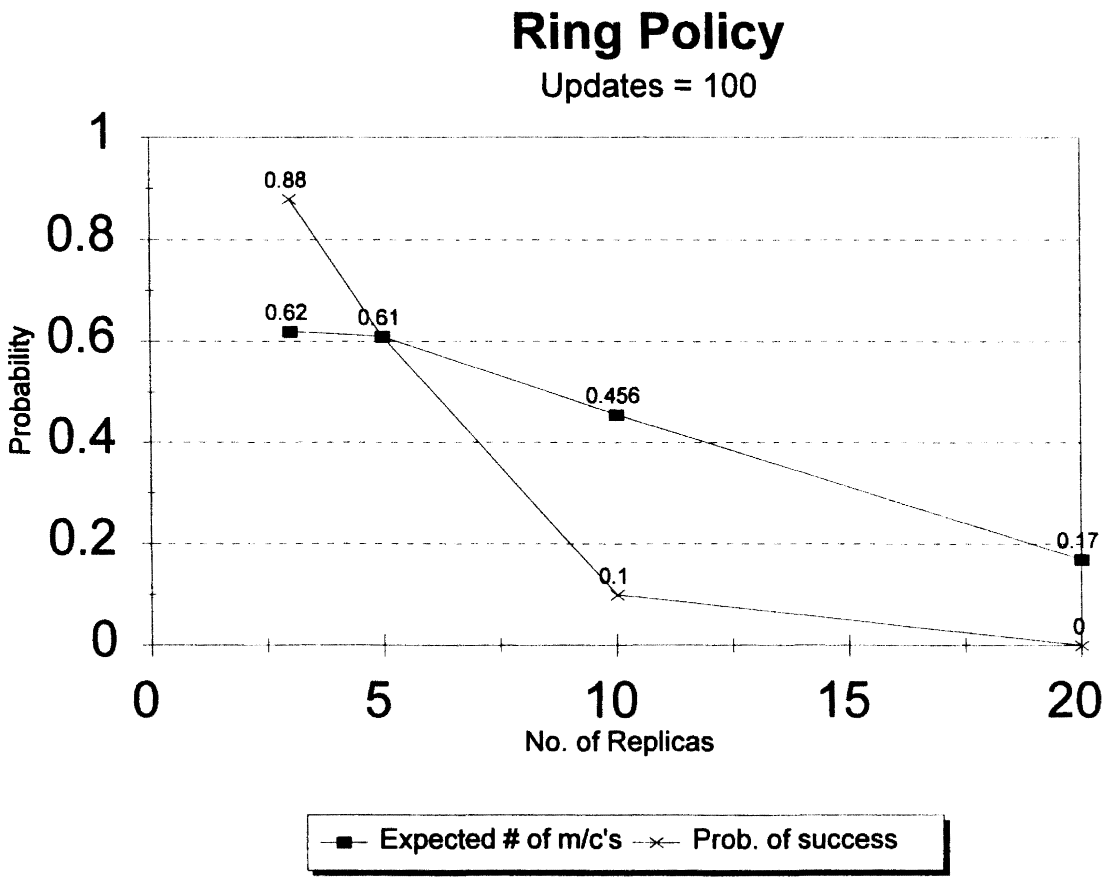

Figure 11. Ring Policy

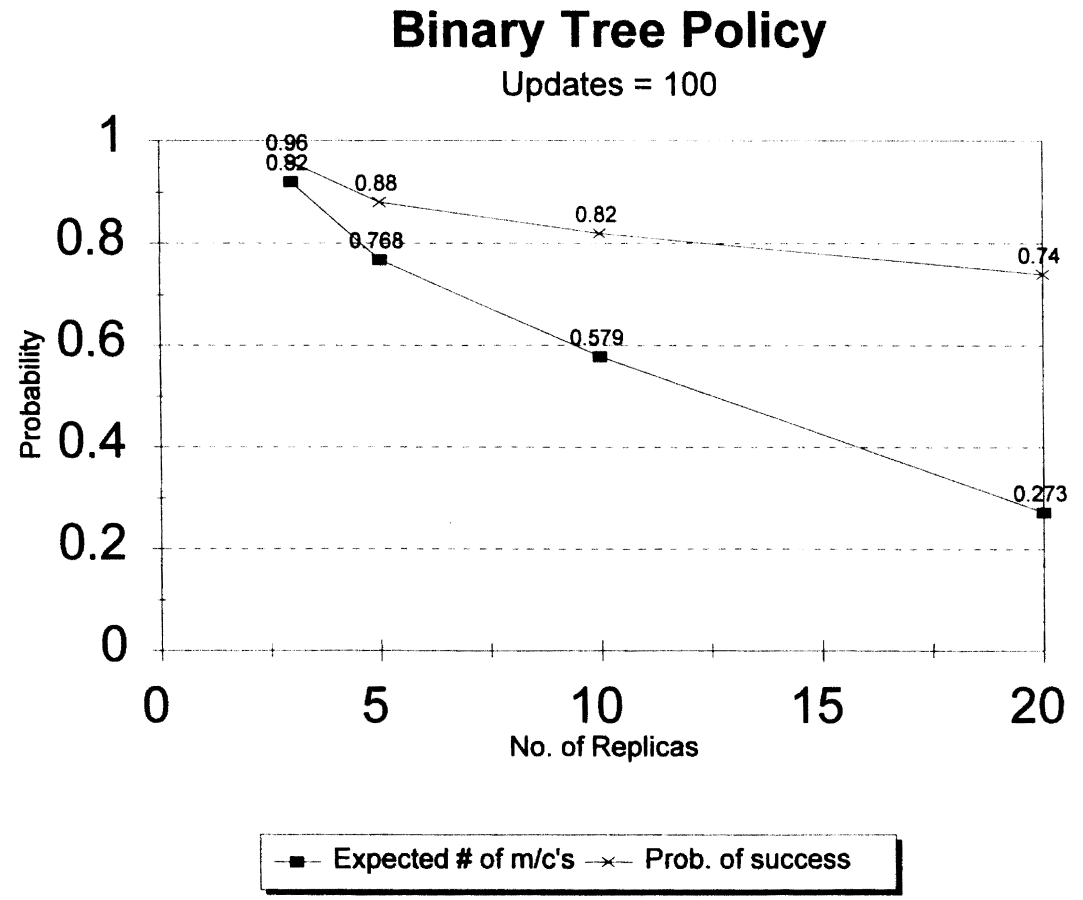

Figure 12. Binary Tree Policy

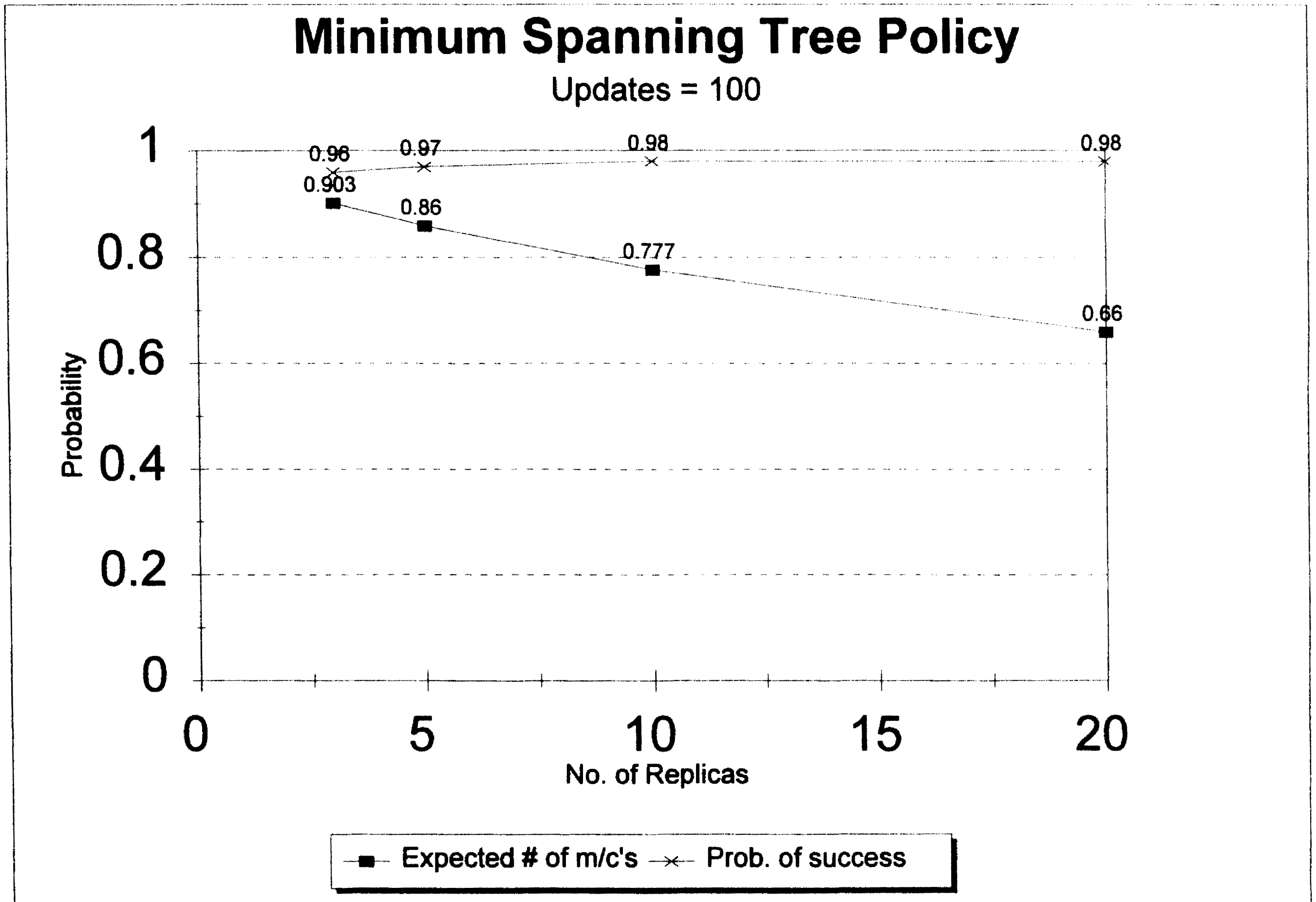

Figure 13. Minimum Spanning Tree Policy

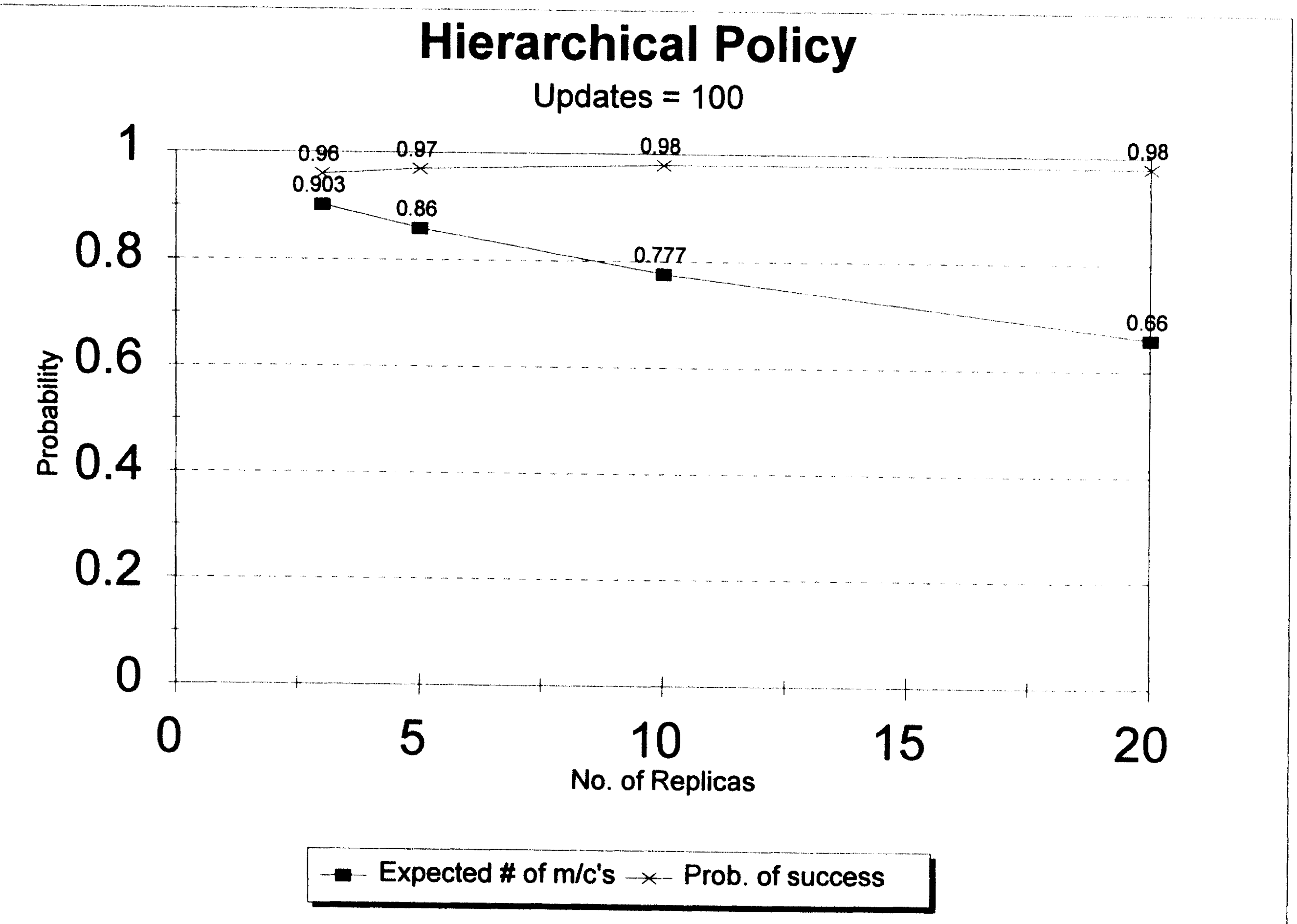

Figure 14. Hierarchical Policy

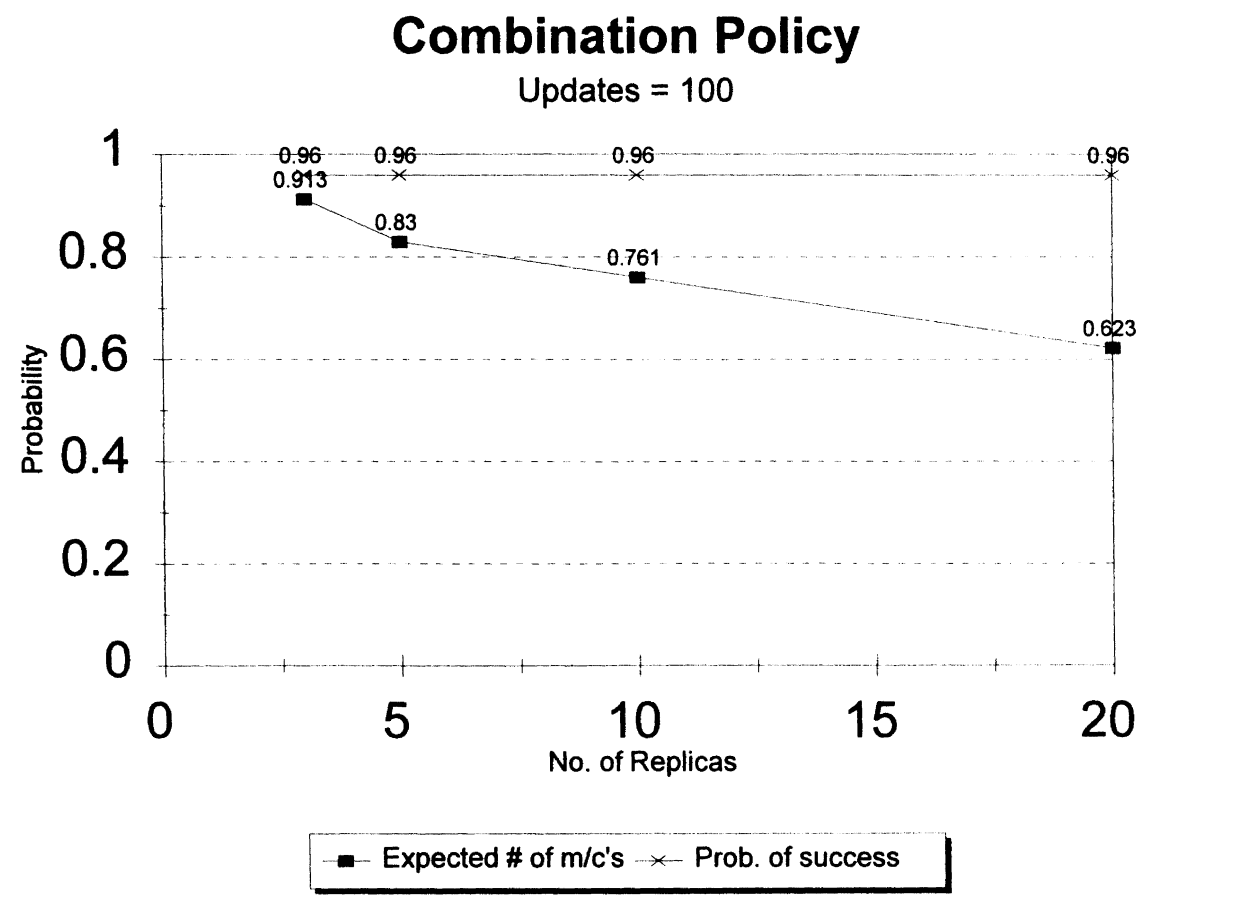

Figure 15. Combination Policy

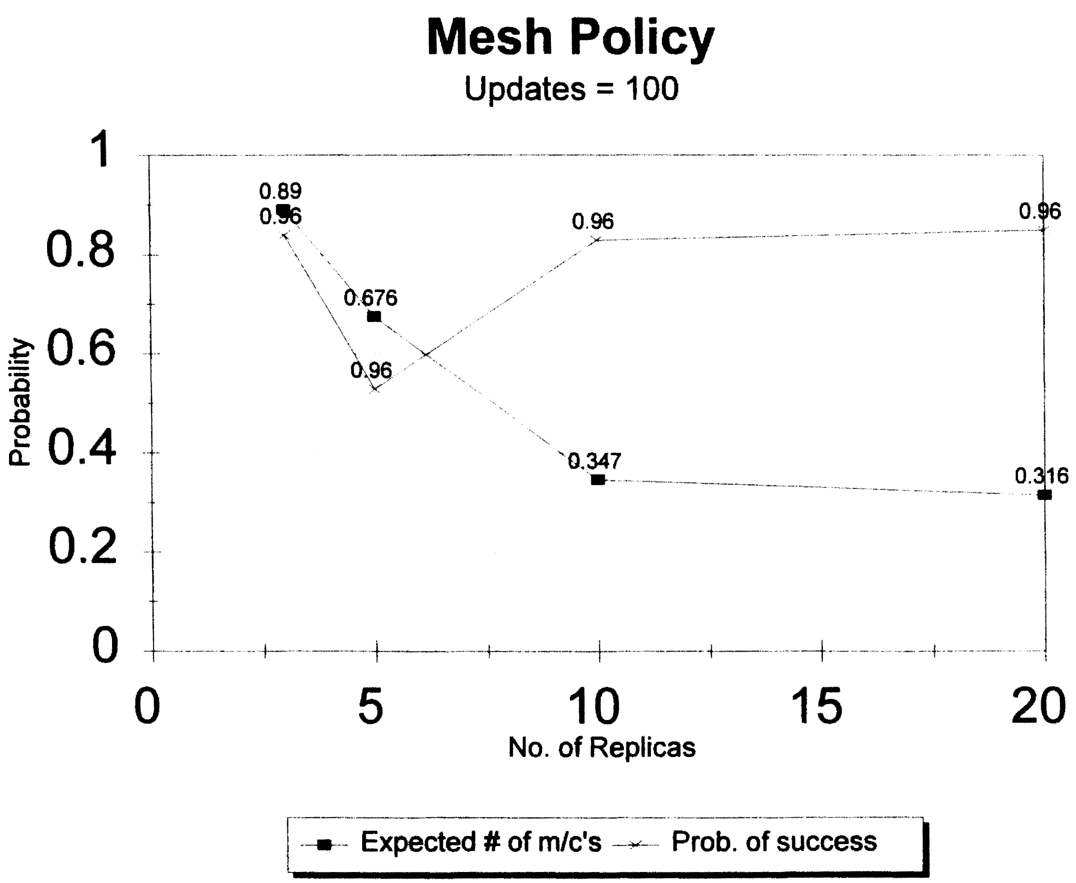

Figure 16. Mesh Policy

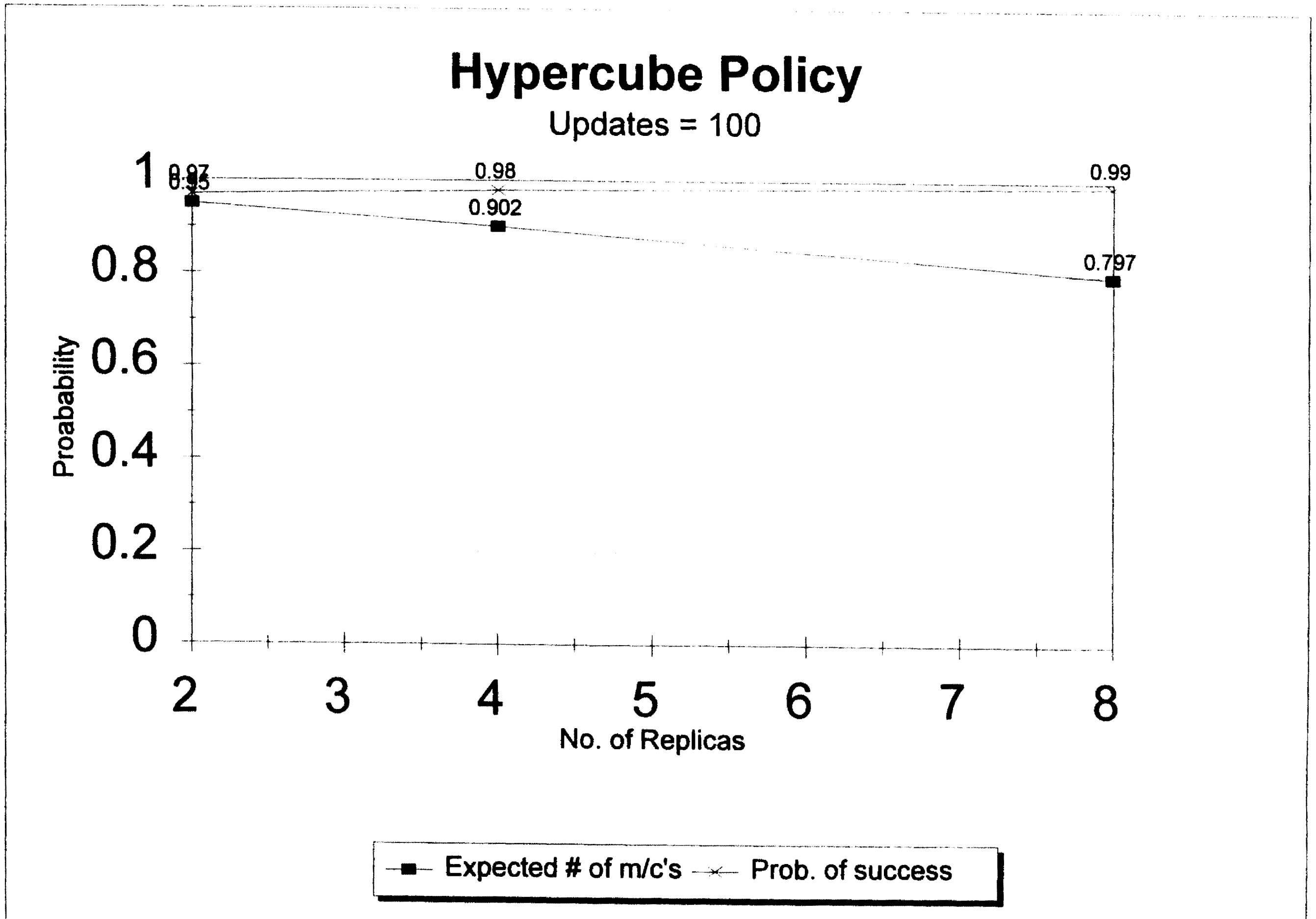

Figure 17. Hypercube Policy

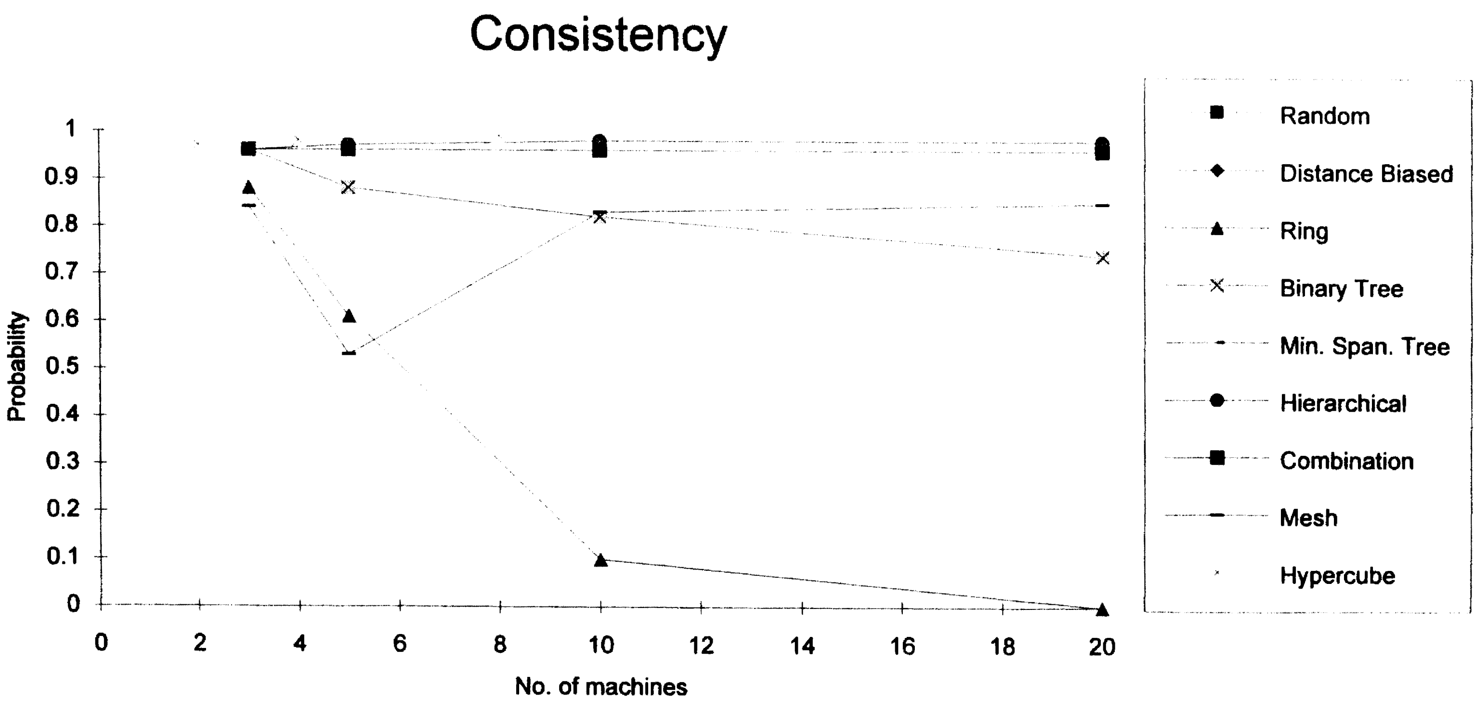

Figure 18. Consistency

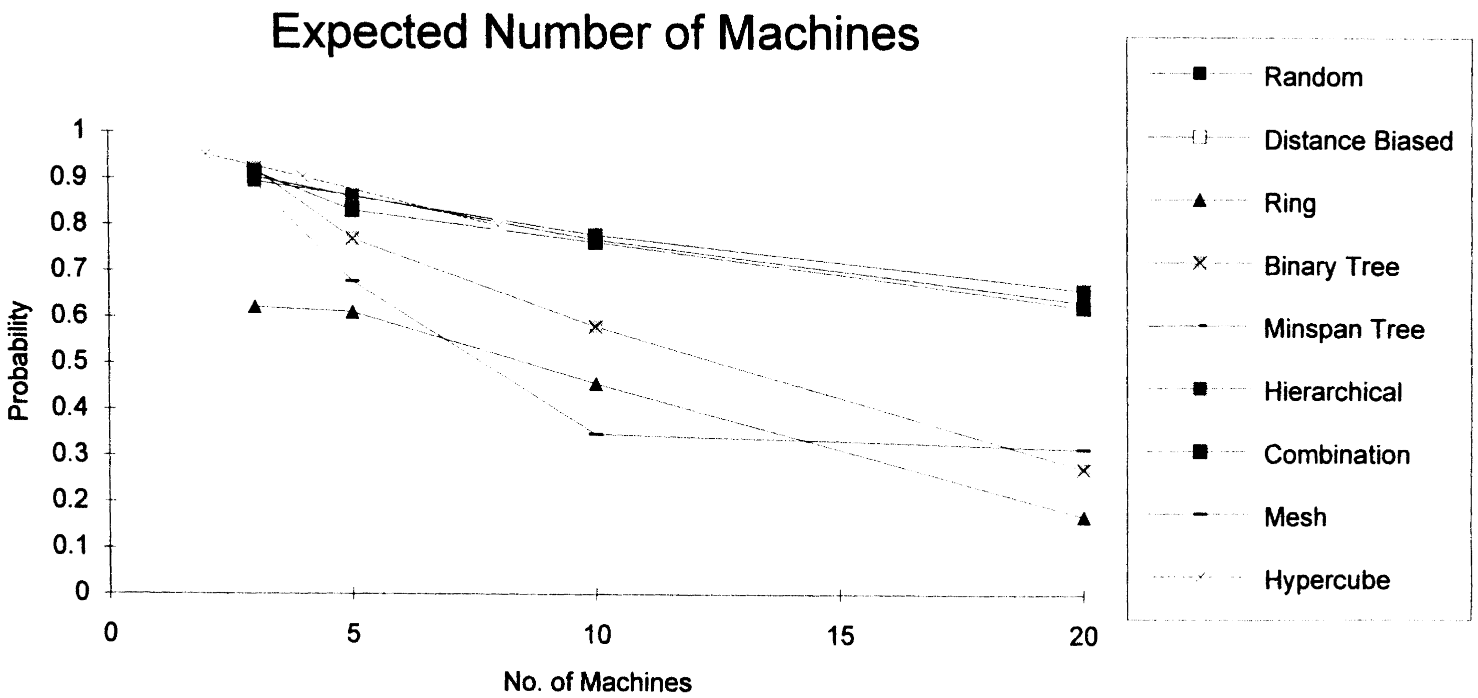

Figure 19. Expected Number of Machines

