MODELING UPDATE CACHING IN WEAK CONSISTENCY PROTOCOLS

By

ZEESHAN SHAFAQ SYED

Bachelor of Engineering

Computer Systems Engineering

N. E. D. University of Engineering and Technology

Karachi, Pakistan

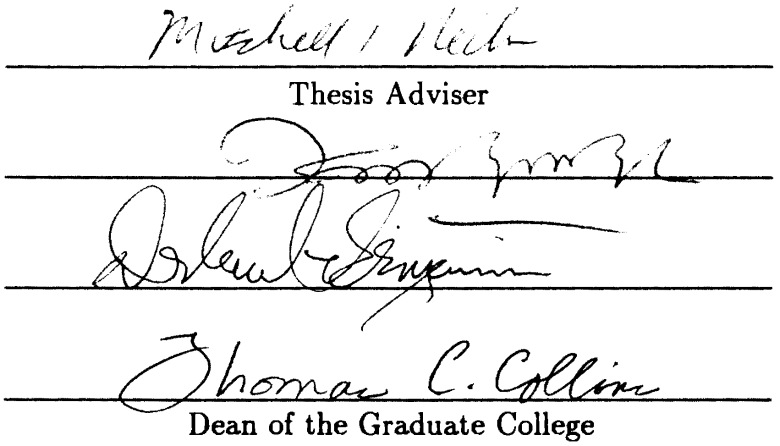
1990

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, 1994

OKLAHOMA STATE UNIVERSITY

MODELING UPDATE CACHING IN WEAK
CONSISTENCY PROTOCOLS

Thesis Approved:



Thesis Adviser

Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my sincere thanks to my graduate advisor Dr. Mitchell L. Neilsen for his guidance, encouragement and help for completion of my thesis work. Without his constant support, supervision and ideas, this thesis would have been impossible. His group seminar meetings helped me to broaden the horizons of knowledge in other areas, which are not related to my thesis work. I would like to thank Dr. K. M. George and Dr. D. Paul Benjamin for serving on my graduate committee and providing me with some feedback for improving my thesis. I want to thanks Dr. George for his advice and suggestions during my stay at Oklahoma State University. I want to express my special thanks to Dr. Mansur Samadzadeh for his guidance, advice and suggestions during my stay at Oklahoma State University. I want to thank Dr. Blayne Mayfield for his support and encouragement. I want to express my thanks to the persons who are providing knowledge bases on the Internet.

My respectful and very special thanks to my late father Syed Muhammad Rafique and my mother Mrs. Safia Rafique for their love, encouragement, support and confidence on me. It was my parents wish that I should pursue my MS from USA. I want to thanks all of my family members for their love and support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

A computer network is an interconnected collection of autonomous computers. A distributed system is a computer network in which the existence of the underlying network is transparent to the user [Tan89]. If the interconnected computers are located in the same room, building or campus, then the network is called a Local Area Network. If the interconnected computers are located in different cities, states, countries or continents, then the network is called Wide Area Network. The Internet is a major public computer network. The size of the Internet or almost any wide area network imposes some constraints on the availability of the information. Ideally, distributed systems should make the information highly available. The system must respond gracefully to host and network failure. The presence of mobile computers imposes additional constraints. For example, a machine can be disconnected from the network temporarily.

The service provided by a distributed system can be implemented using a centralized server or a distributed server. Furthermore, copies of data can be stored at different sites. Copies of data that are stored at different sites are called replicas. To provide good response time and to meet the constraints outlined above, replicas should be geographically located near the clients. Processes manage replicas by communicating over a network through message passing. Two possibilities exist for updating the replicas: update synchronously at all replicas, or update asynchronously at different replicas. Synchronous updating implies strong consistency among the replicas and asynchronous updating implies weak consistency.

The Time Stamped Anti-Entropy (TSAE) protocol is used to propagate updates efficiently in a system that enforces weak consistency . This protocol is discussed in detail in Chapter 2. An analytical model of the TSAE protocol has been proposed

by Golding and Long [GL92a].

1.1 Thesis

We propose a model that measures the effect of caching updates on the probability of successfully propagating an update to all operational nodes. A Monte Carlo simulation is carried out to analyze the proposed Markov model. The simulator, when run with a zero caching probability, produces the same results obtained by the model of Golding and Long [GL92a]. When a non-zero caching probability is used by the simulator, the probability of success decreases. However, the benefits of caching may outweigh this small decrease.

1.2 Organization

The thesis is organized into the following chapters:

- Chapter 2: A detailed literature review of the previous work on weak consistency protocols.
- Chapter 3: The proposed analytical model is described and the derivations for the model are presented.
- Chapter 4: The issues related to simulator design and implementation are discussed.
- Chapter 5: The results obtained are analyzed.
- Chapter 6: A summary of the thesis work and a few ideas related to the extension of the current work are discussed.
- Appendix A: A user manual for the simulator.

- Appendix B: An installation guide.

- Appendix C: Code listing including header files.

CHAPTER 2

LITERATURE REVIEW

2.1 Consistency and Replicated Service

In a distributed system, if there is only one central server and the clients access the service through that server, then the response time will be greatly affected by the distance between the clients and the server. Also, if the central server is down, then no service will be available. Service replication is the mechanism for achieving availability demands and scalability [Gol92]. The replicated data model is shown in Figure 1. If replicated service is available, fast response time can be provided. A large number of replicas makes it possible to spread the query load over more sites. Furthermore, if any one replica has failed, the clients can still access the data. A large number of replicas makes it possible to spread the query load over more sites.

Consistency protocols in which all copies of data at every replica are required to be identical at all times are called strong consistency protocols [BHG87]. Such protocols are used in applications where consistent information is necessary. For example, in an airline reservation systems, all replicas should have the same data. However this imposes additional constraints because all replicas must be updated synchronously. Consequently, the time required to read or write information will be much larger. Also, updates must be propagated immediately to a number of sites. However, the update time might be the peak time for network traffic. Strong consistency protocols performance degrades sharply when communication is unreliable or the network is partitioned [GL92b].

Protocols that allow inconsistencies among different replicas are called weak consistency protocols. An application that uses a weak consistency protocol must be able to tolerate temporary inconsistencies.

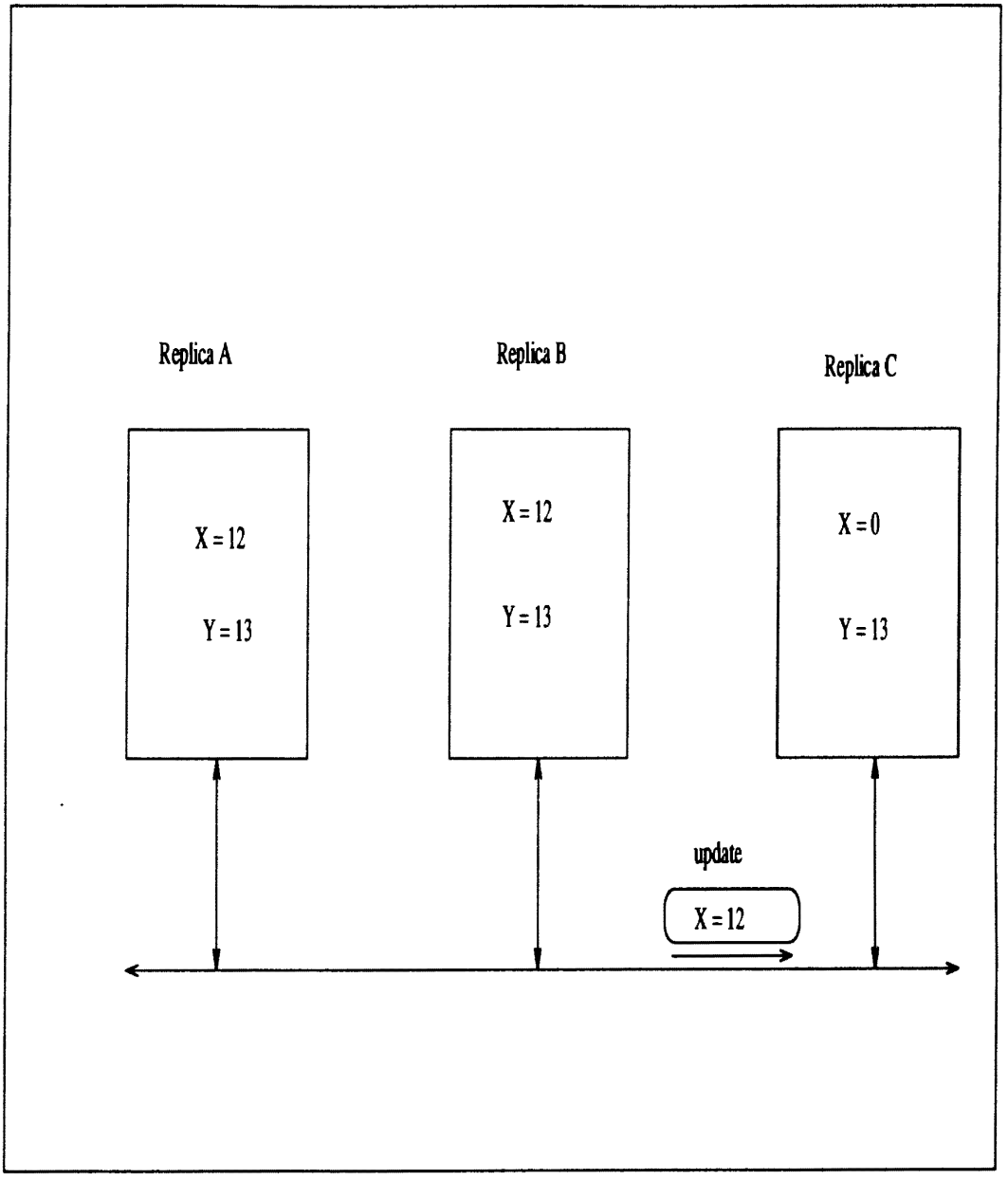


Figure 1. Replicated Data

For example, distributed load balancing is a candidate for a weak consistency protocol [GL92a]. In a wide-area network, the strong consistency constraint is very difficult to satisfy and degrades the response time. In contrast, weak consistency protocols allow updates to be asynchronous. Asynchronous updating provide each replica with more independence. Weak consistency protocols allow for replicas to suffer transient failures, and only guarantee the eventual propagation of an update to all replicas. For example, Figure 1 shows an update that originated at Replica A that has only reached Replica B. Thus, weak consistency protocols work well for an unreliable network such as the Internet.

2.2 Types Of Consistency

Levels of consistency in a replication protocol depends on the consistency provided by the communication protocol. There are three different categories [Gol92]:

1. Message Delivery

2. Delivery Ordering

3. Time of Delivery

2.2.1 Message delivery

Message can be delivered reliably or with best effort. Reliable delivery guarantees message delivery. In case of best effort, the system tries its best to deliver the message, but message delivery is not guaranteed. Atomic delivery means that a message is delivered to each replica or to none. Quorum delivery means that a message is delivered to at least some fraction of the total replicas.

2.2.2 Delivery ordering

Messages can be delivered in any order. A total ordering means that every replica will receive messages in same order. Causal ordering means that any message with a potential causal relation will be delivered in the same order at all replicas. In per-process or FIFO channel ordering, the message from any particular replica are delivered in order, but messages from some other replica may be interleaved.

2.2.3 Time of delivery

Synchronous delivery is performed immediately within a bounded time. Interactive delivery is performed immediately, but may require finite but unbounded time. Bounded delivery may queue the messages or delay them, but delivery will complete within a bounded time. Eventual delivery queues or delays the messages and it may require a finite but unbounded time to deliver.

