Performance evaluation of distributed mutual exclusion algorithms

Kenneth B Been
University of Nevada, Las Vegas

Follow this and additional works at: https://digitalscholarship.unlv.edu/rtds

Repository Citation
https://digitalscholarship.unlv.edu/rtds/280

This Thesis is brought to you for free and open access by Digital Scholarship@UNLV. It has been accepted for inclusion in UNLV Retrospective Theses & Dissertations by an authorized administrator of Digital Scholarship@UNLV. For more information, please contact digitalscholarship@unlv.edu.
INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.
Performance evaluation of distributed mutual exclusion algorithms

Been, Kenneth B., M.S.
University of Nevada, Las Vegas, 1993
Performance Evaluation of Distributed
Mutual Exclusion Algorithms

Kenneth B. Been

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in
Computer Science

Department of Computer Science
University of Nevada, Las Vegas

May 1993
The Thesis of Kenneth B. Been, for the degree of Master of Science in Computer Science, is approved.

Chairperson, Kia Makki, Ph.D.

Examiner Committee Member, John Minor, Ph.D.

Examiner Committee Member, Roy Ogawa, Ph.D.

Examiner Committee Member, Niki Pissinou, Ph.D.

Graduate Faculty Representative, Moses Karakouzian, Ph.D.

Dean of the Graduate College, Ronald Smith, Ph.D.

University of Nevada, Las Vegas
May 1993
Abstract

In any system in which concurrent processes share resources it may be necessary to guarantee the integrity of those resources by restricting their use to one process at a time. This is known as the mutual exclusion problem. The problem of providing for mutual exclusion in a distributed system is a non-trivial one which has inspired a variety of solutions. Due to the complex nature of distributed mutual exclusion algorithms, it is difficult to mathematically analyze their performance. Furthermore, experimental performance evaluation is important because when theoretical analyses are possible, they might not fully reflect how an algorithm would perform in practice. Little work has been done in analyzing, testing, and comparing solutions with respect to performance, correctness, and domain of applicability. This thesis investigates seven well known distributed mutual exclusion algorithms in detail, and uses computer simulation to evaluate the performance and applicability of these various algorithms. Toward this end, a realistic and general model for evaluating distributed algorithms is proposed. Results of the experiments include the discovery of starvation and deadlock problems in two algorithms, the identification of one algorithm as the best performer in a general network in which sites do not fail, and experimental performance analysis of one algorithm which accommodates site failures.
Contents

Abstract iii
List of Figures vi
List of Tables viii
Acknowledgements ix

Chapter 1 Introduction 1
  1.1 Preliminaries ................................................................. 1
  1.2 Research Context ............................................................ 3
    1.2.1 The Primary Site Approach ...................................... 4
    1.2.2 Permission Based Algorithms .................................. 5
    1.2.3 Token Based Algorithms .......................................... 7
    1.2.4 Fault Tolerant Algorithms ....................................... 8
  1.3 Thesis Organization ....................................................... 8

Chapter 2 Representative Token Based, Non-Fault Tolerant, Distributed Mutual Exclusion Algorithms 10
  2.1 Algorithm SK ............................................................. 11
  2.2 Algorithm CSL ........................................................... 11
  2.3 Algorithm MBBO .......................................................... 12
  2.4 Algorithm NT ............................................................. 14
  2.5 Algorithm CSL-2 ......................................................... 15

Chapter 3 Representative Fault Tolerant Distributed Mutual Exclusion Algorithms 18
  3.1 Algorithm NLM ............................................................ 20
  3.2 Algorithm AA ............................................................. 22
  3.3 Implementation Considerations ....................................... 24
    3.3.1 Timer Settings ....................................................... 25
    3.3.2 Implementing AA .................................................. 28
List of Figures

Figure 3.1 Attaining a Quorum in Algorithm AA ................................. 23
Figure 3.2 Logical tree imposed on a network ................................. 29

Figure 5.1 An Example of Starvation with CSL-2 ......................... 40
Figure 5.2 An Example of Deadlock with AA ................................. 42