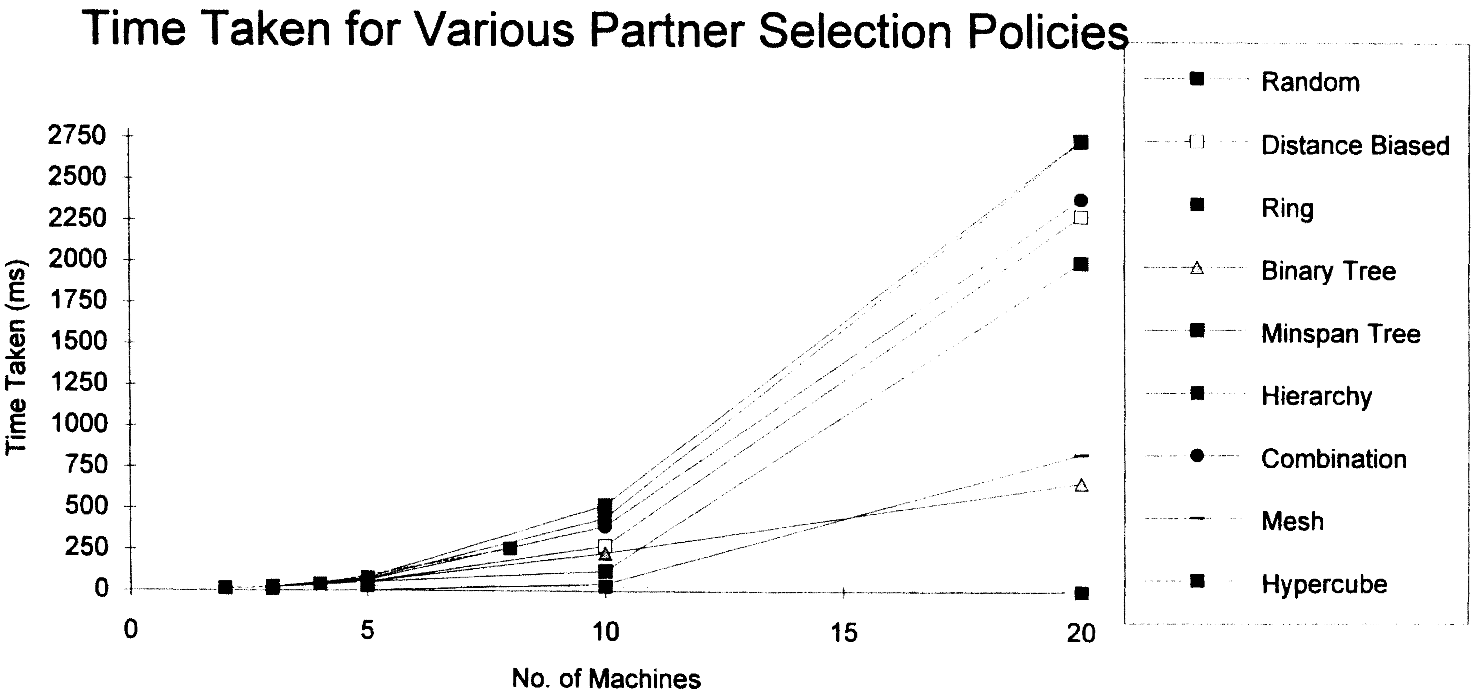

Figure 20. Time Taken

CHAPTER V

SUMMARY, CONCLUSIONS, AND FUTURE WORK

Anti-entropy protocols enforce weak consistency. Consequently, they require asynchronous communication between replicas. Furthermore, they can be used in combination with multicast protocols and can be run in the background.

In this thesis, partner selection policies for timestamped anti-entropy protocols have been thoroughly investigated. Different partner selection policies were compared using a Monte Carlo simulation. New partner selection policies were proposed and analyzed, including the following:

a. Minimum spanning tree policy

b. Hypercube policy

c. Hierarchical policy

d. Combination policy

Policies were analyzed by computing the number of machines that are still up and running after an update has been propagated. This is an indication of how the replicas are updated. The probability that an update reaches all machines is found. Some of the new measures that were proposed are:

a. Consistency

b. Total distance

c. Communication latency

Policies have been devised to establish anti-entropy sessions and propagate the messages to all sites and at the same time reduce the cost incurred. Cost could be measured in distance traveled or monetary cost. Cost in terms of total distance has been calculated.