2.3 Message Updates

The updates are performed by a client at one replica. Then the updates are propagated to other replicas using message passing. In synchronous updates, all replicas receive updates at the same virtual time and hence all replicas are consistent. However, all servers (replicas) stop providing service before propagating an update to all other replicas. In contrast, with asynchronous updates, the updates are propagated eventually, but servers (replicas) do not stop providing service while updating. As discussed earlier, synchronous updates are used in strong consistency protocols, while asynchronous updates are used in weak consistency protocols.

The asynchronous communication is used among the replicas (using weak consistency protocols) for updates. The replicas are grouped together for propagating and receiving updates. The following methods can be used for group communication

[Dea88]:

1. Direct Mail
2. Rumor Mongery
3. Anti-entropy

2.3.1 Direct mail

The source site immediately multicast an update to all other sites. If there are n sites, then n messages are generated per update. Each message traverses all network links between its source and destination. Consequently, the traffic generated is proportional to the number of sites times the average distance between sites [Dea88]. This protocol is efficient, but not fully reliable. Individual sites do not know about the state of other sites. Queues are used to keep the messages at the server to avoid delay in senders. These queues are maintained on stable storage, to prevent it from being affected if the server crashes.

2.3.2 Rumor mongery

Initially all sites are ignorant. When a site receives an update, it considers the update as a 'hot rumor'. Any site holding a hot rumor selects another site at random and passes the update to the chosen site. When too many sites observe the update, the update is no longer a hot rumor. So, only the most recent updates are sent to other replicas. If rumor cycles are too fast, then there is a chance that an update might not reach all sites [Dea88].

2.3.3 Anti-Entropy

In anti-entropy, an update occurs at one site and is propagated to the other replicas. A replica holding an update initiates an anti-entropy session with a partner. In this way,

from time to time, replicas initiate anti-entropy sessions to exchange updates, and hence each update is eventually propagated to all replicas. The protocol guarantees the eventual delivery of updates to all replicas. Anti-entropy sessions can be initiated at off-peak time to reduce peak network traffic [GT92]. A detailed description of the anti-entropy protocol follows in the next section.

2.4 Timestamped Anti-Entropy Protocol Description

Timestamped anti-entropy (TSAE) is developed by Richard A. Golding [GL93, Gol92, GL92a] to be used as weak consistency protocol. TSAE provides reliable, eventual delivery. The message is originated at one replica and is propagated in the background to all other replicas. TSAE supports total or causal message ordering, it supports mobile computing and it fulfills the constraints outlined above. TSAE requires loosely synchronized clocks between all replicas. Replicas have access to stable storage (a hard disk) which is not affected by replica crashes. There are two types of failures: transient failures and permanent failures. In case of transient failure, a site (replica) goes down for a short period of time and then it comes back up and joins the protocol again. In case of permanent failure, the replica is permanently removed from service. Permanent failures are fail-stop; that is in case of permanent failure, a site does not send any failure message, but simply stops.

When a replica wishes to send an update, it stamps the update with the current logical time and a replica identifier and stores the update in a message log. A message log is shown in Figure 2. There are three replicas A, B and C in the group. For message exchange,each replica maintains a summary timestamp vector. The summary timestamp vector is indexed by the replica identifier and contains the greatest timestamp it has received from all other replicas. When a replica enters into an anti-entropy session with a partner, it compares its summary timestamp vector with the partner's summary timestamp vector. Hence, a replica can determine which updates

need to be exchanged. Then, replicas exchange the determined updates. As a result of this exchange, the partners also update their summary timestamp vectors. Thus, after the end of a session, both partners have the same summary timestamp vector.

Consider a distributed system having three replicas A, B and C. At some point in time, replicas A and B enter into an anti-entropy session. The message log of replicas A and B, before the anti-entropy session, are shown in Figure 2 and their respective summary timestamp vectors in Figure 3. Figure 4 shows the replicas after an anti-entropy session between replicas A and B. The summary vectors after anti-entropy are shown in Figure 5. As mentioned earlier, both replicas will have same summary timestamp vector after the anti-entropy session, hence only one vector is shown.

The message logs can become very large, so there should be some method to purge the message logs. The replicas need to determine which messages have been received by all other replicas, to limit the size of their message logs. This is achieved by having an acknowledgment time vector. The acknowledgment time vector is also exchanged during an anti-entropy session. Any message in the log having a timestamp smaller than every timestamp in the acknowledgment timestamp vector has been received and acknowledged by every other replica in the group, so it can be purged.

2.4.1 Best-effort multicast

The performance of the TSAE protocol can be improved by combining it with best-effort multicast. First, the replica receiving an update will multicast the update to all replicas and then engage in anti-entropy sessions. Replicas that have not received the multicast message, due to the unreliable network, will receive it through anti-entropy. Furthermore, the time required to propagate an update to all replicas will decrease sharply.

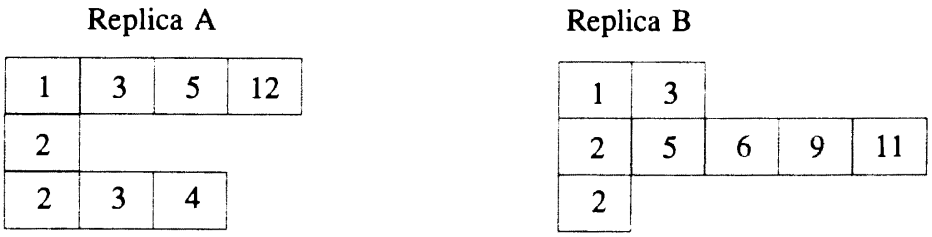


Figure 2. Message Logs for Replicas (Source: [GL93])

Figure 3. Summary Vectors for Replicas (Source: [GL93])

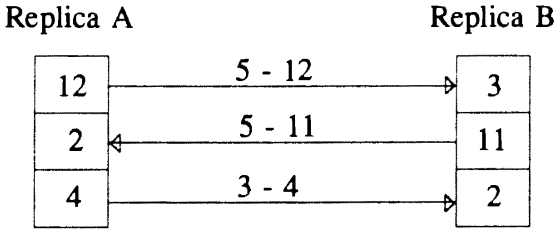


Figure 4. Anti-Entropy Session (Source: [GL93])

Figure 5. Summary Vector after Anti-Entropy Session (Source: [GL93])

2.4.2 Partner selection

The replicas can use several partner selection policies for anti-entropy sessions. Partner selection policy effects message delivery and hence the degree of consistency between replicas, and the amount of network traffic generated by the protocol.

Golding [GL93] has given three classifications for the partner selection policies: random, deterministic and topological. Thomas [Tho93] has extended this classification to include two more policies: hierarchical and combination.

Random Policies

The replica selects its partner randomly for an anti-entropy session. In the uniform policy, each replica has an equal probability of being selected. In a distance-biased policy nearby replicas have a greater probability of being selected. In oldest-biased, the probability of selecting a replica is proportional to the age of its replica in the summary timestamp vector.

Deterministic Policies

These policies use some fixed rule to determine the replica to select as a partner. In oldest-first, a replica with the oldest (smallest) summary timestamp vector value is selected. In latin square a truncated square of size n x n-1 is used, where every row and column has every entry just once [ABM87]. Anti-entropy sessions are divided into rounds. This policy guarantees messages to be received by all replicas in O(log n) time.

Topological Policies

In these policies replicas are organized into some fixed graph structure. In the ring policy, replicas are organized into a ring. In the binary tree policy, replicas are orga-

nized into a binary tree, and message are propagated randomly along tree branches. In the mesh policy, replicas are organized in a two dimensional mesh. In the hypercube policy [Tho93], the replicas are arranged as nodes in a hypercube. In the minimum spanning tree policy, replicas are arranged as nodes of a graph [Tho93]. Anti-entropy sessions are denoted as edges in a minimal spanning tree. The minimal spanning tree is constructed by using PRIM's algorithm [TS91].

Hierarchical Policies

The replicas are divided into a hierarchy. Level 1 is the top level, followed by level 2, and so on. The replicas at level 1 are connected with the most expensive links.

Combination Policies

Numerous policies can be obtained by combining many of the above described policies. For example a random policy can be combined with a distance-biased policy. Thus improved performance can be obtained by combining different policies.

2.5 Group Communication Architecture

The group communication architecture, as proposed by Golding [GL92b] is shown in Figure 6. The architecture consists of four components: an application, message delivery and ordering components, and a group membership component. These components communicate through a shared data structure.

The message delivery component implements a multicast communication service that exchanges messages with other replicas. It decodes incoming messages and route them either to the group membership component or to the log. It may also maintain the summary information of messages received and sent.

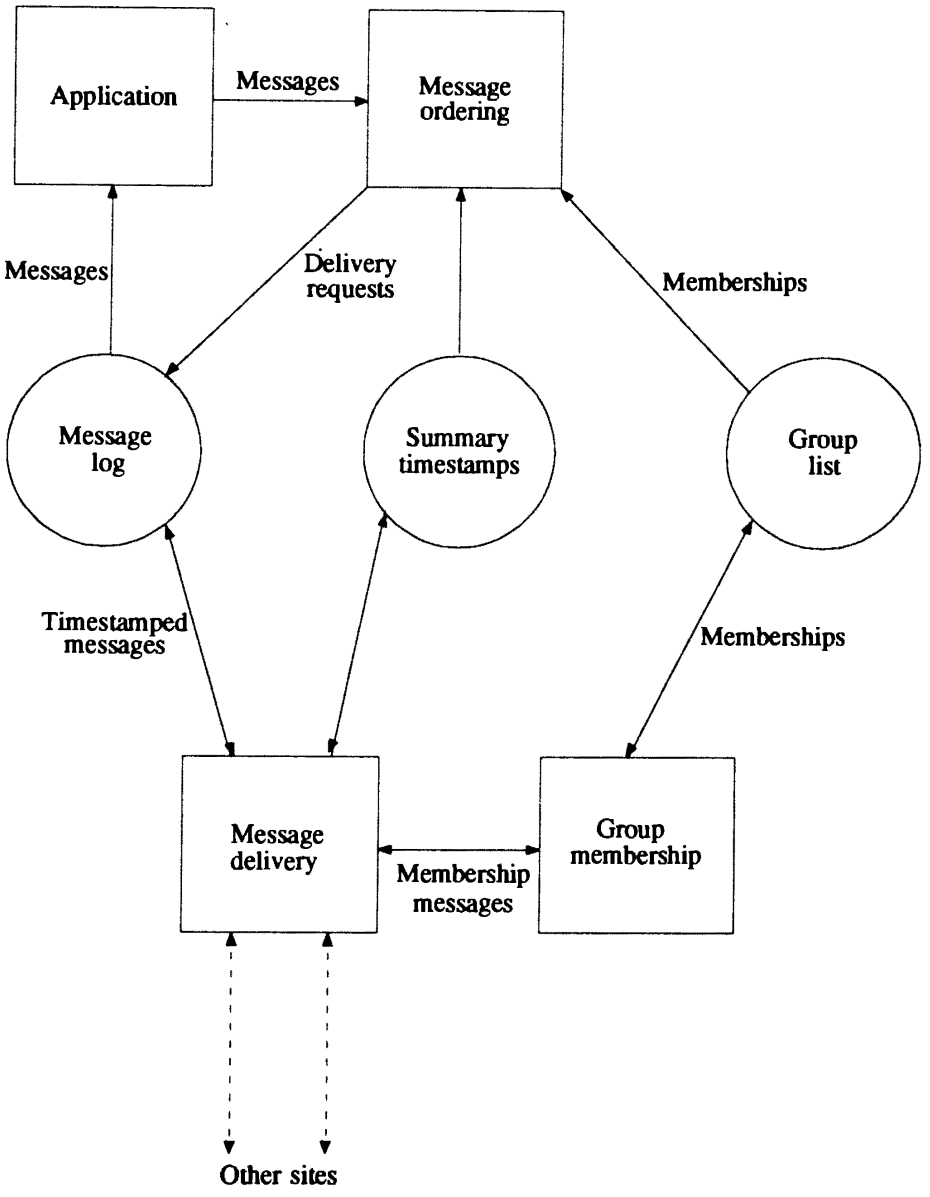


Figure 6. The Components Implementing a Weak Consistency Group Multicast System (Source: [GL92b])

The group membership component maintains the list of replicas (processes) that are in the group. The list is called the local view of the replica group. When there is a change in the list, this component communicates with the peer component of the other replicas, using the group membership protocol. The message ordering component is responsible for ordering of incoming messages. The ordering is application dependent. It also processes outgoing messages so that the ordering components at other process will have enough information to properly order messages.