Figure 6.1 Message Traffic vs. \( \lambda \) (\( N = 21 \)) .................. 47
Figure 6.2 Average Waiting Time vs. \( \lambda \) (\( N = 21 \)) ................. 49
Figure 6.3 Throughput vs. \( \lambda \) (\( N = 21 \)) .......................... 50
Figure 6.4 Expected Maximum Waiting Time vs. \( \lambda \) (\( N = 21 \)) .... 50
Figure 6.5 Messages per Critical Section vs. \( N, \lambda = 0.2 \) .......... 51
Figure 6.6 Messages per Critical Section vs. \( N, \lambda = 0.06 \) ..... 52
Figure 6.7 Messages per Critical Section vs. \( N, \lambda = 0.001 \) .... 52
Figure 6.8 Average Waiting Time vs. \( N, \lambda = 0.2 \) ..................... 53
Figure 6.9 Average Waiting Time vs. \( N, \lambda = 0.06 \) ..................... 53
Figure 6.10 Average Waiting Time vs. \( N, \lambda = 0.001 \) .............. 54
Figure 6.11 Throughput vs. \( N, \lambda = 0.2 \) .............................. 55
Figure 6.12 Throughput vs. \( N, \lambda = 0.06 \) ............................. 56
Figure 6.13 Throughput vs. \( N, \lambda = 0.001 \) .............................. 56
Figure 6.14 Expected Maximum Waiting Time vs. \( N, \lambda = 0.2 \) ... 57
Figure 6.15 Expected Maximum Waiting Time vs. \( N, \lambda = 0.06 \) .. 57
Figure 6.16 Expected Maximum Waiting Time vs. \( N, \lambda = 0.001 \) .. 58
Figure 6.17 Messages per Critical Section vs. \( \lambda \), Algorithm NLM, \( N = 21 \), \( \beta = 1/2000 \) ................................................................. 62
Figure 6.18 Average Waiting Time vs. \( \lambda \), Algorithm NLM, \( N = 21 \), \( \beta = 1/2000 \) ......................................................... 63
Figure 6.19 Expected Maximum Waiting Time vs. \( \lambda \), Algorithm NLM, \( N = 21 \), \( \beta = 1/2000 \) ......................................................... 63
Figure 6.20 Throughput vs. \( \lambda \), Algorithm NLM, \( N = 21 \), \( \beta = 1/2000 \) ......................................................... 64
Figure 6.21 Messages per Critical Section vs. \( N \), Algorithm NLM, \( \beta = 1/2000 \) ......................................................... 65
Figure 6.22 Average Waiting Time vs. \( N \), Algorithm NLM, \( \beta = 1/2000 \) ......................................................... 65
Figure 6.23 Expected Maximum Waiting Time vs. \( N \), Algorithm NLM, \( \beta = 1/2000 \) ......................................................... 66
Figure 6.24 Throughput vs. \( N \), Algorithm NLM, \( \beta = 1/2000 \) ........................................................................ 66
Figure 6.25 Messages per Critical Section vs. $1/\beta$, Algorithm NLM, $N = 21$ 68
Figure 6.26 Messages per Critical Section vs. $1/\beta$, Algorithm NLM, $N = 50$ 68
Figure 6.27 Average Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 21$ 69
Figure 6.28 Average Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 50$ 69
Figure 6.29 Expected Maximum Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 21$ 70
Figure 6.30 Expected Maximum Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 50$ 70
Figure 6.31 Throughput vs. $1/\beta$, Algorithm NLM, $N = 21$ 71
Figure 6.32 Throughput vs. $1/\beta$, Algorithm NLM, $N = 50$ 71
List of Tables

Table 2.1 Maximum Request Forwards, Algorithm MBBO, $N = 21$ .............................................. 14
Table 4.1 Simulation Model Parameters ......................................................................................... 33
Table 6.1 Statistics Measured ........................................................................................................ 45
Table 6.2 Performance Summary .................................................................................................... 59
Acknowledgements

Kia Makki and Niki Pissinou have spent many hours guiding me into the labyrinth of computer science research. Their support has been irreplaceable.