From the results of the simulation, the Distance Biased Policy gives a steady performance. If global knowledge of the distributed environment can be obtained, the Minimum Spanning Tree Policy is appropriate. It has a high probability of success and simultaneously chooses the least expensive links for engaging in anti-entropy sessions. The Hypercube Policy has the highest probability of success provided the sites can be organized to fit the hypercube structure. The Combination Policy gives similar performance as the Random Policy. The Hierarchical Policy has the same characteristics as the Distance Biased Policy, but takes more time.

REFERENCES

[Alon87] Noga Alon, Ammon Barak, and Udi Manber. On disseminating information reliably without broadcasting. Proceedings of the 7th International Conference on Distributed Computing Systems, 74-81, 1987.

[Bernstein87] Bernstein, P. A., Hadzilacos, V., and Goodman, N. Concurrency Control and Recovery in Database Systems. Addison-Wesley Publishing Company, 1987.

[Demers88] Alan Demers et al., Epidemic algorithms for replicated database maintenance. Operating Systems Review, 22(1):8-32 (January 1988).

[Golding92] Golding, R. A., and Taylor, K. , Group Membership in the Epidemic Style. Technical Report UCSC-CRL-92-13, Computer and Information Sciences Board, University of California at Santa Cruz, 1-11, 1992.

[Golding93a] Golding, R. A., and Taylor, K. , Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, Computer and Information Sciences , University of California at Santa Cruz, 1-14, 1993.

[Golding93b] Golding, R. A., and Long, D. E., Simulation Modelling of weak consistency protocols. Proceedings of the International Workshop on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS), 233-238, January 1993.

[Lamport78] Lamport, L., Time, clocks, and the ordering of events in a distributed system. Communications of the ACM, 21(7):558-565, 1978.

[Lamport79] Lamport, L., How to make a multiprocessor computer that correctly executes multiprocess program. IEEE Transactions on Computers, C-28:690-691,1979.

[Lilienfield] Lilienfield, A.M., Foundations of Epidemiology. Oxford University Press, 1976.

[Tremblay91] Tremblay, J.P., and Sorenson, P.G., An Introduction to Data Structures with Applications. McGrawHill, 1991.

APPENDIXES

APPENDIX A

PROGRAM FOR SIMULATING 20% AND 5% FAILURE RATES

```
/* Program for Simulating 20% and 5% failure rates */
/* Global Declarations  */
#include <stdio.h>
#include <stdlib.h>                /* Header Include files */
#include <string.h>
#include <math.h>
#include <time.h>
#include <sys/types.h>
#define Lambda  1       /* Flag returned if successful */
#define Rho     2       /* Flag returned if initiating site fails */
#define Sigma   3       /* Flag returned if selected site fails */
#define Trials  100
#define STRSIZE 100
#define MAXSIZE 100        /* Constants */
#define MSHSIZE 5
#define PAIRSIZE 2
typedef struct site SITES;
typedef struct node  NODE;
typedef struct queue QUEUE;

struct site{
      float distance[MAXSIZE];
      float time[MAXSIZE];                /* Structure that contains the */
      int traversed[MAXSIZE];            /* distances between sites */
          };

struct node {
          int elem;
          NODE *next;                  /* Node structure has an element */
          };
struct queue {
          NODE *f,*r;                 /* Queue structure points to node type*/
          int n;
          };

SITES nodes[MAXSIZE],tmp[MAXSIZE];    /* Nodes/Sites Structures */
int Aray[MAXSIZE],Stck[MAXSIZE];        /* Array of unique nodes&Stack*/
int MSH[MSHSIZE][MSHSIZE];              /* Mesh 2 - dimensional array */
float rand_efficiency = 0.0;          /* Calculate the expected # of m/cs*/
double seed = 1.0;                      /* Seed value */
int top_of_stack= 0,godown = 0;        /* Pointers to top of stack */
```

```
#include "global.h"
/*  Program to engage in anti-entropy sessions using  */
/*  Rigid and Flexible Policy */

main(argc,argv)
int  argc;
char *argv[];
{
int id;
void random_flex_policy();
void uniform_rigid_police();

if(argc != 2){
  perror(" Correct format -> executable <filename>!\n");
  exit(1);
  }

/* Interactive options for choosing */
for( ; ; ){
    if(get_option(&id) == 1)     /* get the required option */
      switch(id){
          case 0:  printf(" Done simulation and quitting maan ! \n");
                 exit(1);
          case 1:  uniform_rigid_police(argv);  /* Rigid Policy */
                 break;
          case 2:  random_flex_policy(argv);     /* Flexible Policy */
                 break;
          }
      }
}

/* Function to simulate the random flexible policy */
void random_flex_policy(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_flex_policy();

null_struct();
```

```
init_struct();                              /* Initialise the structures */
read_in_data(argv,&nofsites);           /* Read in the value */
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_flex_policy(nofsites);
}

/* Driver routine  for flexible random selection */
void driver_flex_policy(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(uniform_flex_policy(nofreplicas,&time_taken)){
       success++;
       actual_time += time_taken;
       }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using random flexible technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}


/* Function to select machines randomly even if some machines fail */
uniform_flex_policy(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,x,F_Aray[MAXSIZE],nextfail=0,prev = 0;
void seed_random();
```

```
init_FAray(F_Aray);              /* Initialise the Failed sites Array */
tot_opsites = nofreplicas;       /* Initialise the total number of operational
sites*/

Push(prev);     /* Push site '0' onto the stack(site originating the update) */

while( !stop){
      seed_random();
  /* Pick any site from remaining sites */
      x = spec_rnum(1,nofreplicas-1);
 /* If next site is not in stack & not  Failed */
      if( check_in_stack(x) == 0){
          if( check_in_Aray(F_Aray,x) == 0){
              remaining_sites = tot_opsites - nofactivereplicas;
              /* Find transition */
              call_transition_find(remaining_sites,&flag);
              if( flag == Rho){            /* Site itself failed */
                  tot_opsites--;
                  if(nofactivereplicas == 1){
                      failed = 1;                        /* The update is lost */
                      nofactivereplicas--;
                      stop = 1;
                      }
                   else {
                      if(prev == top_of_stack-1)
                       /*Get a new parent site from Stack */
                          F_Aray[nextfail] = Pop();
                       else F_Aray[nextfail] = prev;
                       if(top_of_stack >0)
                           prev = Stck[top_of_stack-1];
                        nextfail++;
                        nofactivereplicas--;
                        }
                  }
              else if(flag == Sigma){      /* The partner site failed */
                      tot_opsites--;
                      F_Aray[nextfail] = x;
                      nextfail++;
                      if(nofactivereplicas == tot_opsites)
                         stop = 1;
                      }
              else if( flag == Lambda){      /* Success */
                       nofactivereplicas++;
```

```
                        *time_taken += nodes[prev].distance[x];
                        Push(x);                      /* push current site onto to stack */
                        prev = x;
                   /* Completely successful , all m/c's receive the update */
                        if( nofactivereplicas == nofreplicas)
                            stop = 1;
                   /* All the operational m/c's receive  the update */
                        else if( nofactivereplicas == tot_opsites)
                                stop = 1;
                        }
               }
           }
       }      /* Calculate expected number of machines operational */
rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed){
    return(1);
    }
else return(0);
}

/* Function to find which transtion to make */
call_transition_find(remaining_sites,flag)
int remaining_sites,*flag;
{
int loc_flag;
double new_scale;
double r_num(),val;

new_scale = 1.0+(remaining_sites * .20);
val = r_num()*new_scale;
loc_flag = prob_range(val,new_scale);
if(loc_flag == Lambda)
    *flag = Lambda;
else if(loc_flag == Rho)
    *flag = Rho;
else if(loc_flag == Sigma)
    *flag = Sigma;
}
```

```
/* Function to check if the randomly picked probability
/* lies within the assumed interval of 80 %, if 90% then change scale */
/* appropriately */
prob_range(x,new_scale)
double  x,new_scale;
{
if( 0.0 <= x  && x <= 0.80)
    return(Lambda);
else if( 0.80 < x && x <= 1.0)
         return(Rho);
else if(1.0 < x && x <= new_scale)
         return(Sigma);
}


/*Function to seed the random number generator  */
void seed_random()
{
int i;
double r_num();

for( i = 0; i < MAXSIZE; i++)
     r_num();
}



/* Function to  convert a  random number generated within a
   specific  range i.e between a high and a low value specified  */
spec_rnum(low,high)
int low,high;
{
double r_num();
int k;

k = low + (high + 1 - low)*r_num();

return(k);
}


/* Function to  get a  random number */
double r_num()
```

```
{
/* a,m,q,r     = constants   */
/* lo,hi,test = variables   */
/* seed        = seed value  */

double a = 16807.0, m = 2147483647.0,q = 127773.0,r = 2836.0;
double lo,hi,test,floor();

hi   = floor(seed/q);
lo   = seed - q*hi;
test = a*lo - r*hi;
if(test > 0.0) seed = test;
else           seed = test + m;
return(seed/m);
}

/* Function to Push an element onto the stack */
Push(elem)
int elem;
{
Stck[top_of_stack]= elem;
top_of_stack++;
}

/* Function to Pop an element off the stack */
Pop()
{
if(top_of_stack != 0){
  top_of_stack--;
  return(Stck[top_of_stack]);
  }
else {
    perror("Error\n");
    exit(0);
    }
}

/* Function to intialise the Failed aray and the Stack */
init_FAray(F)
int F[];
{
int i;
for(i = 0; i < MAXSIZE; i++){
    F[i] = -1;
    Stck[i] = -1;
    }
top_of_stack = 0;
}

/* Function to check if the  given site is in the Failed array */
check_in_Aray(F_Aray,x)
int F_Aray[],x;
{
int i = 0;

while(F_Aray[i] != -1){
      if(F_Aray[i] == x)
         return(1);
      else i++;
      }
return(0);
}

/* Function to check if the  given site is in the Stack */
check_in_stack(x)
int x;
{
int i = 0;

while(i < top_of_stack ){
      if(Stck[i] == x)
         return(1);
      else i++;
      }
return(0);
}

/* Function to intialise the distances among the sites of the network
this function also helps calculate the total distance  */
find_total_dist(nofsites,total_dist)
int nofsites;
float *total_dist;
{
int i,j;
for( i = 0; i < nofsites; i++)
    for( j = i+1; j < nofsites; j++){
            if( nodes[i].distance[j] != -1.0)
                *total_dist = *total_dist + nodes[i].distance[j];
        }
}


/* Function to put an unique element into an array */
put_into_Aray(elem)
int elem;
{

int i = 0;

while(Aray[i] != -1){
      if(elem != Aray[i])
         i++;
      else if( elem == Aray[i])
                return(0);
      }

Aray[i] = elem;
return(1);
}


/* Function to initialise the contents of the Aray */
void init_aray(nofsites)
int nofsites;
{
int i;

for(i = 0; i <= nofsites;i++)
    Aray[i] = -1;
}


/* Function to intialise the structure  */
void null_struct()
{
memset(&nodes,NULL,sizeof(SITES));
memset(&tmp,NULL,sizeof(SITES));
}
```

```
/* Function to initialise the structure with -1 values */
void init_struct()
{
int i,j;

for( i = 0; i < MAXSIZE; i++)
    for( j = 0; j < MAXSIZE; j++){
        nodes[i].distance[j] = -1.0;
        nodes[i].time[j] = -1.0;
        nodes[i].traversed[j] = 0;
        tmp[i].traversed[j] = 0;
        }
}


/* Function to read in the data  */
read_in_data(argv,nofsites)
char *argv[];
int *nofsites;
{
int i = 0,j = 0;
float val = 0.0;
FILE *fp;


if((fp = fopen(argv[1],"r")) == NULL){
    perror(" Error in opening of input file !\n");
    exit(1);
   }
fscanf(fp,"%d\n",nofsites);
if(*nofsites == 0)
  perror(" Cannot continue , please check input file !\n");

while(!feof(fp)){
      fscanf(fp,"%d %d %f\n",&i,&j,&val);
      nodes[i].distance[j] = nodes[j].distance[i] = val;
      tmp[i].distance[j] = tmp[j].distance[i] = val;
      }
nodes[0].distance[0] = -1.0;
tmp[0].distance[0] = -1.0;
fclose(fp);
}
```

```
/* Function that uses pythogoras' theorem to find out distances for
   unknown sites */
void init_vals(nofsites)
int nofsites;
{
int i,j;
float res;

for( i = 1; i < nofsites; i++)
    for( j = i+1; j < nofsites; j++)
        if( nodes[i].distance[j] == -1.0){
           if( nodes[0].distance[j] > nodes[0].distance[i])
              res = pow(nodes[0].distance[j],2) - pow(nodes[0].distance[i],2);
           else {
               res = pow(nodes[0].distance[i],2) + pow(nodes[0].distance[j],2);
               }
           tmp[i].distance[j] = tmp[j].distance[i] = nodes[i].distance[j] =
nodes[j].distance[i] = sqrt(res);
           }
}


/* Function to copy back the values to the structure */
void copy_back(nofsites)
int nofsites;
{
int i,j;

/* Initialise the structure with the values */
for( i = 0; i < nofsites; i++)
    for( j = 0; j < nofsites; j++){
        nodes[i].distance[j] = tmp[i].distance[j];
        nodes[i].traversed[j] = 0;
        }
}

/* Function to simulate the uniform rigid policy */
void uniform_rigid_police(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_uni_pol();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_uni_pol(nofsites);
}

/* Driver routine  for rigid random selection */
void driver_uni_pol(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(uniform_rigid_pol(nofreplicas,&time_taken)){
       success++;
       actual_time += time_taken;
       }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using random rigid technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}



/* Function to simulate the rigid policy */
```

```
uniform_rigid_pol(nofreplicas,time_taken)
int nofreplicas,*time_taken;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,x,F_Aray[MAXSIZE],nextfail=0,prev = 0;
double any_no;
void seed_random();

init_FAray(F_Aray);
tot_opsites = nofreplicas;

Push(prev);     /* Push site '0' onto the stack(site originating the update) */

while( !stop){
    seed_random();
    x = spec_rnum(1,nofreplicas-1);
    if( check_in_stack(x) == 0){
         remaining_sites = tot_opsites - nofactivereplicas;
         call_transition_find(remaining_sites,&flag);
         if( flag == Rho){
           tot_opsites--;
           failed = 1;
           nofactivereplicas--;
           stop = 1;
           }
         else if(flag == Sigma){
              tot_opsites--;
              failed = 1;
              stop = 1;
              }
         else if( flag == Lambda){
               nofactivereplicas++;
               *time_taken += nodes[prev].distance[x];
               Push(x);
               prev = x;
           /* Completely successful , all m/c's receive the update */
               if( nofactivereplicas == nofreplicas)
                 stop = 1;
           /* All the operational m/c's receive  the update */
               else if( nofactivereplicas == tot_opsites)
                    stop = 1;
               }
      }
      }
rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
   return(1);
else return(0);
}


/* Function to get the options required */
get_option(id)
int *id;
{
print_options();
scanf("%d",id);
if(*id > 12)
   *id = 0;
return(1);
}


/* Function to print the Input Options  */
print_options()
{
printf("\n");
printf("%-53s\n","Press option 1 for simulating Random Rigid policy");
printf("%-53s\n","Press option 2 for simulating Random Flexible policy");
printf("%-28s\n","Press option 0 for quitting");
}
```