The Refdbms system implements a distributed bibliographic database. It is based on TSAE. The Tattler system is a distributed availability monitor for Internet and it uses a weak-consistency protocol.

2.6 Protocols for Epidemic Group Membership Management

Golding [GT92] has proposed the following protocol for group membership.

2.6.1 Data structures

Each replica maintains a view of the members in the group of which it is part. Updates in group membership are propagated among replicas, like the updates in the data. The status and timestamp fields are used for group membership. The replica which is a member of a group has a status of 'member'. One which has left the group has a status of 'left' and a failed replica has a status of 'failed'.

2.6.2 Initializing a new group

A replica can create a new group through operation initialization. The initialization operation creates a new group and sets up a group membership.

2.6.3 Group join

A replica can join a group by contacting enough members and by finding enough sponsors to satisfy the k-resilience condition. A knows-about graph that can withstand k simultaneous replica failures and still be correct is called k-resilient membership graph. The knows-about graph is a directed graph for representing membership in the group.

2.6.4 Group leave

When a member leaves a group, it should not destroy the k-resilience condition.

2.6.5 Failure recovery

Anti-entropy inherently handles the temporary failures. As soon as replica comes up again it starts receiving the anti-entropy sessions. Permanent failure is a difficult problem and must be handled to maintain k-resilience.

CHAPTER 3

PROPOSED MODEL

The proposed model measures the effect of caching updates in weak consistency protocols. The Markov model of the system for three sites is shown in Figure 7. Each state is labelled as <m,f>. Anti-entropy sessions are assumed to be a Poisson process. Let λ_a denote the probability that a site having an update engages in anti-entropy with a site that doesn't have the update. Let λ_f denote the probability that a site fails. Let f denote the total number of replicas and m denote the number of available replicas that have observed the update. Let ψ denote the probability that the update is cached instead of being propagated. The following assumptions are made to simplify the analysis:

1. Nodes (replicas) are independent.

2. Failure means that an update has been lost completely. Success means that an update has been received by all available sites.

3. Failure can either occur before or after an anti-entropy session, but not during a session.

4. The network is fully connected.

5. All failures are permanent. Temporary failures are not considered.

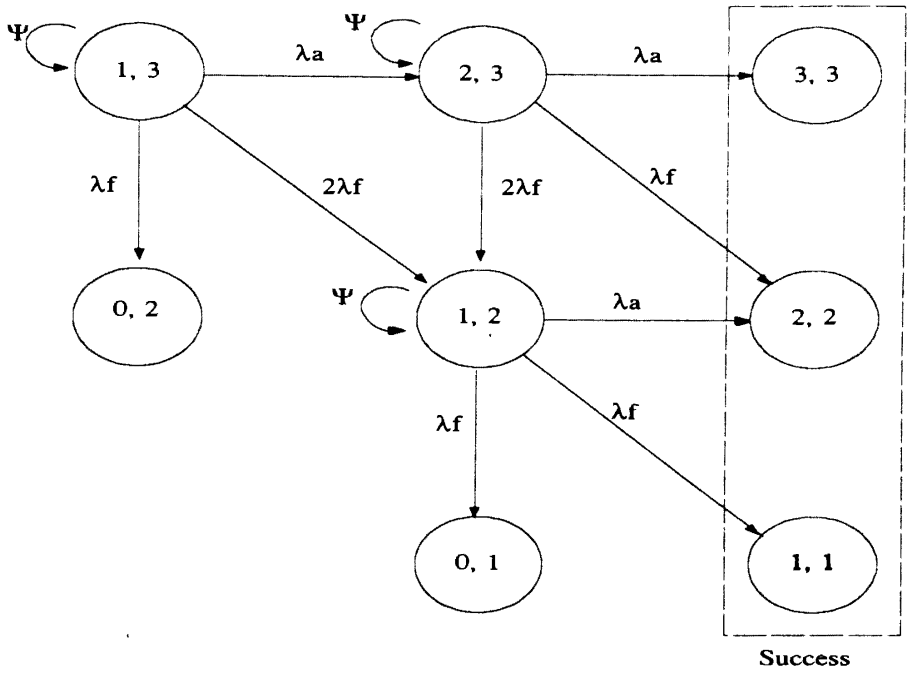


Figure 7. Markov Model

Caching should improve the performance because it will reduce the communication overhead. Several updates can be cached and sent through the network at once. However, caching will also decrease the probability of success. To analyze the effect of caching updates, we use the model shown in Figure 8.

3.1 Caching Updates

Let α denotes the probability that an update is cached, instead of being propagated immediately through an anti-entropy session. The states are identified by the ordered pairs `<m,f>`. The transitions are labelled as T1, T2 and T3, with the following meanings:

- T1: The successful propagation of an update to another site (which does not have the update). So m increases by 1 after a T1 transition and f remains the

same.

- T2: A site site which has the update has failed. This transition decreases m by 1 and hence decreases f by 1.

- T3: A site which does not have the update has failed. This transition decreases f by 1, but m remains the same.

The model in Figure 8 shows that there are only three possible transitions T1, T2 and T3 out of an intermediate state. But when the the system reaches a final state (state of success or state of failure), then the system stays there forever, for a particular update. To execute the experiment a number of times, every final state in the system will return back to the initial state `<1,f>` with probability 1. The generic transition diagram is shown in Figure 9. Consider the intermediate state given by `<m,f>`.

- After transition T1, the new state will be `<m+1,f>`.

- After transition T2, the new state will be `<m-1,f-1>`.

- After transition T3, the new state will be `<m,f-1>`.

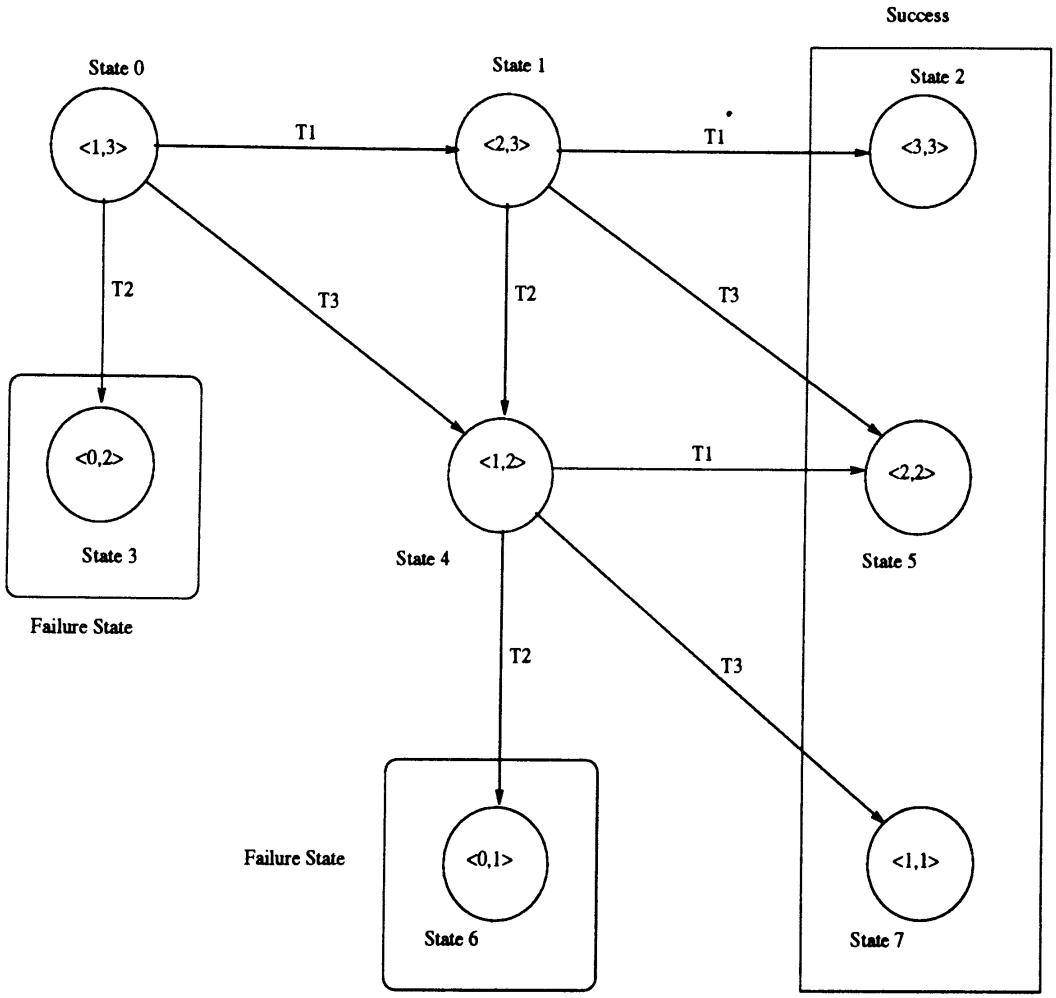


Figure 8. State transition diagram of proposed Markov Model for 3 sites

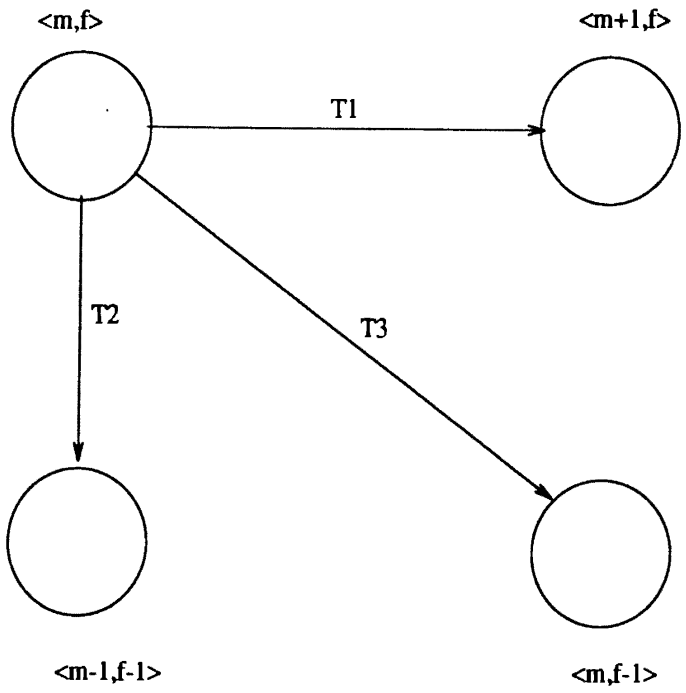


Figure 9. General State Transition Diagram

3.2 Mathematical Formulae

If there are n sites in the system. Then, the total number of feasible states in the Markov model for the system is given by

$$(n) + \sum_{i=2}^{n} i$$

which can be expressed as

$$(n-1) + \sum_{i=1}^{n} i$$

which can be expanded to

$$(n-1) + \frac{n(n+1)}{2}$$

which is finally equal to

$$\frac{n^2 + 3n - 2}{2}$$

For example, for $n = 3$, there are 8 feasible states, as shown in Figure 8.