APPENDIX B

PROGRAM FOR SIMULATING VARIOUS PARTNER SELECTION POLICIES

```
#include "global.h"
/*  Program to engage in anti-entropy sessions using  */
/*  Various Partner Selection Policies   */
main(argc,argv)
int  argc;
char *argv[];
{
int id;
void random_flex_policy();
void distance_bias();
void ring_policy();
void binary_tree();
void minspan();
void mesh();
void hypercube();
void hierarchy();
void combo();

if(argc != 2){
  perror(" Correct format -> executable <filename>!\n");
  exit(1);
  }

for( ; ; ){
   if(get_option(&id) == 1)    /* get the required option */
     switch(id){
         case 0:  printf(" Done simulation and quitting maan ! \n");
                  exit(1);
         case 1:  random_flex_policy(argv);
                  break;
         case 2:  distance_bias(argv);
                  break;
         case 6:  ring_policy(argv);
                  break;
         case 7:  binary_tree(argv);
                  break;
         case 8:  minspan(argv);
                  break;
         case 9:  mesh(argv);
                  break;
         case 10: hypercube(argv);
                  break;
         case 11: hierarchy(argv);
```

```
                break;
        case 12: combo(argv);
                break;
        }
    }
}

/* Function to simulate the random flexible policy */
void random_flex_policy(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_flex_policy();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_flex_policy(nofsites);
}

/* Driver routine  for flexible random selection */
void driver_flex_policy(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(uniform_flex_policy(nofreplicas,&time_taken)){
      success++;
      actual_time += time_taken;
```

```
        }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using random flexible technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}


/* Function to select machines randomly even if some machines fail */
uniform_flex_policy(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,x,F_Aray[MAXSIZE],nextfail=0,prev = 0;
double any_no;
void seed_random();

init_FAray(F_Aray);
tot_opsites = nofreplicas;

Push(prev);     /* Push site '0' onto the stack(site originating the update) */
while( !stop){
      seed_random();
      x = spec_rnum(1,nofreplicas-1);
      if( check_in_stack(x) == 0){
          if( check_in_Aray(F_Aray,x) == 0){
              remaining_sites = tot_opsites - nofactivereplicas;
              call_transition_find(remaining_sites,&flag);
              if( flag == Rho){
                  tot_opsites--;
                  if(nofactivereplicas == 1){
                      failed = 1;
                      nofactivereplicas--;
                      stop = 1;
                      }
                   else {
                      if(prev == top_of_stack-1)
                          F_Aray[nextfail] = Pop();
                       else F_Aray[nextfail] = prev;
```

```
                         if(top_of_stack >0)
                             prev = Stck[top_of_stack-1];
                          nextfail++;
                          nofactivereplicas--;
                          }
                   }
                else if(flag == Sigma){
                        tot_opsites--;
                        F_Aray[nextfail] = x;
                        nextfail++;
                        if(nofactivereplicas == tot_opsites)
                           stop = 1;
                        }
                else if( flag == Lambda){
                         nofactivereplicas++;
                         *time_taken += nodes[prev].distance[x];
                         Push(x);
                         prev = x;
                    /* Completely successful , all m/c's receive the update */
                         if( nofactivereplicas == nofreplicas)
                             stop = 1;
                    /* All the operational m/c's receive  the update */
                         else if( nofactivereplicas == tot_opsites)
                                  stop = 1;
                         }
               }
           }
       }
rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed){
    return(1);
    }
else return(0);
}

/* Function to find which transtion to make */
call_transition_find(remaining_sites,flag)
int remaining_sites,*flag;
{
int loc_flag;
double new_scale;
double r_num(),val;
```

```
new_scale = 1.0+(remaining_sites * .05);
val = r_num()*new_scale;
loc_flag = prob_range(val,new_scale);
if(loc_flag == Lambda)
    *flag = Lambda;
else if(loc_flag == Rho)
    *flag = Rho;
else if(loc_flag == Sigma)
    *flag = Sigma;
}

/* Function to check if the randomly picked probability
   lies within the assumed interval */
prob_range(x,new_scale)
double  x,new_scale;
{
if( 0.0 <= x  && x <= 0.95)
    return(Lambda);
else if( 0.95 < x && x <= 1.0)
         return(Rho);
else if(1.0 < x && x <= new_scale)
         return(Sigma);
}


/*Function to seed the random number generator  */
void seed_random()
{
int i;
double r_num();

for( i = 0; i < MAXSIZE; i++)
    r_num();
}


/* Function to  convert a  random number generated within a
   specific  range i.e between a high and a low value specified  */
spec_rnum(low,high)
int low,high;
{
double r_num();
int k;
```

```
k = low + (high + 1 - low)*r_num();

return(k);
}


/* Function to  get a  random number */
double r_num()
{
/* a,m,q,r     = constants   */
/* lo,hi,test = variables   */
/* seed        = seed value  */

double a = 16807.0, m = 2147483647.0,q = 127773.0,r = 2836.0;
double lo,hi,test,floor();

hi   = floor(seed/q);
lo   = seed - q*hi;
test = a*lo - r*hi;
if(test > 0.0) seed = test;
else           seed = test + m;
return(seed/m);
}

/* Function to Push an element onto the stack */
Push(elem)
int elem;
{
Stck[top_of_stack]= elem;
top_of_stack++;
godown++;
}

/* Function to Pop an element off the stack */
Pop()
{
if(top_of_stack != 0){
  top_of_stack--;
  godown--;
  return(Stck[top_of_stack]);
  }
else {
    perror("Error\n");
    exit(0);
    }
}


/* Function to intialise the Failed aray and the Stack */
init_FAray(F)
int F[];
{
int i;
for(i = 0; i < MAXSIZE; i++){
   F[i] = -1;
   Stck[i] = -1;
   }
top_of_stack = 0;
godown = 0;
}

/* Function to check if the  given site is in the Failed array */
check_in_Aray(F_Aray,x)
int F_Aray[],x;
{
int i = 0;

while(F_Aray[i] != -1){
     if(F_Aray[i] == x)
        return(1);
     else i++;
     }
return(0);
}

/* Function to check if the  given site is in the Stack */
check_in_stack(x)
int x;
{
int i = 0;

while(i < top_of_stack ){
     if(Stck[i] == x)
        return(1);
     else i++;
     }
```

```
return(0);
}

/* Function to intialise the distances among the sites of the network
this function also helps calculate the total distance  */
find_total_dist(nofsites,total_dist)
int nofsites;
float *total_dist;
{
int i,j;
for( i = 0; i < nofsites; i++)
    for( j = i+1; j < nofsites; j++){
            if( nodes[i].distance[j] != -1.0)
               *total_dist = *total_dist + nodes[i].distance[j];
        }
}

/* Function to initialise the contents of the Aray */
void init_aray(nofsites)
int nofsites;
{
int i;

for(i = 0; i <= nofsites;i++)
   Aray[i] = -1;
}

/* Function to intialise the structure  */
void null_struct()
{
memset(&nodes,NULL,sizeof(SITES));
memset(&tmp,NULL,sizeof(SITES));
}

/* Function to initialise the structure with -1 values */
void init_struct()
{
int i,j;

for( i = 0; i < MAXSIZE; i++)
    for( j = 0; j < MAXSIZE; j++){
        nodes[i].distance[j] = -1.0;
        nodes[i].time[j] = -1.0;
```

```
          nodes[i].traversed[j] = 0;
          tmp[i].traversed[j] = 0;
          }
}


/* Function to read in the data  */
read_in_data(argv,nofsites)
char *argv[];
int *nofsites;
{
int i = 0,j = 0;
float val = 0.0;
FILE *fp;

if((fp = fopen(argv[1],"r")) == NULL){
    perror(" Error in opening of input file !\n");
    exit(1);
   }
fscanf(fp,"%d\n",nofsites);
if(*nofsites == 0)
  perror(" Cannot continue , please check input file !\n");

while(!feof(fp)){
      fscanf(fp,"%d %d %f\n",&i,&j,&val);
      nodes[i].distance[j] = nodes[j].distance[i] = val;
      tmp[i].distance[j] = tmp[j].distance[i] = val;
      }
nodes[0].distance[0] = -1.0;
tmp[0].distance[0] = -1.0;
fclose(fp);
}


/* Function that uses pythogoras' theorem to find out distances for
  unknown sites */
void init_vals(nofsites)
int nofsites;
{
int i,j;
float res;

for( i = 1; i < nofsites; i++)
```

```
    for( j = i+1; j < nofsites; j++)
        if( nodes[i].distance[j] == -1.0){
            if( nodes[0].distance[j] > nodes[0].distance[i])
                res = pow(nodes[0].distance[j],2) - pow(nodes[0].distance[i],2);
            else {
                res = pow(nodes[0].distance[i],2) + pow(nodes[0].distance[j],2);
                }
            tmp[i].distance[j] = tmp[j].distance[i] = nodes[i].distance[j] =
nodes[j].distance[i] = sqrt(res);
            }
}


/* Function to copy back the values to the structure */
void copy_back(nofsites)
int nofsites;
{
int i,j;

for( i = 0; i < nofsites; i++)
    for( j = 0; j < nofsites; j++){
        nodes[i].distance[j] = tmp[i].distance[j];
        nodes[i].traversed[j] = 0;
        }
}

/* Function to simulate the distance biased policy */
void distance_bias(argv)
char *argv[];
{

int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_distance_policy();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);
```

```
find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_distance_policy(nofsites);
}

/* Driver routine  for distance biased policy */
void driver_distance_policy(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();
void copy_back();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(i!=0)
       copy_back(nofreplicas);
    if(run_dist_pol(nofreplicas,&time_taken)){
       success++;
       actual_time += time_taken;
       }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using distance-bias technique =
%.3f\n\n",(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  =
%.3fms\n\n",actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function to run the distance biasing */
run_dist_pol(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,next_replica,F_Aray[MAXSIZE],nextfail=0,prev = 0;
float min;

init_FAray(F_Aray);
tot_opsites = nofreplicas;
```

```
Push(prev);     /* Push site '0' onto the stack(site originating the update) */

while( !stop){
    min = 9999.0;
    if(find_min_site(nofreplicas,&min,&next_replica,prev)){
      if( check_in_stack(next_replica) == 0){
        if( check_in_Aray(F_Aray,next_replica) == 0){
          remaining_sites = tot_opsites - nofactivereplicas;
          call_transition_find(remaining_sites,&flag);
          if( flag == Rho){
            tot_opsites--;
            if( nofactivereplicas == 1){
              failed = 1;
              nofactivereplicas--;
              stop = 1;
              }
             else {
                if(prev == top_of_stack-1)
                  F_Aray[nextfail] = Pop();
                else F_Aray[nextfail] = prev;
                if( top_of_stack >0)
                   prev = Stck[top_of_stack-1];
                nextfail++;
                nofactivereplicas--;
                }
            }
           else if(flag == Sigma){
               tot_opsites--;
               F_Aray[nextfail] = next_replica;
               nextfail++;
               if(nofactivereplicas == tot_opsites)
                 stop = 1;
               }
           else if( flag == Lambda){
                nofactivereplicas++;
               *time_taken += nodes[prev].distance[next_replica];

nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
                Push(next_replica);
                prev = next_replica;
                if( nofactivereplicas == nofreplicas)
                   stop = 1;
                else if( nofactivereplicas == tot_opsites)
```

```
                                     stop = 1;
                            }
                 }
                 else nodes[prev].distance[next_replica] =
nodes[next_replica].distance[prev] = 0.0;
             }
             else
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
        }
        else stop = 1;
     }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
    return(1);
else return(0);
}

/* Function to find the minimum distance from the given start node */
find_min_site(nofsites,min,next_replica,start)
int nofsites,*next_replica,start;
float *min;
{
int j,flag = 0;


for( j = 0; j < nofsites; j++){
     if( (nodes[start].distance[j] != 0.0) && (nodes[start].distance[j] != -1.0) &&
(nodes[start].traversed[j] != 1)){
          if( !flag){
              *min = nodes[start].distance[j];
              *next_replica = j;
              flag = 1;
              }
           else if( nodes[start].distance[j] < *min){
                    *min = nodes[start].distance[j];
                    *next_replica = j;
                    }
           }
     }

if(*min == 9999.0)
    return(0);
```

```
else return(1);
}

/* Function to find the maximum distance from the given start node */
find_max_site(nofsites,max,next_replica,start)
int nofsites,*next_replica,start;
float *max;
{
int i , j,flag = 0;

for( j = 0; j < nofsites; j++){
    if( (nodes[start].distance[j] != 0.0) && (nodes[start].distance[j] != -1.0) &&
(nodes[start].traversed[j] != 1)){
         if( !flag){
             *max = nodes[start].distance[j];
             *next_replica = j;
             flag = 1;
             }
          else if( nodes[start].distance[j] > *max){
                   *max = nodes[i].distance[j];
                   *next_replica = j;
                   }
          }
    }

if(*max == 0.0)
   return(0);
else return(1);
}

/* Function to simulate the Ring Topology */
void ring_policy(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void drive_ring();

null_struct();
init_struct();
```

```
read_in_data(argv,&nofsites);
init_vals(nofsites);

printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
drive_ring(nofsites);
}

/* Driver routine that simulates the Ring policy  */
void drive_ring(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();

for( i = 0; i < Trials; i++){
     time_taken = 0.0;
     init_aray(nofreplicas);
     if(run_ring_pol(nofreplicas,&time_taken)){
        success++;
        actual_time += time_taken;
        }
     }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Ring technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  =
%.3fms\n\n",actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function to run the Ring Policy */
run_ring_pol(nofreplicas,time_taken)
int nofreplicas;
float *time_taken;
{
int  i=0,nofactivereplicas = 1;
int  stop = 0,x,failed = 0,done = 1;
int  firsthalf,prev,flag,tot_opsites,remaining_sites;
```

```
tot_opsites = nofreplicas;
firsthalf = nofreplicas/2;
prev = 0;

/* Do for first half of the ring structure */
while( !stop){
       i++;
       if( i <= firsthalf){
           remaining_sites = tot_opsites - nofactivereplicas;
           call_transition_find(remaining_sites,&flag);
           if( flag == Rho){
               tot_opsites--;
               nofactivereplicas--;
               failed = 1;
               stop = 1;
               }
            else if( flag == Sigma){
                     tot_opsites--;
                     failed = 1;
                     stop = 1;
                     }
             else if( flag == Lambda){
                      nofactivereplicas++;
                      *time_taken += nodes[prev].distance[i];
                       prev = i;
                       }
           }
        else stop = 1;
       }

stop = prev = 0;
i = 1;

/* Do for second half of the ring structure */
while( !stop){
       i++;
       x = nofreplicas - i;
       if( x > firsthalf){
           remaining_sites = tot_opsites - nofactivereplicas;
           call_transition_find(remaining_sites,&flag);
           if( flag == Rho){
               tot_opsites--;
               nofactivereplicas--;
             failed = 1;
             stop = 1;
             }
          else if( flag == Sigma){
                 tot_opsites--;
                 failed = 1;
                 stop = 1;
                 }
           else if( flag == Lambda){
                  nofactivereplicas++;
                  *time_taken += nodes[prev].distance[x];
                   prev = x;
                   }
        }
      else stop = 1;
     }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
   return(1);
else return(0);
}


/* Function to initialise the Mesh 2-dimensional array */
void init_msh()
{
int i,j;

for(i = 0; i < MSHSIZE;i++)
   for(j = 0; j < MSHSIZE;j++)
      MSH[i][j] = -1;
}

/* Function to insert  the sites into the  Mesh grid */
in_vals_MSH(nofreplicas)
int nofreplicas;
{
int i = 0,j = 0,k = 0,stop = 0;

while(!stop){
     if((k < MSHSIZE) && (j < nofreplicas))
         MSH[i][k] = j;
      if( j == nofreplicas)
          stop = 1;
      else if((k == MSHSIZE) && (j < nofreplicas)){
              i++;
              k = -1;
              j--;
              }
      j++; k++;
      }
}

/* Function to simulate Mesh policy  */
void mesh(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_mesh();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_mesh(nofsites);
}

/* Driver function to call the Mesh routine */
void driver_mesh(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
int i,success = 0;
void init_aray();
void init_msh();

init_msh();
```

```
in_vals_MSH(nofreplicas);

for( i = 0; i < Trials; i++){
    time_taken = 0.0;
    init_aray(nofreplicas);
    if(run_mesh(nofreplicas,&time_taken)){
      success++;
      actual_time += time_taken;
      }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Mesh technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function that runs the Mesh policy */
run_mesh(nofreplicas,time_taken)
int nofreplicas;
float *time_taken;
{
int nofactivereplicas = 1,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,stop = 0,stop1 = 0,i,j;

tot_opsites = nofreplicas;
for( i = 0; (i < MSHSIZE) && !stop; i++){
    stop1 = 0;
    for( j = 0; (j < MSHSIZE) && !stop; j++){
        if( (i != 0) && ( j != 0)){
            remaining_sites = tot_opsites - nofactivereplicas;
            call_transition_find(remaining_sites,&flag);
            if( flag == Rho){
               tot_opsites--;
               if(nofactivereplicas == 1){
                  failed = 1;
                  nofactivereplicas--;
                  stop = stop1 = 1;
                  }
                else {
                   nofactivereplicas--;
                   stop1 = 1;
```

```
                 if( nofreplicas < MSHSIZE*(i+1)){
                   failed = 1;
                   stop = 1;
                   }
                 }
             }
          else if(flag == Sigma){
               tot_opsites--;
               stop1 = 1;
               if(nofactivereplicas == tot_opsites)
                 stop = 1;
               else if( j == 0)
                     failed = stop = 1;
               else if( nofreplicas == MSHSIZE*(i+1))
                     stop = 1;
               else if( nofreplicas < MSHSIZE*(i+1)){
                     failed = 1;
                     stop = 1;
                     }
               }
          else if( flag == Lambda){
              nofactivereplicas++;
              if(j == 0)
                *time_taken += nodes[MSH[i-1][j]].distance[MSH[i][j]];
              else *time_taken += nodes[MSH[i][j-1]].distance[MSH[i][j]];
              if( i+1 == MSHSIZE)
                *time_taken += nodes[MSH[0][0]].distance[MSH[i][j]];
               if( nofactivereplicas == nofreplicas)
                  stop = stop1 = 1;
               else if( nofactivereplicas == tot_opsites)
                     stop = stop1 = 1;;
               }
            }
         }

     }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
   return(1);
else return(0);
}
```

```
/* Function to simulate the Binary Tree policy */
void binary_tree(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_bin();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_bin(nofsites);
}


/* Driver function to call the Binary Tree  routine */
void driver_bin(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time=0.0;
int i,success = 0;
void init_aray();

for( i = 0; i < Trials; i++){
    time_taken = 0.0;
    init_aray(nofreplicas);
    if(run_binary_tree(nofreplicas,&time_taken)){
       success++;
       actual_time += time_taken;
       }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Binary Tree  technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function that runs the Binary Tree  policy */
run_binary_tree(nofreplicas,time_taken)
int nofreplicas;
float *time_taken;
{
QUEUE Q;
NODE *node,*make_node(),*DQ();
int succ1,succ2, new_succ,parent;
int nofsites = 0, failed = 0,done = 1;
int flag,stop = 0,stop1 = 0,tot_opsites,nofactivereplicas=1,remaining_sites;

init_Q(&Q);
node = make_node(0);        /* Initialise the Queues */
Enque(&Q,node);
tot_opsites = nofreplicas;
while(!stop){
     node = DQ(&Q);
     parent= extract_val_from_node(node); /* Get successors for parent*/
     succ1 = 2*parent+1;
     succ2 = 2*parent+2;
     if(succ1 < nofreplicas){
        remaining_sites = tot_opsites - nofactivereplicas;
        call_transition_find(remaining_sites,&flag);
        if( flag == Rho){
            tot_opsites--;
            nofactivereplicas--;
            stop = 1;
            failed = 1;
            }
        else if( flag == Sigma){
                tot_opsites--;
                new_succ = 2*succ1+2;
                if(new_succ < nofreplicas){
                   node = make_node(new_succ);
                   Enque(&Q,node);
                   }
                }
        else if( flag == Lambda){
                nofactivereplicas++;
            node = make_node(succ1);
            Enque(&Q,node);
            *time_taken += nodes[parent].distance[succ1];
           }
      }
    if(succ2 < nofreplicas && !stop){
      remaining_sites = tot_opsites - nofactivereplicas;
      call_transition_find(remaining_sites,&flag);
      if( flag == Rho){
         tot_opsites--;
         nofactivereplicas--;
         stop = 1;
         failed = 1;
         }
      else if( flag == Sigma){
            tot_opsites--;
            new_succ = 2*succ2+2;
            if(new_succ < nofreplicas){
              node = make_node(new_succ);
              Enque(&Q,node);
              }
            }
      else if( flag == Lambda){
            nofactivereplicas++;
            node = make_node(succ2);
            Enque(&Q,node);
            *time_taken += nodes[parent].distance[succ2];
            }
      }
    if( Is_empty_q(Q))
       stop = 1;
    }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
kill_Q(&Q);
if( done && !failed)
   return(1);
else return(0);
}
```

```
/* Function to kill the nodes of the queue */
kill_Q(tmp_Q)
QUEUE *tmp_Q;
{
NODE *ptr,*tmp;

ptr = tmp_Q->f;
while(ptr!= NULL){
      tmp = ptr;
      ptr = ptr->next;
      kill(tmp);
      }
}

/* Function to initialize the Queue */
init_Q(Q)
QUEUE *Q;
{
Q->n = 0;
Q->f = Q->r = NULL;
}

/* Function to initialize the Queue */
NODE *make_node(elem)
int elem;
{
NODE *tmp;

tmp = (NODE *)malloc(sizeof(NODE));
tmp->elem = elem;
tmp->next = NULL;
return(tmp);
}

/* Function to insert an element into the queue */
Enque(Q,node)
QUEUE *Q;
NODE *node;
{
if(Q->f == NULL){
   Q->f = Q->r = node;
   (Q->n)++;
   }
```

```
else {
    Q->r->next = node;