Consider the model in Figure 8. Let us assume that the system is at state **<m,f>**. Since m is the number of sites that have observed the updates and f is total number of available sites, then

- f-m = number of sites that have not observed the update;
- f-1 = number of other operational sites
- therefore, (f-m)/(f-1) is the fraction of sites that have not received the update.
- The rate of anti-entropy is λ_a.
- If the caching probability is α, then the rate of anti-entropy is $\lambda_a(1-\alpha)$.

- Therefore, the probability of useful anti-entropy session (denoted by transition T1) is given by

$$m\frac{f-m}{f-1}\lambda_a(1-\alpha)$$

.

Since m is the number of sites having the update and λ_f is the probability that a site will fail, the probability of transition T2 is given by

$$m\lambda_f$$

.

Since f denotes the the number of available sites and m denotes the number of sites that have observed the update,

- (f-m) denotes the number of sites that have not received the update;
- the probability that the any of the sites that have not received update has failed is given by

$$(f-m)\lambda_f$$

.

3.3 Expected Behaviour

It is evident from the formula for T1, that the probability of success will decrease and hence indirectly, the probability of failure will increase. The effect of caching is measured in the results obtained through the simulation of the proposed Markov model.

CHAPTER 4

SIMULATOR DESIGN AND IMPLEMENTATION ISSUES

To simulate the Markov model discussed in Chapter 3, a Monte Carlo simulator is designed and implemented. The simulator code is attached in Appendix C. The simulator uses the Markov model, rather than using event driven simulation. The simulator is designed in such a way that the results can be obtained in a variety of ways.

The probability of caching an update, can be specified for each replica. So the simulator can run for different values of α.

The relative propagation rate [GL92a], ρ, is defined as the ratio of the successful anti-entropy rate to the permanent site failure rate. That is

$$\rho = \lambda_a / \lambda_f$$

An array of ρ values can be specified, for each run of the simulator. For a particular value of λ_f, different values of λ_a are calculated. These values are then used to run the simulator to measure the probability of success.

4.1 System State

Based on the number of sites, n, the total number of feasible states can be calculated using the formula derived, in Chapter 3. Since a state of a system is identified by **<m,f>**, the following structure is used for states

typedef struct{ int m; int f; } STATE;

The array of states is generated and initialized. This array is indexed from zero. The first member of the array is always state **<1,n>**.

4.2 Generation of Adjacency Matrix

Once the states are generated, the adjacency matrix, denoting the probability of transition to the next state, is generated, using the formulae for transitions T1, T2 and T3 (as discussed in Chapter 3). The probability of transition to a non-adjacent state (keeping in view the model of Figure 8) is 0. For example, the probability of transition from state 0 <1,3> (refer to Figure 8, for three sites) to state 2 <3,3>, is zero, because state 2 is not adjacent to state 0. But from state 0 <1,3> to state 1 <2,3> is transition T1, which can be calculated using the formula of Chapter 3. Table 1 shows the adjacency matrix for 3 sites, with ρ equal to 50, α equal to .2, and 3000 updates.

Table 1 Adjacency Probability Matrix

State	0	1	2	3	4	5	6	7
0	0.000	0.930	0.000	0.023	0.047	0.000	0.000	0.000
1	0.000	0.000	0.930	0.000	0.047	0.023	0.000	0.000
2	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
3	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
4	0.000	0.000	0.000	0.000	0.000	0.952	0.024	0.024
5	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
6	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
7	1.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

4.3 Scaling Transition Probabilities

The sum of all transitions should be 1 for each row in the adjacency matrix. Since λ_a and λ_f are independent, the sum may not be 1. For each row of the adjacency matrix, the transition probabilities are scaled to sum up to 1. Table 1 shows the matrix after scaling.

4.4 Random Number Generator

The random number generator routine written by M. H. MacDougall, [Mac87] is used. A uniform random number generator is used for generating the state transition probabilities.

4.5 Success and Failure Probability

A failure state is a state from which an update cannot be propagated in the system. Therefore, states of the form <0,a> are failure states. A success state is a state in which an update reaches all operational sites. Therefore, states of form <a,a> are success states.

A failure state or success state is a final state. Once the system reaches a final state, the time it spends there is counted as 1. After each update reaches a final state (whether it is successful or not), the system returns to the initial state <1,n>, to determine the fate of the next update.

From all non-final states (intermediate states), the system eventually reaches to a final state (the time spent in each non-final state is zero).

The simulator is started in state 0 <1,n>. Depending on the outcome of the random number generator and the transition probability out of that state (any value in that particular state's row in the adjacency matrix), the next state is calculated and the system advances to the next state. Eventually, the system must reach a final state.

The system is run for a specific number of updates, and the percentage of time that the system ends up in success or failure, is calculated. Hence, the probability of success is calculated for each run; that is, for a particular value of α, ρ and number of sites n.

4.6 Output of Simulator

The simulator expects some parameters from standard input. When all parameters are supplied to the simulator, it generates output on standard output It also generates a user specified data file.

The output sent to standard output is the value of λ_f and the number of updates. Then, the ρ the probability of success, the α and the number of sites for each run of the simulator, is printed on standard output.

The user specified data file contains ρ, the probability of success, α, and the number of sites. These four columns are printed in the output file without column labels, so that the output data file can be fed to a graph plotting utility.

CHAPTER 5

ANALYSIS

The simulator can be run in a variety of ways to study the effect of each parameters on the probability of success.

As defined in Chapter 3, a success state is a state in which all operational sites have received the update.

To study the effect of α, the probability of caching of update, on the success rate, Figure 10 is plotted for varying values of ρ. The simulator was run for 3000 updates and for ten sites. Two curves are plotted in the figure. One for the $\alpha = 0.0$ and the other for $\alpha = 0.5$. It is evident from the graph that the probability of success increases as ρ increases. For higher values of ρ, the probability of success approaches 100 percent. The probability of succees is higher for $\alpha = 0.0$ than for $\alpha = 0.5$, as anticipated, for every value of ρ. The reason for this behavior is that caching reduces the rate of anti-entropy. Hence as α increases, the probability of success decreases.

To analyze the effect of α on the probability of success, Figure 11 is plotted for different values of α. The value of λ_f is chosen to be 0.02, and ρ to be 50, which is quite reasonable (as determined experimentally). The simulator was run for 3000 updates with ten sites. The probability of success decreases as the caching probability, α, increases. The decrease in the probability of success is quite flat for $\alpha < 0.8$. But for $\alpha > 0.8$, the probability of success decreases quite rapidly. One interesting observation can be made from the graph, that when α is 1, even then the system goes to some successful state (although the probability of success is just 10%). For $\alpha =$ to 0.0, the probability of T1 transition becomes zero. That is, the system never follows the useful anti-entropy transition.

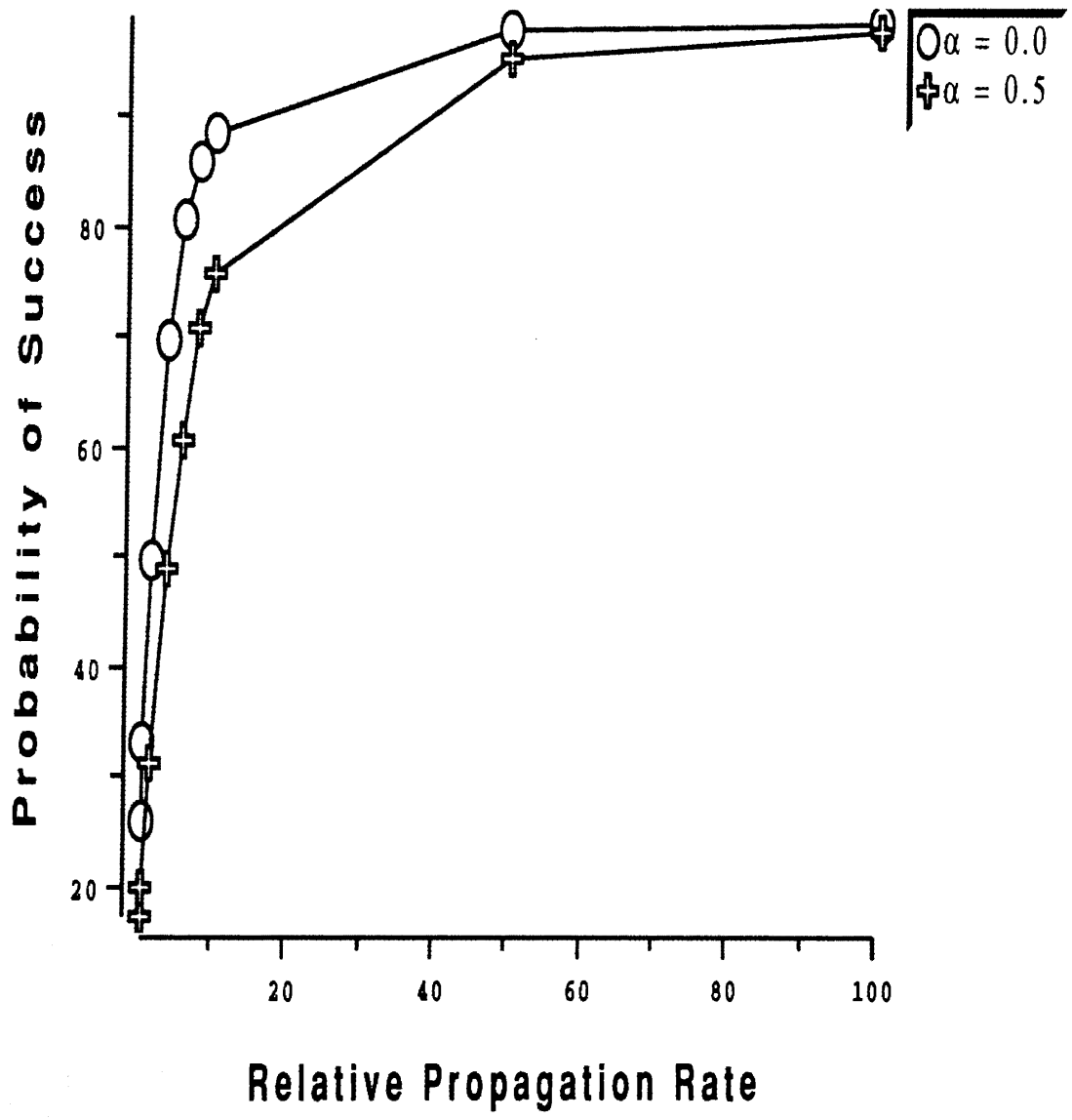


Figure 10. Success Probability vs Relative Propagation Rate and Caching Probability