    Q->r = Q->r->next;
    (Q->n)++;
    }
}

/* Function to remove an item from the Queue */
NODE *DQ(Q)
QUEUE *Q;
{
NODE *tmp;

if(Q->f != NULL){
  tmp = Q->f;
  Q->f = Q->f->next;
  (Q->n)--;
  if(Q->n == 0)
     Q->f = NULL;
  return(tmp);
  }
}


/* Function to check if the Queue is empty or not */
Is_empty_q(Q)
QUEUE Q;
{
return(Q.f == NULL);
}

/* Function to extract the element in the node */
extract_val_from_node(node)
NODE *node;
{
return(node->elem);
}

/* Function to simulate the hypercube policy */
void hypercube(argv)
char *argv[];
{
int i,j,nofsites=0;
```

```
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_hypercube();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_hypercube(nofsites);
}

/* Driver function to call the Hypercube routine */
void driver_hypercube(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time;
int i,success = 0;
void init_aray();


for( i = 0; i < Trials; i++){
     init_aray(nofreplicas);
     time_taken = 0.0;
     if(i!=0)
        copy_back(nofreplicas);
     if(run_hypercube_pol(nofreplicas,&time_taken)){
        success++;
        actual_time += time_taken;
        }
     }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Hypercube technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}
```

```
/* Function to run the Hypercube policy */
run_hypercube_pol(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,next_replica,F_Aray[MAXSIZE],nextfail=0,prev = 0;
int subset;

init_FAray(F_Aray);
tot_opsites = nofreplicas;
subset = nofreplicas/2;

Push(prev);     /* Push site '0' onto the stack(site originating the update) */
while( !stop){
      if(find_succ(prev,&next_replica,nofreplicas,subset)){
         if( check_in_stack(next_replica) == 0){
            if( check_in_Aray(F_Aray,next_replica) == 0){
                remaining_sites = tot_opsites - nofactivereplicas;
                call_transition_find(remaining_sites,&flag);
                if( flag == Rho){
                    tot_opsites--;
                    if( nofactivereplicas == 1){
                        failed = 1;
                        nofactivereplicas--;
                        stop = 1;
                        }
                     else {
                          if(prev == top_of_stack-1)
                             F_Aray[nextfail] = Pop();
                          else rearrange_stck(F_Aray,&nextfail,prev);
                          if( top_of_stack > 0)
                              prev = Stck[top_of_stack-1];
                          nextfail++;
                          nofactivereplicas--;
                          }
                     }
                 else if(flag == Sigma){
                         tot_opsites--;
                         F_Aray[nextfail] = next_replica;
                         nextfail++;
                         if(nofactivereplicas == tot_opsites)
                            stop = 1;
```

```
                           }
                   else if( flag == Lambda){
                            nofactivereplicas++;
                           *time_taken += nodes[prev].distance[next_replica];

nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
                            Push(next_replica);
                            prev = next_replica;
                            if( nofactivereplicas == nofreplicas)
                                stop = 1;
                            else if( nofactivereplicas == tot_opsites)
                                     stop = 1;
                            }
                 }
                 else nodes[prev].distance[next_replica] =
nodes[next_replica].distance[prev] = 0.0;
             }
             else
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
        }
        else if( (top_of_stack > 0) && (godown >= 0))
                  prev = Stck[godown--];
        else stop = 1;
     }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
    return(1);
else return(0);
}

/* Function to find the successor of site from the edges of the hypercube */
find_succ(prev,succ,nofreplicas,subset)
int prev,*succ,nofreplicas,subset;
{
if( (prev == 0) && (subset == 4)){
    if( (nodes[prev].distance[prev+1] != 0.0) &&
(nodes[prev].traversed[prev+1]!=1)){
         *succ = prev + 1;
         return(1);
         }
    else if( (nodes[prev].distance[prev+subset-1] != 0.0) &&
(nodes[prev].traversed[prev+subset-1]!=1)){
```

```
            *succ = prev + subset - 1;
            return(1);
            }
    else if( (nodes[prev].distance[prev+subset] != 0.0) &&
(nodes[prev].traversed[prev+subset]!=1)){
            *succ = prev + subset;
            return(1);
            }
    }
else if( (prev == 0) && (subset != 4)){
        if( (nodes[prev].distance[prev+1] != 0.0) &&
(nodes[prev].traversed[prev+1]!=1)){
            *succ = prev + 1;
            return(1);
            }
        else if( (nodes[prev].distance[subset+1] != 0.0) &&
(nodes[prev].traversed[subset+1]!=1)){
                *succ = subset + 1;
                return(1);
                }
         }
else if( prev == subset == 4){
        if( (nodes[prev].distance[prev%subset] != 0.0) &&
(nodes[prev].traversed[prev%subset]!=1)){
            *succ = prev + subset;
            return(1);
            }
        else if( (nodes[prev].distance[prev+1] != 0.0) &&
(nodes[prev].traversed[prev+1]!=1)){
                *succ = prev + 1;
                return(1);
                }
        else if( (nodes[prev].distance[prev+subset-1] != 0.0) &&
(nodes[prev].traversed[prev+subset-1]!=1)){
                *succ = prev + subset - 1;
                return(1);
                }
         }
else if( (prev !=0) && ( subset == 4)){
        if( (prev > subset) && (nodes[prev].distance[prev-subset] != 0.0) &&
(nodes[prev].traversed[prev-subset]!=1)){
            *succ = prev - subset;
            return(1);
            }
        else if( (prev < subset) && (nodes[prev].distance[prev+subset] != 0.0)
&& (nodes[prev].traversed[prev+subset]!=1)){
                *succ = prev + subset;
                return(1);
                }
        else if( ((prev+1)%subset==4) &&
(nodes[prev].distance[(prev+1)%subset] != 0.0) &&
(nodes[prev].traversed[(prev+1)%subset]!=1)){
                *succ = (prev + 1)%subset;
                return(1);
                }
        else if( ((prev+1)%subset==0) &&
(nodes[prev].distance[(prev+1)%subset] != 0.0) &&
(nodes[prev].traversed[(prev+1)%subset]!=1)){
                *succ = (prev + 1)%subset;
                return(1);
                }
        else if( (nodes[prev].distance[prev-1] != 0.0) &&
(nodes[prev].traversed[prev-1]!=1)){
            *succ = prev - 1;
            return(1);
            }
        else if( (nodes[prev].distance[prev+1] != 0.0) &&
(nodes[prev].traversed[prev+1]!=1)){
            *succ = prev + 1;
            return(1);
            }

        }
else if( prev !=0){
        if( (nodes[prev].distance[prev-1] != 0.0) &&
(nodes[prev].traversed[prev-1]!=1)){
            *succ = prev - 1;
            return(1);
            }
        else if( (nodes[prev].distance[(prev+1)%nofreplicas] != 0.0) &&
(nodes[prev].traversed[(prev+1)%nofreplicas]!=1)){
                *succ = (prev + 1)%nofreplicas;
                return(1);
                }

        }
```

```
return(0);
}

/* Function to pick an element from the aray */
pick_elem()
{
int i = 0;

while(Aray[i] != -1){
      i++;
      }

return(spec_rnum(1,i-1));
}

/* Function to rearrange the contents of the stack  */
rearrange_stck(F_Aray,nextfail,prev)
int F_Aray[],*nextfail,prev;
{
int tmp[MAXSIZE],i = 0,j = 0;

for( i = 0;i < MAXSIZE;i++)  /* Copy current stack to temp location*/
     tmp[i] = -1;
while( i < top_of_stack){
       if( Stck[i] != prev)
           tmp[j++] = Stck[i];
       i++;
       }
godown = top_of_stack = j;

for( i = 0; i < MAXSIZE; i++)
     Stck[i] = -1;

for( i = 0;i < top_of_stack; i++)
     Stck[i] = tmp[i];
F_Aray[*nextfail++] = prev;
}
```

```
/* Function to check if an element is in the Aray */
In_aray(elem)
int elem;
{
int i = 0;

while(Aray[i] != -1){
      if(elem != Aray[i])
         i++;
      else if( elem == Aray[i])
                return(1);
      }
return(0);
}

/* Function to simulate the Hierarchical policy */
void hierarchy(argv)
char *argv[];
{

int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_hierarchy();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_hierarchy(nofsites);
}

/* Driver routine  for Hierarchical policy */
void driver_hierarchy(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time = 0.0;
```

```
int i,success = 0;
void init_aray();
void copy_back();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(i!=0)
      copy_back(nofreplicas);
    if(run_hierarchical(nofreplicas,&time_taken)){
      success++;
      actual_time += time_taken;
      }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Hierarchical technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  =
%.3fms\n\n",actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function to run the Hierarchical policy */
run_hierarchical(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,flag1=0,next_replica;
int F_Aray[MAXSIZE],nextfail=0,prev = 0;
float min,max;

init_FAray(F_Aray);
tot_opsites = nofreplicas;

Push(prev);     /* Push site '0' onto the stack(site originating the update) */
while(!stop){
    if( !flag1){
       max = 0.0;
       if(find_max_site(nofreplicas,&max,&next_replica,prev));
       flag1=0;
       }
    else{
          min = 9999.0;
          if(find_min_site(nofreplicas,&min,&next_replica,prev));
          else stop = 1;
          }

          if( check_in_stack(next_replica) == 0){
             if( check_in_Aray(F_Aray,next_replica) == 0){
                remaining_sites = tot_opsites - nofactivereplicas;
                call_transition_find(remaining_sites,&flag);
                if( flag == Rho){
                   tot_opsites--;
                   if( nofactivereplicas == 1){
                      failed = 1;
                      nofactivereplicas--;
                      stop = 1;
                      }
                    else {
                        if(prev == top_of_stack-1)
                          F_Aray[nextfail] = Pop();
                        else F_Aray[nextfail] = prev;
                        if( top_of_stack >0)
                           prev = Stck[top_of_stack-1];
                        nextfail++;
                        nofactivereplicas--;
                        }
                    }
                else if(flag == Sigma){
                      tot_opsites--;
                      F_Aray[nextfail] = next_replica;
                      nextfail++;
                      if(nofactivereplicas == tot_opsites)
                        stop = 1;
                      }
                else if( flag == Lambda){
                       nofactivereplicas++;
                      *time_taken += nodes[prev].distance[next_replica];

nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
                       Push(next_replica);
                       prev = next_replica;
                       if( nofactivereplicas == nofreplicas)
                          stop = 1;
                       else if( nofactivereplicas == tot_opsites)
```

```
                  stop = 1;
              }
        }
        else nodes[prev].distance[next_replica] =
nodes[next_replica].distance[prev] = 0.0;
      }
      else
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
  }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
  return(1);
else return(0);
}

/* Function to simulate the Minimal Spanning Tree policy */
/* using PRIM's algorithm   */
void minspan(argv)
char *argv[];
{
int i,j,nofsites=0,success = 0;
float total_dist = 0.0;
float time_taken,actual_time;