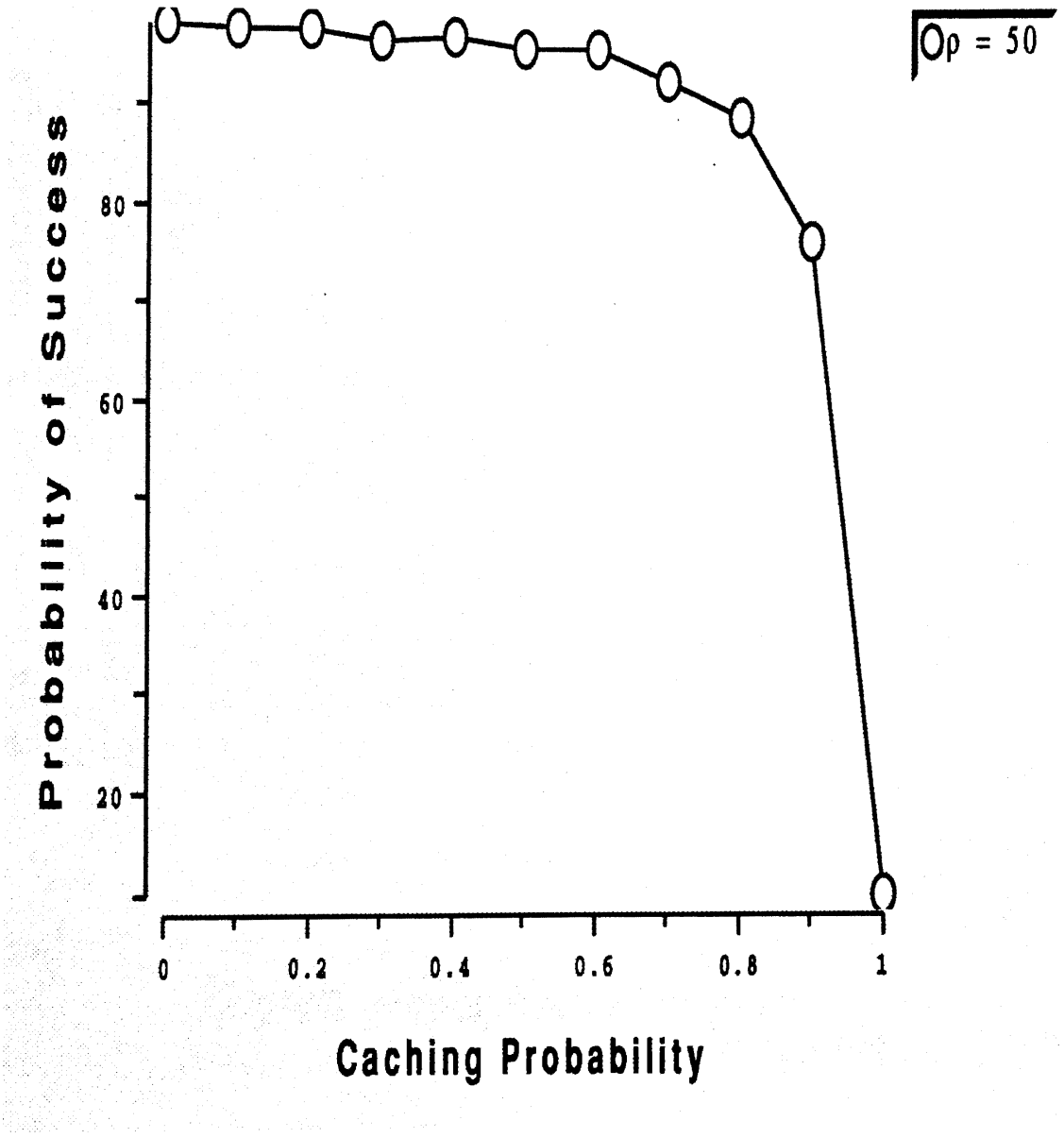


Figure 11. Success Probabiliy vs Caching Probability

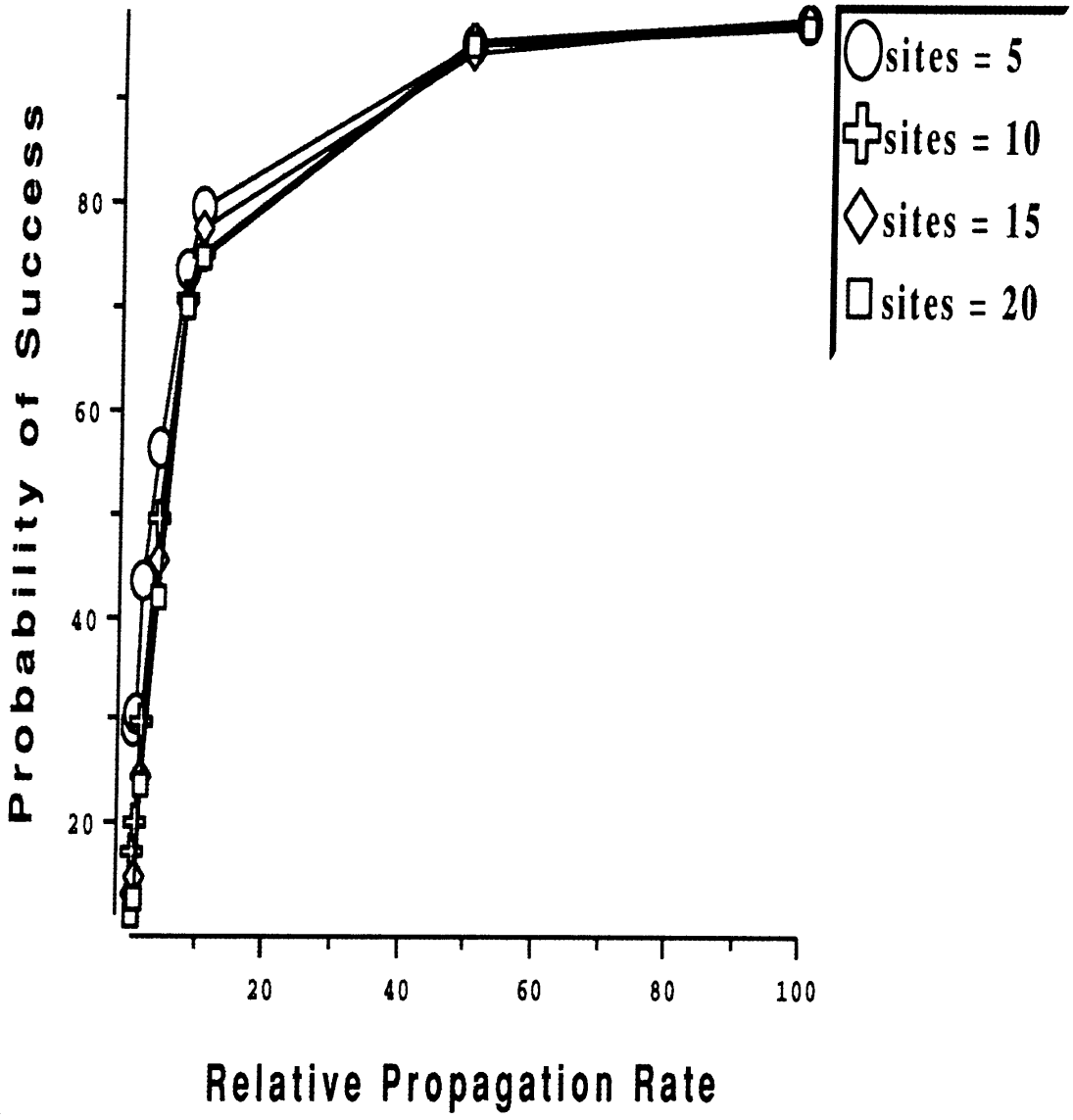


Figure 12. Success Probability vs Relative Propagtion Rate and the Number of Sites

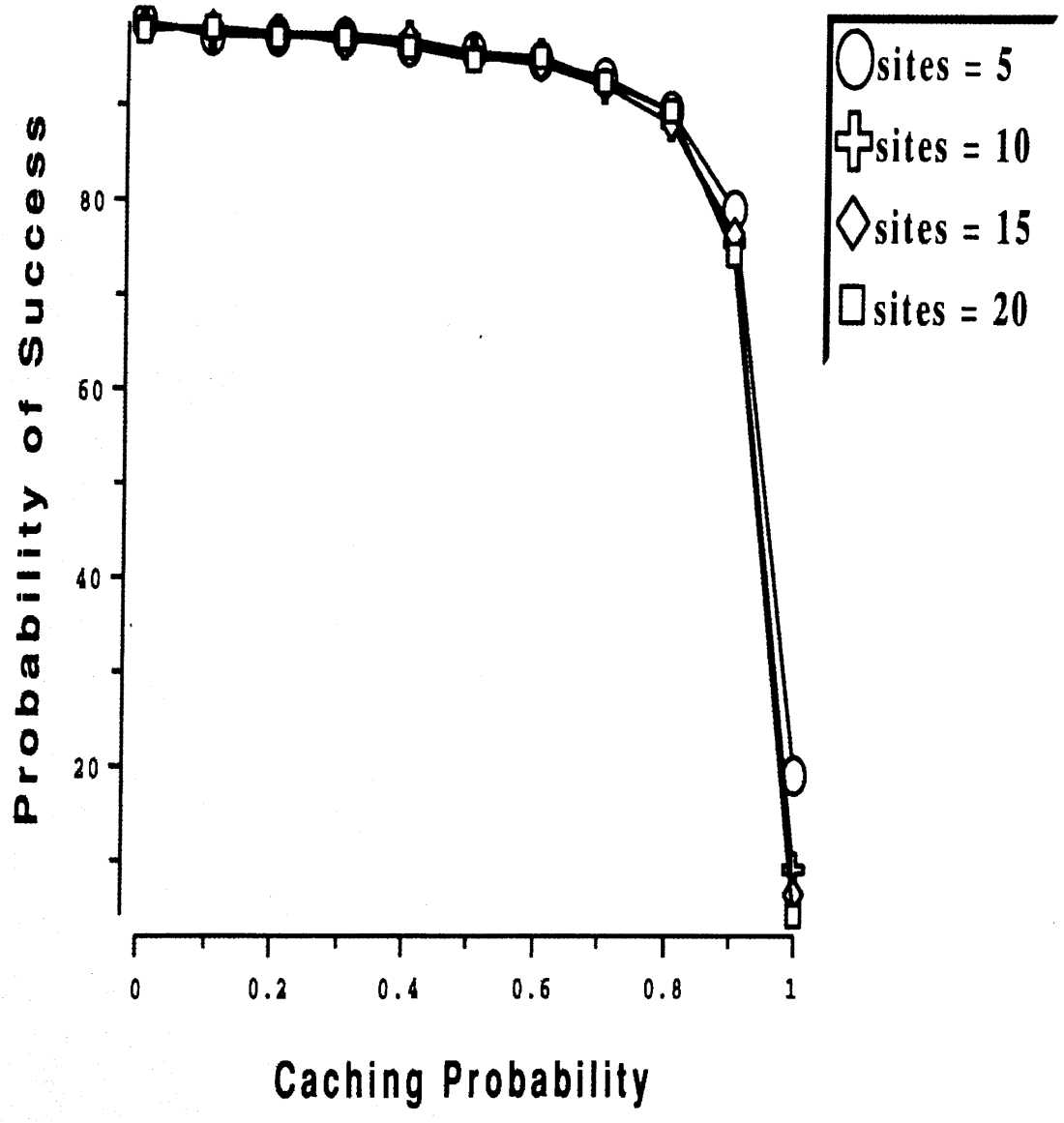


Figure 13. Success Probability vs Caching Probability and the Number of Sites

It may reach a success state due to the failure transition T3 (refer to Figure 8). The observation to be made is that state <1,1> is a successful state, too.