void null_struct();
void init_struct();
void init_vals();
void init_aray();
void copy_back();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);

for( i = 0; i < Trials; i++){
   init_aray(nofsites);
   time_taken = 0.0;
   if(i!=0)
```

```
      copy_back(nofsites);
    if(run_minspan(nofsites,&time_taken)){
      success++;
      actual_time += time_taken;
      }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Minimal Spanning Tree policy =
%.3f\n\n", (float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function to run the Minimal spanning tree algorithm */
run_minspan(nofreplicas,time_taken)
int nofreplicas;
float *time_taken;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,next_replica,F_Aray[MAXSIZE],nextfail=0,prev = 0;
float min;

init_FAray(F_Aray);
tot_opsites = nofreplicas;

Push(prev);     /* Push site '0' onto the stack(site originating the update) */
while( !stop){
     if(call_go_thru(nofreplicas,&min,&next_replica,&prev)){
        if( check_in_stack(next_replica) == 0){
           if( check_in_Aray(F_Aray,next_replica) == 0){
              remaining_sites = tot_opsites - nofactivereplicas;
              call_transition_find(remaining_sites,&flag);
              if( flag == Rho){
                 tot_opsites--;
                 if( nofactivereplicas == 1){
                    failed = 1;
                    nofactivereplicas--;
                    stop = 1;
                    }
                  else {
                      if(prev == top_of_stack-1)
                        F_Aray[nextfail] = Pop();
                      else F_Aray[nextfail] = prev;
                      if( top_of_stack >0)
                         prev = Stck[top_of_stack-1];
                      nextfail++;
                      nofactivereplicas--;
                      }
                  }
               else if(flag == Sigma){
                     tot_opsites--;
                     F_Aray[nextfail] = next_replica;
                     nextfail++;
                     if(nofactivereplicas == tot_opsites)
                        stop = 1;
                     }
               else if( flag == Lambda){
                      nofactivereplicas++;
                       *time_taken += min;
  nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
                     Push(next_replica);
                      prev = next_replica;
                      if( nofactivereplicas == nofreplicas)
                         stop = 1;
                      else if( nofactivereplicas == tot_opsites)
                             stop = 1;
                      }
             }
             else nodes[prev].distance[next_replica] =
nodes[next_replica].distance[prev] = 0.0;
          }
          else
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
      }
      else stop = 1;
    }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed){
   return(1);
   }
else return(0);
}
```

```
/* Function to go through the Stack and find the minimum edge incident to
the graph */
call_go_thru(nofreplicas,min,next_replica,prev)
int nofreplicas,*next_replica,*prev;
float *min;
{
int
edge1[PAIRSIZE],edge2[PAIRSIZE],result_edge[PAIRSIZE],tmp_edge[PAIR
SIZE];
int i = 0, j , nextreplica1,nextreplica2,prev1,prev2,tmp_prev;
float min1,min2,min3,tmp_min;
void init_edgelist();

init_edgelist(edge1,edge2,result_edge,tmp_edge);

while( i < top_of_stack){
      if( Stck[i+1] != -1){
          min1 = min2 = min3 = 9999.0;
          if( find_min_site(nofreplicas,&min1,&nextreplica1,Stck[i])){
              edge1[0] =  Stck[i];
              edge1[1] =  nextreplica1;
              }
          if( find_min_site(nofreplicas,&min2,&nextreplica2,Stck[i+1])){
              edge1[0] =  Stck[i+1];
              edge1[1] =  nextreplica2;
              }
          compare_edge(edge1,min1,edge2,min2,result_edge,&min3);
          if( tmp_edge[0] != -1)

compare_edge(tmp_edge,tmp_min,result_edge,min3,tmp_edge,&tmp_min);
          else {
               tmp_edge[0] = result_edge[0];
               tmp_edge[1] = result_edge[1];
               tmp_min = min3;
               }
          }
      i++;
      }
if( i > 2){
   *prev = tmp_edge[0];
   *next_replica = tmp_edge[1];
   *min = tmp_min;
   return(1);
```

```
    }
 else if( i == 2){
        *next_replica = result_edge[0];
        *next_replica = result_edge[1];
        *min = min3;
        return(1);
        }
 else if( i == 1){
          if( find_min_site(nofreplicas,&min1,&nextreplica1,Stck[i-1])){
              *prev = Stck[i-1];
              *next_replica = nextreplica1;
              *min = min1;
              return(1);
              }
          else return(0);
        }
 else if(i == 0)
       return(0);
}

/* Function to initialise the edgelists */
void init_edgelist(edge1,edge2,result_edge,tmp_edge)
int edge1[],edge2[],result_edge[],tmp_edge[];
{
int i;

for( i = 0; i < PAIRSIZE; i++)
     edge1[i] = edge2[i] = result_edge[i] = tmp_edge[i] = -1;
}

/* Function to compare two  edge lists and place the result into result edge
list along with min value */
compare_edge(edge1,min1,edge2,min2,result_edge,min3)
int edge1[],edge2[],result_edge[];
float min1,min2,*min3;
{
if( (edge1[0] == -1) && (edge2[0] != -1)){
     result_edge[0] = edge2[0];
     result_edge[1] = edge2[1];
     *min3  = min2;
     }
else if( (edge1[0] != -1) && (edge2[0] == -1)){
        result_edge[0] = edge1[0];
        result_edge[1] = edge1[1];
        *min3  = min1;
        }
else if( (edge1[0] != -1) && (edge2[0] != -1)){
        if( min1 <= min2){
           result_edge[0] = edge1[0];
           result_edge[1] = edge1[1];
           *min3  = min1;
           }
        else {
           result_edge[0] = edge2[0];
           result_edge[1] = edge2[1];
           *min3  = min2;
            }
        }
}

/* Function to simulate the Combination policy */
void combo(argv)
char *argv[];
{
int i,j,nofsites=0;
float total_dist = 0.0;
void null_struct();
void init_struct();
void init_vals();
void driver_combo_pol();

null_struct();
init_struct();
read_in_data(argv,&nofsites);
init_vals(nofsites);
printf(" Nofsites = %d\n",nofsites);

find_total_dist(nofsites,&total_dist);
printf("\nTotal time for the network = %.3fms\n\n",total_dist);
driver_combo_pol(nofsites);
}
```

```
/* Driver routine  for Combination policy */
void driver_combo_pol(nofreplicas)
int nofreplicas;
{
float time_taken,actual_time=0.0;
int i,success = 0;
void init_aray();
void copy_back();

for( i = 0; i < Trials; i++){
    init_aray(nofreplicas);
    time_taken = 0.0;
    if(i!= 0)
      copy_back(nofreplicas);
    if(run_combine_policy(nofreplicas,&time_taken)){
      success++;
      actual_time += time_taken;
      }
    }
printf("Random efficiency = %.3f\n",rand_efficiency/(float)Trials);
printf("Probability of success using Combination technique = %.3f\n\n",
(float)success/(float)Trials);
printf("Time taken for updates to reach all sites  = %.3fms\n\n",
actual_time/(float)Trials);
rand_efficiency = 0.0;
}

/* Function to run the combination policy by calling */
/* Uniform flexible policy and Distance biased policy */
run_combine_policy(nofreplicas,time_taken)
float *time_taken;
int nofreplicas;
{
int nofactivereplicas = 1, stop = 0,done = 1, failed = 0,tot_opsites;
int remaining_sites,flag,counter,next_replica;
int F_Aray[MAXSIZE],nextfail=0,prev = 0;
float min;
void seed_random();

counter = spec_rnum(1,nofreplicas);
init_FAray(F_Aray);
tot_opsites = nofreplicas;
```

```
Push(prev);     /* Push site '0' onto the stack(site originating the update) */
while(!stop){
       if(counter != 0){
          counter--;
          seed_random();
          next_replica = spec_rnum(1,nofreplicas-1);
          }
        else{
            min = 9999.0;
            if(find_min_site(nofreplicas,&min,&next_replica,prev));
            else stop = 1;
            }
        if( check_in_stack(next_replica) == 0){
              if( check_in_Aray(F_Aray,next_replica) == 0){
                  remaining_sites = tot_opsites - nofactivereplicas;
                  call_transition_find(remaining_sites,&flag);
                  if( flag == Rho){
                      tot_opsites--;
                      if( nofactivereplicas == 1){
                          failed = 1;
                          nofactivereplicas--;
                          stop = 1;
                          }
                       else {
                            if(prev == top_of_stack-1)
                               F_Aray[nextfail] = Pop();
                            else F_Aray[nextfail] = prev;
                            if( top_of_stack >0)
                                prev = Stck[top_of_stack-1];
                            nextfail++;
                            nofactivereplicas--;
                            }
                       }
                  else if(flag == Sigma){
                          tot_opsites--;
                          F_Aray[nextfail] = next_replica;
                          nextfail++;
                          if(nofactivereplicas == tot_opsites)
                             stop = 1;
                          }
                  else if( flag == Lambda){
                           nofactivereplicas++;
                          *time_taken += nodes[prev].distance[next_replica];
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
                              Push(next_replica);
                              prev = next_replica;
                              if( nofactivereplicas == nofreplicas)
                                  stop = 1;
                              else if( nofactivereplicas == tot_opsites)
                                       stop = 1;
                              }
                  }
                  else nodes[prev].distance[next_replica] =
nodes[next_replica].distance[prev] = 0.0;
              }
              else
nodes[prev].traversed[next_replica]=nodes[next_replica].traversed[prev]=1;
      }

rand_efficiency = rand_efficiency + (float)nofactivereplicas/(float)nofreplicas;
if( done && !failed)
    return(1);
else return(0);
}

/* Function to get the options required */
get_option(id)
int *id;
{
print_options();
scanf("%d",id);
if(*id > 12)
   *id = 0;
return(1);
}


/* Function to print the Input Options  */
print_options()
{
printf("\n");
printf("%-53s\n","Press option 1 for simulating Random policy");
printf("%-53s\n","Press option 2 for simulating Distance biased policy");
printf("%-53s\n","Press option 6 for simulating Ring policy");
printf("%-53s\n","Press option 7 for simulating Binary Tree policy");
printf("%-53s\n","Press option 8 for simulating Minimal Spanning Tree
policy");
printf("%-53s\n","Press option 9 for simulating Mesh policy");
printf("%-53s\n","Press option 10 for simulating Hypercube  policy");
printf("%-53s\n","Press option 11 for simulating Hierarchical  policy");
printf("%-53s\n","Press option 12 for simulating Combination  policy");
printf("%-28s\n","Press option 0 for quitting");
}
```

VITA

PRAKASH JOHN THOMAS

Candidate for the Degree of

Master of Science

Thesis : PARTNER SELECTION TECHNIQUES FOR TIMESTAMPED ANTI-ENTROPY PROTOCOLS

Major Field: Computer Science

Biographical Sketch:

Personal Data: Born in Tiruvalla, Kerala, India, June 14, 1968, son of M. J. Thomas and Joykutty Thomas.

Education: Graduated from St. Joseph's Boys Higher Secondary School, Coonoor, India, in May 1986; received Bachelor of Engineering in Computer Science and Engineering from Bharathiar University at Coimbatore in June 1990; completed requirements for the Master of Science degree at Oklahoma State University in July 1993.

Professional Experience: Graduate Research Assistant, Department of Agricultural Engineering, Oklahoma State University, September 1991, to August 1993.