To analyze the effect of the α and ρ on increasing the number of sites, Figure 12 is plotted. The value of λ_f used was 0.02, the simulator was run for 3000 updates. The value of α used was 0.5. The probability of success is lower for higher number of sites. But for the higher values of the ρ, the difference becomes insignificant. The difference is due to the fact that if there are more sites the caching probability increases the chances of caching updates within a larger number of sites, which in turn decreases the probability of success.

Figure 13 shows the effect of varying α on the probability of success for a constant ρ and for different numbers of sites. If the number of sites is increased, the probability of succees decreases when the number of sites in the system increases. Again the same interesting observation can be made that at α equals 1, the probability of success is not zero, for a larger number of sites.

CHAPTER 6

SUMMARY, CONCLUSIONS AND FUTURE WORK

The work presented is summarized in this chapter. Also conclusions and and future enhancements are discussed.

6.1 Summary

The model to reflect the effect of caching of updates, in TSAE protocol is presented. The weak consistency protocol is modeled using a Markov Model. The state transition diagram is discussed in detail in Chapter 3. The transitions from one state to another are classified into three types, namely, T1, T2 and T3. The formulae used to calculate the probability associated with these transitions are derived and presented. The formula to calculate the total number of feasible states (which is dependent on the number of sites in a system) is also derived.

A Monte Carlo simulator is designed and implemented to solve the proposed model. Whenever the system reaches a success state, it is counted as successful propagation of an update to all operational replicas. But if system ends up in a failure state, it means that the update has been lost. In this way, for a specified number of updates, the probability of success or failure can be measured. The simulator can be run in a variety of way to analyze the effect of caching updates.

6.2 Conclusion

It was anticipated that caching of updates will decrease the probability of success. Through the results of simulation the anticipated behaviour of the system is verified.

The probability of success decreases when the caching probability increases. The comparative graph with 0 caching probability and 0.5 probability is drawn to empha-

size the effect of caching updates.

The effect of caching probability (α) on the success probability is analyzed at a fixed relative propagation rate (ρ). The system behaves well for $\alpha < 0.8$, but the probability of success decreases sharply for $\alpha > 0.8$.

The probability of success decreases with an increase in the number of sites. This effect is also verified by the simulation results.

6.3 Future Work

The caching of updates reduces the propagation time. The reason being that instead of several small messages (of updates) one big message can be sent after caching of the updates. This reduces the overhead involved in each anti-entropy sessions. The underlying assumption being that messages are not of very large size. Hence, caching of updates, speeds up the performance by reducing the effective propagation time.

The above mentioned feature of caching of updates is not modeled in the proposed model. A new model can be proposed to depict the effect of caching of updates on the propagation time and the simulator can be developed based on that model.

BIBLIOGRAPHY

[ABM87] Nog Alon, Ambon Bark, and Udi Manber. On disseminating information reliability without broadcasting. In *Proceedings of the 7th International Conference on Distributed Computing Systems*, pages 74–81. IEEE Computer Society, 1987.

[BHG87] P. A. Bernstien, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addision-Wesley Publishing Company, 1987.

[Dea88] Alan Demers and et al. Epidemic algorithms for replicated database maintenance. *Operating Systems Review*, 22:8–32, 1988.

[GL92a] Richard A. Golding and Darrel D. E. Long. The performance of weak-consistency replication protocols. Technical Report UCSC-CRL-92-30, Concurrent Systems Laboratory - Computer and Information Sciences - University of California at Santa Cruz, 1992.

[GL92b] Richard A. Golding and Darrell D. E. Long. Design choices for weak-consistency group communication. Technical Report UCSC-CRL-92-45, Concurrent Systems Laboratory - Computer and Information Sciences - University of California at Santa Cruz, 1992.

[GL93] Richard A. Golding and Darrel D. E. Long. Modeling replica divergence in a weak-consistency protocol for global-scale distributed data bases. Technical Report UCSC-CRL-93-09, Concurrent Systems Laboratory - Computer and Information Sciences - University of California at Santa Cruz, 1993.

[Gol92] Richard A. Golding. A weak-consistency architectures for distributed information services. Technical Report UCSC-CRL-92-31, Concurrent Systems Laboratory - Computer and Information Sciences - University of California at Santa Cruz, 1992.

[GT92] Richard A. Golding and Kim Taylor. Group membership in the epidemic style. Technical Report UCSC-CRL-92-13, Concurrent Systems Laboratory - Computer and Information Sciences - University of California at Santa Cruz, 1992.

[Mac87] M. H. MacDougall. Simulating computer systems. MIT Press, 1987. Random number generator routine.

[Tan89] Andrew S. Tanenbaum. *Computer Networks.* Prentice Hall Inc. Englewood Cliffs NJ, second edition, 1989.

[Tho93] Parkash John Thomas. Partner selection techniques for time stamped anti-entropy protocols. Master's thesis, Department of Computer Science - Oklahoma State University, 1993.

[TS91] J. P. Tremblay and P. G. Sorenson. *An Introduction to Data Structures with Applications.* McGraw Hill, 1991.

APPENDIX A

USER MANUAL

The name of simulator is **sim**. It requires command line arguments and other parametrs from standard input. But standard input can be redirected from a file (if desired).

A.1 Interactive Execution of the Simulator

The command line parameters include the value of λ_f, the number of iterations (for which simulator is to run) and the output filename. The output file is used by the program to store the simulation results. The simulator sends the simulation results to an output file and standard output. Hence, the simulator is invoked using:

$$\text{sim} < \lambda_f >< \text{number of updates} >< \text{output filename} >$$

For $\lambda_f = 0.02$, with 3000 iterations, and an output file named *chk.out*, the simulator is invoked using:

sim 0.02 3000 chk.out

A.2 Arguments From Standard Input

The simulator prompts the user for the parameters it requires. These parameters include:

- `tot_sites_values`: The number of simulations to be performed. For example, if the user only wants a single simulation, then the user should enter 1 (only one n value).

- `tot_sites_array`: Here the user needs to specify the number of sites for each simulation to be performed.

- `alpha_values`: The user enters the number of different α values, to simulate the system for this run. Suppose user wants to use two values: 0.0 and 0.2.

Then user needs to enter 2 (the count of alpha values).

- `alpha_array`: Now, the user needs to enter the values of α. For the above example: 0 0.2

- The count of `rou_values`: The user enters the number of ρ (relative propagation rate $= \lambda_a/\lambda_f$) values, for this simulation. Suppose user wants to run for 8 different ρ values: 0.8 1.0 2.0 4.0 6.0 8.0 10.0 100.0, then the user needs to enter 8.

- `rou_array`: The user needs to enter the ρ values. For the above example: 0.8 1.0 2.0 4.0 6.0 8.0 10 100.0.

Note that this flexibility is provided, so that the user can run the simulator in a number of ways. The simulator can be executed for various combinations of the above parameters in a single run. Refer to SAMPLE.SCR (sample script file) and SAMPLE.DAT (the corresponding output file) for an interactive input example.

A.3 Redirection of Standard Input

The user can redirect standard input from a file. Suppose the file prepared for input redirection is *chk.inp*. The simulator will be invoked using

sim 0.02 3000 chk.out < chk.inp

Refer to **SAMPLE_F.INP** for the format of the input file, **SAMPLE_F.SCR** for the corresponding script file and **SAMPLE_F.DAT** for the corresponding output data file.

APPENDIX B

INSTALLATION GUIDE

The simulator consists of following files:

- `zee_sim.c`: The main simulator program.
- const.h: The constant header file.
- datastrc.h: The header file containing data structure definitions.
- `zee_sim.h`: Constant definitions related to the random number generator.
- README and SAMPLE files

To run the simulator for a larger number of sites, the parameter **MAX_STATES** needs to be changed (the value of it can be calculated using the formula in Chapter 3 for the total number of feasible states). Similarly, to run the simulator for more ρ values, more α values or more site values, the constants **MAX_ROU_COUNT**, **MAX_ALPHA_COUNT**, and **MAX_SITES_COUNT** need to be modified.

Of course, if there is any change made to the header file(s), the program needs to be recompiled.

APPENDIX C

SIMULATOR CODE AND SAMPLE FILES

```
/*************************************************************************/
/*                          File: zee_sim.c                              */
/*                          Main Simulator file                          */
/*                                                                       */
/*************************************************************************/


#include "zee_sim.h"
#include "const.h"
#include "datastrc.h"

/* Function declarations */
int calculate_result();
int print_detailed_result();            /* Prints the detailed result */
int init();
real uniform(real a, real b);
real ranf();
int get_input(int argc, char *argv[]);  /* Gets Input parameter */
int print_input_parameters();           /* Prints the input parameters */
int initialize();                       /* Initialize Data Structures  */
int generate_states();                  /* Generate feasible states    */
int print_state_id();                   /* Prints State_id (m,f)       */
int get_state_index(int m, int f);      /* Returns the state_index     */
int generate_adj_matrix();              /* Generates Adjacency Matrix  */
int print_adj_matrix();                 /* Prints the adjacency Matrix */
int scale_transitions();                /* Scales the transitions out  */
                                        /* of one state                */
int calculate_summary_result();         /* Calculates the summary      */
                                        /* result                      */
int print_summary_result();             /* prints the summaary result  */
int get_rou_array();                    /* gets the rou_array          */
int print_rou_array();                  /* prints the rou_array        */
int get_alpha_array();                  /* gets the alpha_array        */
int print_alpha_array();                /* prints the alpha_array      */
int get_tot_sites_array();              /* gets the total sites array  */
int print_tot_sites_array();            /* prints the alpha_array      */
int print_final_form();                 /* prints in final format      */


/* End of function declarations */



/* Variable Declarations        */
/* Input parameters */
double alpha;                       /* The probabilty of caching */
```

```
double lambda_a;                    /* The global usefule anti-entropy rate */
double lambda_f;                    /* The global failure rate              */
int tot_sites;                      /* Total No. of sites for simulation    */
int end_time;

/* end of paramter variable declarations */

/* Simulation Variables for tracking Transitions in Markov Model */
STATE State[MAX_STATES];         /* States are ordered pair <m, f>,,[]]] */
float adj[MAX_STATES][MAX_STATES]; /* Stores the transition probability
                                      from each state                    */
int num_states;                    /* Total No. of feasible states       */
float state_time[MAX_STATES];       /* Time Spent in each state          */
float rou_array[MAX_ROU_COUNT];     /* The rou_values for this run       */
int rou_count;                      /* The count of rou_values           */
float alpha_array[MAX_ALPHA_COUNT]; /* The rou_values for this run       */
int alpha_count;                    /* The count of rou_values           */
int tot_sites_array[MAX_SITES_COUNT];/* The rou_values for this run      */
int tot_sites_count;                /* The count of rou_values           */

char state[MAX_STATES][MAX_STRING_LEN];
int current_state,next_state,tr_index,row,col;
int count, state_freq[MAX_STATES],link_freq[MAX_LINKS],total;
float current_time,branch_prob[MAX_STATES];
float total_state_time[MAX_STATES];
double prob;
double success_prob_total;
double failure_prob_total;

FILE *outfile;
char outfname[MAX_FNAME];

main(int argc, char *argv[])
{
  int state_index;
  int i;                            /* array index of rou_array          */
  int a;                            /* array index of alpha_array        */
  int s;                            /* array index of tot_sites_array    */

  get_input(argc, argv);
  rou_count = 0;
  alpha_count = 0;
  get_tot_sites_array();
  get_alpha_array();
  get_rou_array();
```

```
  printf("Lambda_f: %4.2f    No. of Updates: %d\n\n",lambda_f, end_time);
  printf("Rou     Success Rate%%     Alpha     Total Sites\n");
  printf("=============================================\n");

  for (s = 0; s < tot_sites_count; s++) {
    tot_sites = tot_sites_array[s];
    for (a = 0;a<alpha_count;a++) {
      alpha = alpha_array[a];
        for (i=0; i<rou_count;i++) {
          lambda_a = rou_array[i] * lambda_f;
          initialize();
          generate_states();
          generate_adj_matrix();
          scale_transitions();
          /* print_adj_matrix(); */
          calculate_result();
          calculate_summary_result();
          print_final_format();
        } /* for i, rou_array */
      } /* for a, alpha array */
    } /* for s, total sites array */
  close(outfile);
} /* end main */


/*************************************************************************/
/*                          Function calculate_result                    */
/*   Calculates the success and failure probability on basis of          */
/*   Uniform distribution                                                */
/*************************************************************************/
int calculate_result()
{
  int i,j;

  current_time = 0.0;
  current_state = 0;
  for (i=0;i<num_states;i++) {
    state_freq[i] = 0;
    total_state_time[i] = 0.0;
   }
  for (i=0;i<MAX_LINKS;i++) link_freq[i] = 0;
  while (current_time < end_time) {
   /* gather statistics */
   state_freq[current_state] += 1;

   /* initialize arrays */
```

```
    branch_prob[0] = adj[current_state][0];
    branch_prob[num_states-1] = 1.0;
    for (i=1;i<num_states-1;i++)
      branch_prob[i] = branch_prob[i-1] + adj[current_state][i];
    /* decide which state to branch to */
    prob = uniform(0.0,1.0);
    count = 0;
    while (branch_prob[count] < (float) prob)
      count++;
    next_state = count;
    current_time += state_time[current_state];
    total_state_time[current_state] += state_time[current_state];
    tr_index = current_state*num_states+next_state;
    link_freq[tr_index] += 1;
    current_state = next_state;
  } /* end while */
} /* calculate_result */
/*****************************************************************************/
/*                              Function: print_detailed_result              */
/*     Prints frequency, %frequency, time and %time                          */
/*****************************************************************************/
print_detailed_result()
{
  int i;
  printf("Markov Model Statistics\n\n");
  printf("State Statistics\n");
  printf("----------------\n");
  total = 0;
  for (i=0;i<num_states;i++)
      total += state_freq[i];
  printf("State   ");printf("Name                ");
  printf("  Freq  ");
  printf("Freq %%   ");
  printf("  Time  ");
  printf(" Time %%\n");
  for (i=0;i<num_states;i++) {
     printf("%-3d     ",i);
     printf("%-20s",state[i]);
     printf("%6d  ",state_freq[i]);
     printf("%5.1f%%   ",(float) state_freq[i] * 100.0/ (float) total);
     printf("%6.1f  ",total_state_time[i]);
     /* printf(" %5.1f%%\n",(float) total_state_time[i]* 100.0/
                          (float) current_time);  */
     printf(" %5.1f%%\n",(float) total_state_time[i]* 100.0/
                          (float) current_time);
  } /* end for */
```

```
  printf("\n\n");
  printf("Link Statistics\n");
  printf("---------------\n");
  total = 0;
  for (i=0;i<MAX_LINKS;i++)
    total += link_freq[i];
  printf("Link     ");printf("Id      ");
  printf("  Freq  ");
  printf("Freq %%\n");
  for (i=0;i<MAX_LINKS;i++) {
    if(link_freq[i] > 0) {
      printf("%-3d     ",i);
      row = i/num_states;
      col = i%num_states;
      printf("%2d -> %2d ",row,col);  printf("%6d  ",link_freq[i]);
      printf("%5.1f%%\n",(float) link_freq[i] * 100.0/ (float) total);
    } /* end if */
  } /* end for */
} /* end print_detailed_result */

/*****************************************************************************/
/*                        Function: scale_transitions                        */
/*     Scales the transitions out of one state to 1                          */
/*     The sum of transition prob. will be equal to 1                        */
/*****************************************************************************/
scale_transitions()
{
  int i,j;
  float t;

  /* Calculate totals and divide matrix */
  /* to create an adjacency matrix */
  for (i=0;i<num_states;i++) {
     t = 0.0;
     for (j=0;j<num_states;j++)
   t += adj[i][j];
     for (j=0;j<num_states;j++) {
       if (t != 0.0)
         adj[i][j] /= t;
     } /* for j */
  }
} /* end scale_transitions */



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

```
/*      These random number generator routines were written by            */
/*  M.H. MacDougall, "Simulating Computer Systems", MIT Press, 1987       */
/* ----------  UNIFORM [0, 1] RANDOM NUMBER GENERATOR ------------        */
/*                                                                        */
/*  function ranf(): Generates a randum number                            */
/**************************************************************************/
real ranf()
  {
    short *p,*q,k;
    long Hi,Lo;
    /* generate product using double precision simulation  (comments  */
    /* refer to In's lower 16 bits as "L", its upper 16 bits as "H")  */
    p=(short *)&In[strm]; Hi= *(p)*A;                    /* 16807*H->Hi */
    *(p)=0; Lo=In[strm]*A;                              /* 16807*L->Lo */
    p=(short *)&Lo; Hi+= *(p);      /* add high-order bits of Lo to Hi */
    q=(short *)&Hi;                        /* low-order bits of Hi->LO */
    *(p)= *(q+1)&0X7FFF;                               /* clear sign bit */
    k= *(q)<<1;
    if (*(q+1)&0X8000) k++;                          /* Hi bits 31-45->K */
    /* form Z + K [- M] (where Z=Lo): presubtract M to avoid overflow */
    Lo-=M;
    Lo+=k;
    if (Lo<0) Lo+=M;
    In[strm]=Lo;
    return((real)Lo*4.656612875E-10);              /* Lo x 1/(2**31-1) */
  } /* end of function:ranf()   */
/**************************************************************************/
/*   function:uniform()                                                   */
/*------------  UNIFORM [a, b] RANDOM VARIATE GENERATOR  -------------*/
/*                                                                        */
/**************************************************************************/
real uniform(a,b)
  real a,b;
    { /* 'uniform' returns a psuedo-random variate from a uniform     */
      /* distribution with lower bound a and upper bound b.           */
      /* if (a>b) then error(0,"uniform Argument Error: a > b"); */
      return(a+(b-a)*ranf());
    } /* end of function:uniform */

/**************************************************************************/
/*                    Function get_input()                                */
/*    Gets input from command line:                                       */
/*               alpha, lambda_a, lamda_f, tot_sites, end_time            */
/**************************************************************************/
int get_input(int argc, char *argv[])
{
```

```
  if (argc !=4) {
    perror("usage:zee_sim <lamda_f> <No. of Updates> <Output File Name>\n");
    exit(1);
  }  /* end of arg_count check */

  /* strip off the parameters from command line */
  lambda_f = atof(argv[1]);
  end_time = atoi(argv[2]);
  strcpy(outfname,argv[3]);
  if ( (outfile = fopen(outfname,"w")) == NULL ) {
    perror("Out File Openning Error\n");
    exit(1);
  }

}

/*************************************************************************/
/*                        Function: print_input_parameters               */
/*     Prints the input parameters for current run                       */
/*************************************************************************/
int print_input_parameters()
{
  printf("The Input parameters for current run is:\n");
  printf("=======================================\n");
  printf(" alpha = %6.4f, lamda_a = %6.4f,",alpha, lambda_a);
  printf(" lamda_f = %6.4f,",lambda_f);
  printf(" Total Sites n = %d, Total Time = %d\n",tot_sites, end_time);

}

/*************************************************************************/
/*                        Function: initialize()                         */
/*    Initializes the data structures                                    */
/*************************************************************************/
int initialize()
{
  memset(adj, '\0', sizeof(adj));
  memset(State, '\0', sizeof(State));
  memset(state_time, '\0', sizeof(state_time));
}
/*************************************************************************/
/*                        Function: generate_states                      */
/*    Generate feasible states, state id: <m,f>                          */
/*    where m is the no. of sites having updates                         */
/*          f is the total available sites                               */
/*************************************************************************/
```

```
int generate_states()
{
  int state_index = -1;
  int m;           /* the no. of sites having updates */
  int f;           /* The total no. of sites having updates */

  num_states = (tot_sites-1) + ( (tot_sites)*(tot_sites+1) )/2;

  /* Generate States label for each feasibel States */

  /* Ist row of States  f = tot_sites, m = 1 to tot_sites */
  f = tot_sites;
  for (m = 1; m<=f;m++) {
    State[++state_index].m = m;
    State[state_index].f = f;
    if ( m==f)
       state_time[state_index] = TIME_IN_FINAL_STATE;
  }

  /* For States, Row from 2 to tot_states */
  for (f = tot_sites -1; f>0; f--) {
    for (m = 0; m <= f; m++) {
      State[++state_index].m = m;
      State[state_index].f = f;
      /* The final states are: SUCCESS: m = f
                               Failure  m = 0 */
      if ( ( m==f) || (m==0) )
        state_time[state_index] = TIME_IN_FINAL_STATE;
    } /* for m */
  }   /* for f */

  /* printf("The current value of state_index should be equal to num_states -1\n")
  printf("The total feasible states, num_states = %d\n",num_states);
  printf("The state_index = %d\n", state_index);
  print_state_id(); */

} /* end of function generate_states */

/*************************************************************************/
/*                        Function: print_state_id                       */
/*     Prints the state id <m,f> of each state                           */
/*************************************************************************/
int print_state_id()
{
   int state_index;
   char open_br = '(';
```

```
   printf("State   ");
   printf("Name                ");
   printf("Time in Each State  \n");
   for (state_index = 0; state_index < num_states; state_index++) {
     printf("%-3d     ",state_index);
 printf("(%d,%d)",State[state_index].m, State[state_index].f);
     printf("%25.1f\n",state_time[state_index]);
   } /* end for state_index */
   /* printf("TEST3\n"); */

} /* end of function print_state_id */

/*************************************************************************/
/*                        Function: get_state_index                      */
/*    Returns the state_index, if a state_id <m.f> is provided           */
/*************************************************************************/
int get_state_index(int m, int f)
{
  int i;
  int t_index = -1;

  for (i=0; i< num_states;i++) {
    if (State[i].m == m)
      if (State[i].f == f) {
        t_index = i;
        break;
      } /* if m */
  } /* for i */

  return (t_index);
} /* End of function get_state_index */

/*************************************************************************/
/*                       Function:generate_adj_matrix                    */
/*     Generates the adjacency matrix, based on alpha, lambda_a, lambda_f  */
/*     and the current state, for which a row is generated               */
/*************************************************************************/
int generate_adj_matrix()
{
  int i;
  int m,f;                  /* The state whose transitions are under */
                            /* considerations                        */
  int nextm,nextf;          /* Next State */
  int state_index;          /* The state index on basis of nextm, nextf */
```

```
  for (i=0;i<num_states;i++) {
    /* ther is no transition from final state (failure or success */
    /* SUCCESS: m = f,     FAILURE m = 0                          */
    if ( (State[i].m != State[i].f) && (State[i].m !=0) ) {
      /* There are possible transitions out of non-final states <m,f> */
      /* T1: to <m+1,f>: useful anti-entropy
       *          (m(f-m)/(f-1)) * (lambda_a *(1-alpha))
       * T2: to <m-1,f-1>: one of the site having update is failed
       *           m*lambda_f
       * T3: to <m,f-1>: one of the site having no update is failed
       *          (f-m)*lambda_f
       */
      m = State[i].m;
      f = State[i].f;

      /* T1: */
      nextm = m+1;
      nextf = f;
      state_index = get_state_index(nextm,nextf);
      adj[i][state_index] = ( m*(f-m)/(f-1) ) * ( lambda_a * (1-alpha) );

      /* T2: */
      nextm = m-1;
      nextf = f-1;
      state_index = get_state_index(nextm,nextf);
      adj[i][state_index] = m*lambda_f;

      /* rem maybe int to float conv. for multiplications */

      /* T3: */
      nextm = m;
      nextf = f-1;
      state_index = get_state_index(nextm,nextf);
      adj[i][state_index] = (f-m) * lambda_f;
    } /* if State[i].m */
    else {
      /* Final State, the probability of self transition = 1*/
      adj[i][0] = 1.0;
    } /* else */

  } /* for i */
} /* End of function generate_adj_matrix */

/*************************************************************************/
/*                      Function: print_adj_matrix                       */
/*     Prints the adjacency matrix                                       */
```

```
/*************************************************************************/
print_adj_matrix()
{
  int i,j;
  printf("The tranistion probability Matrix (adj_matirx\n");
  printf("=========================================\n");
  printf("St   Matrix\n");
  for (i=0;i<num_states;i++) {
    printf("%d ",i);
    for (j=0;j<num_states;j++) {
        printf("%5.3f ",adj[i][j]);
    } /* for j */
    printf("\n");
  } /*  for i */
} /* End of function print_adj_matrix */

/*************************************************************************/
/*                      Function: calculate_summary_result               */
/*    Calculate_summary_result, The %Success Probability, %Failure       */
/*    Probability                                                        */
/*************************************************************************/
calculate_summary_result()
{
   int i;

   success_prob_total = failure_prob_total = 0.0;
   for (i=0; i<num_states;i++) {
     if (State[i].m == 0)
       /* The Failure FInal State has m=0 */
       failure_prob_total = failure_prob_total + total_state_time[i];
     if (State[i].m == State[i].f)
       /* Success state has m==f */
       success_prob_total = success_prob_total + total_state_time[i];
   } /* end for i */
} /* end function calculate_summary_result */


/*************************************************************************/
/*                      Function: print_summary_result                   */
/*    Calculate_summary_result, The %Success Probability, %Failure       */
/*    Probability                                                        */
/*************************************************************************/
print_summary_result()
{

  double per_success_prob, per_failure_prob;
```

```
  per_success_prob = success_prob_total * 100.0/(float)end_time;
  per_failure_prob = failure_prob_total * 100.0/(float)end_time;

  printf("Row = %3.1f, ",lambda_a/lambda_f);
  printf("%Success_probability = %5.2f%%", per_success_prob);
  printf(", %Failure_probability = %5.2f%%\n", per_failure_prob);
} /* end function print_summary_result */

/*****************************************************************************/
/*                          Function:get_rou_array                           */
/*    Get the count of rou and the values of it                              */
/*****************************************************************************/
get_rou_array()
{
  int i;
  printf("The count of the rou_values\n");
  scanf("%d",&rou_count);
  printf("Enter the values of rou_array\n");
  for (i=0;i<rou_count;i++)
    scanf("%f",&rou_array[i]);
} /* end of function get_rou_array */

/*****************************************************************************/
/*                         Function:print_rou_array                          */
/*    Prints the rou_array                                                   */
/*****************************************************************************/
print_rou_array()
{
  int i;

  printf("The rou_count is: %d\n",rou_count);
  for (i=0; i<rou_count;i++)
    printf("%4.1f\n",rou_array[i]);
  printf("\n");
}

/*****************************************************************************/
/*                          Function:get_alpha_array                         */
/*    Get the count of alpha and the values of it                            */
/*****************************************************************************/
get_alpha_array()
{
  int i;
  printf("The count of the alpha_values\n");
  scanf("%d",&alpha_count);
```

```
  printf("Enter the values of alpha_array\n");
  for (i=0;i<alpha_count;i++)
    scanf("%f",&alpha_array[i]);
} /* end of function get_rou_array */

/*****************************************************************************/
/*                        Function:print_alpha_array                         */
/*    Prints the alpha_array                                                 */
/*****************************************************************************/
print_alpha_array()
{
  int i;

  printf("The alpha_count is: %d\n",alpha_count);
  for (i=0; i<alpha_count;i++)
    printf("%4.1f\n",alpha_array[i]);
  printf("\n");
}

/*****************************************************************************/
/*                         Function:get_tot_sites_array                      */
/*    Get the count of total sites and the values of it                      */
/*****************************************************************************/
get_tot_sites_array()
{
  int i;
  printf("The count of the tot_sites_values\n");
  scanf("%d",&tot_sites_count);
  printf("Enter the values of tot_sites_array\n");
  for (i=0;i<tot_sites_count;i++)
    scanf("%d",&tot_sites_array[i]);
} /* end of function get_rou_array */

/*****************************************************************************/
/*                        Function:print_alpha_array                         */
/*    Prints the alpha_array                                                 */
/*****************************************************************************/
print_tot_sites_array()
{
  int i;

  printf("The tot_sites_count is: %d\n",tot_sites_count);
  for (i=0; i<tot_sites_count;i++)
    printf("%d\n",tot_sites_array[i]);
  printf("\n");
}
```

```
/*****************************************************************************/
/*                        Function: print_final_format                       */
/*    Calculate_summary_result, The %Success Probability, %Failure           */
/*    Probability                                                            */
/*****************************************************************************/
print_final_format()
{

  double per_success_prob, per_failure_prob;

  per_success_prob = success_prob_total * 100.0/(float)end_time;
  per_failure_prob = failure_prob_total * 100.0/(float)end_time;

  printf("%5.1f        %5.2f", lambda_a/lambda_f, per_success_prob);
  printf("       %5.2f             %d\n", alpha, tot_sites);
  /* Printing in output file */
  fprintf(outfile,"%5.1f        %5.2f", lambda_a/lambda_f, per_success_prob);
  fprintf(outfile,"       %5.2f            %d\n", alpha, tot_sites);
} /* end function print_summary_result */



/*****************************************************************************/
/*          File: datastrc.h                                                 */
/*          Structure for state of network                                   */
/*          State of network is <m,f>                                        */
/*                                                                           */
/*****************************************************************************/
typedef struct {
  int    m;            /* The no. of sites having update  */
  int    f;            /* Total available (working) sites */
} STATE;



/*****************************************************************************/
/*                    File: const.h                                          */
/*                    Constants declaration                                  */
/*****************************************************************************/
#define MAX_STATES 100
#define MAX_LINKS MAX_STATES*MAX_STATES
#define MAX_STRING_LEN 20
#define TIME_IN_FINAL_STATE 1
#define MAX_ROU_COUNT 10
#define MAX_ALPHA_COUNT 15
```

```
#define MAX_SITES_COUNT 10
#define MAX_FNAME 20


/*****************************************************************************/
/*          File: zee_sim.h                                                  */
/*          contains include files for zee_sim.c                             */
/*          and initialization for random number                             */
/*          generators.                                                      */
/*****************************************************************************/
#include <malloc.h>
#include <stdio.h>
#include <math.h>


/* For Uniform function */
#define A 16807L            /* multiplier (7**5) for 'ranf' */
#define M 2147483647L       /* modulus (2**31-1) for 'ranf' */

static long In[16]= {0L,    /* seeds for streams 1 thru 15  */
  1973272912L,  747177549L,   20464843L,  640830765L, 1098742207L,
    78126602L,   84743774L,  831312807L,  124667236L, 1172177002L,
  1124933064L, 1223960546L, 1878892440L, 1449793615L,  553303732L};

static int strm=1;          /* index of current stream */



typedef double real;

/* end of declaration for uniform functions */


/*  SAMPLE.DAT file  */

  0.8        47.43        0.20           3
  1.0        51.13        0.20           3
  2.0        63.07        0.20           3
  4.0        75.17        0.20           3
  6.0        80.37        0.20           3
  8.0        85.17        0.20           3
 10.0        87.87        0.20           3
100.0        98.57        0.20           3

/* SAMPLE.SCR file */
```

```
Script started on Tue Mar  8 11:44:24 1994
$ sim 0.02 3000 SAMPLE.DAT

The count of the tot_sites_values

1

Enter the values of tot_sites_array

3

The count of the alpha_values

1

Enter the values of alpha_array

0.2

The count of the rou_values

8

Enter the values of rou_array

0.8 1 2 4 6 8 10 100

Lambda_f: 0.02    No. of Updates: 3000
```

Rou	Success Rate%	Alpha	Total Sites
0.8	47.43	0.20	3
1.0	51.13	0.20	3
2.0	63.07	0.20	3
4.0	75.17	0.20	3
6.0	80.37	0.20	3
8.0	85.17	0.20	3

```
 10.0        87.87        0.20            3

100.0        98.57        0.20            3

$ exit


script done on Tue Mar  8 11:45:30 1994

/* SAMPLE_F.DAT file */

  0.8        47.43        0.20            3
  1.0        51.13        0.20            3
  2.0        63.07        0.20            3
  4.0        75.17        0.20            3
  6.0        80.37        0.20            3
  8.0        85.17        0.20            3
 10.0        87.87        0.20            3
100.0        98.57        0.20            3

/* SAMPLE_F.INP file */

1
3
1
0.2
8
0.8 1 2 4  6 8 10 100

/* SAMPLE_F.SCR file */

Script started on Tue Mar  8 11:40:56 1994
$ sim 0.02 3000 SAMPLE_F.DAT < SAMPLE_F.INP

The count of the tot_sites_values

Enter the values of tot_sites_array

The count of the alpha_values

Enter the values of alpha_array

The count of the rou_values

Enter the values of rou_array
```

```
Lambda_f: 0.02    No. of Updates: 3000


Rou     Success Rate%     Alpha     Total Sites

================================================

  0.8         47.43         0.20             3

  1.0         51.13         0.20             3

  2.0         63.07         0.20             3

  4.0         75.17         0.20             3

  6.0         80.37         0.20             3

  8.0         85.17         0.20             3

 10.0         87.87         0.20             3

100.0         98.57         0.20             3

$ exit

script done on Tue Mar  8 11:42:32 1994
```

VITA

Zeeshan Shafaq Syed

Candidate for the Degree of

Master of Science

Thesis: MODELING UPDATE CACHING IN WEAK CONSISTENCY PROTOCOLS

Major Field: Computer Science

Biographical Data:

Personal Data: Born in SaidPur (former East Pakistan), Pakistan, on February 19, 1969, the son of Syed Muhammad Rafique and Safia Rafique.

Education: Graduated from Govt. Dehli Boys Secondary School (Secondary School Certificate), Karachi, Pakistan, in 1982; graduated from Adamjee Govt. Science College (Higher School Certificate), Karachi, Pakistan, in 1984; graduated from N. E. D. University Of Engineering and Technology, Karachi, Pakistan, October 1990, received Bachelor of Engineering in Computer Systems Engineering. Completed the requirements for the Master of Science degree with a major in Computer Science at Oklahoma State University in July 1994.

Experience: Programmer/Graduate Research Assistant, International Student Services Department, Oklahoma State University, September, 1992 to May 1994. Programmer, Agriculture Economics Department, Oklahoma State University, February 1992, May 1992. Systems Analyst, Information System Department, Fazal Textile Mills Limited, Karachi, Pakistan, September 1991 to December 1991. Computer Engineer, Ericsson Telecom, Karachi, Pakistan, October 1990 to August 1991. Analyst Programmer, Information Systems Department, Fazal Textile Mills Limited, Karachi, Pakistan, May 1990 to September 1990).