I would also like to thank my parents, Irwin and Ruth Been, who taught me to be open-minded, inquisitive, and honest.

Finally, thanks go to the examining committee members for reviewing and criticizing the work.
Chapter 1

Introduction

In any system in which concurrent processes share resources it may be necessary to guarantee the integrity of those resources by restricting their use to one process at a time. This is known as the mutual exclusion problem. In this thesis we are concerned with the problem of providing mutual exclusion in a distributed system. Distributed mutual exclusion has a wide variety of applications, such as distributed operating systems and distributed databases[13, 22].

1.1 Preliminaries

In this thesis distributed system will refer to a collection of autonomous sites (processors) which may communicate with each other only by sending messages. Sites do not share a common memory or clock. Message transmission is relatively slow, particularly if communication lines become congested. In such an environment, two
criteria emerge for evaluating performance of algorithms: (1) number of messages per critical section, and (2) delay in granting critical section. Of course the solution must be free from starvation and deadlock, and, as will be seen in chapter 5, some of the algorithms that have been published fail to meet these requirements.

Although many solutions have been proposed for the distributed mutual exclusion problem,¹ little work has been done in performance evaluation. Ricart and Agrawala[17] have implemented a comprehensive simulation, but have studied only one algorithm that they propose. Chang, et al.[3, 5], performed simulation experiments involving several algorithms, but made some unrealistic assumptions, such as constant message transmission time.

Experimental performance evaluation is important because theoretical analyses may not reflect how an algorithm will perform in practice. For example, an algorithm which organizes the network into a dynamic, logical tree may claim to have $O(\log N)$ message cost since the message traffic depends on the height of the tree. But if the tree degenerates into a linear list, or if the tree never exceeds constant height in practice, the performance may in fact be $O(N)$ or $O(1)$. Furthermore, each published algorithm may contain different assumptions about the network; for example, some algorithms require that messages between two sites are delivered in the order in which they are sent, while other algorithms relax this restriction.

The goal of the current research is twofold: first, to develop a realistic, general,

¹see [3] for a good survey
and usable simulation model for investigating the performance of algorithms; and second, to compare the performance of specific algorithms [5, 12, 14, 23, 6, 1, 15] within this common model. The algorithms we study are well known representatives from two important overlapping classes of solutions: token based and fault tolerant.

1.2 Research Context

Before delving into the details of the algorithms experimentally analyzed in this thesis, it is enlightening to see how those algorithms compare qualitatively with other distributed mutual exclusion algorithms. This chapter discusses three approaches to providing distributed mutual exclusion, and introduces some ideas that will be used in the performance analysis in chapter 6.

The field of distributed algorithms is interesting in that it combines simplicity with complexity. While algorithm performance and correctness may be mathematically intractable, and may vary widely with changing distributed environments, the ideas which underly an algorithm will often be simple and intuitive. This chapter covers the intuitive side of the field.

Most distributed mutual exclusion algorithms fall into one of two categories: permission based or token based. The permission based algorithms work on the principle that if enough sites give the o.k. to a particular site, then that site may enter its critical section knowing that no other site will do so at the same time. In the token based algorithms, the token is a unique and singular message which functions as a kind of
"key" that unlocks critical sections. This token is passed from site to site, and only a site which possesses it may enter its critical section.

The majority of algorithms proposed to date have assumed a failure-free network; neither sites nor communication lines are assumed to ever fail. Those algorithms that do provide fault tolerance also fall into one of the two classes of solutions discussed above, but they involve significant unique features, and will be considered separately in this thesis.

Before considering these major classes of solutions, however, we'll look at one solution which is simple, popular, and highly intuitive. Discussion of this solution and its problems will illustrate many of the important considerations that arise in distributed mutual exclusion.

1.2.1 The Primary Site Approach

Perhaps the simplest way to provide for distributed mutual exclusion is to appoint one site, $S_0$, as the arbiter for the system[2]. Whenever a site $S_i$ wishes to execute its critical section (i.e., that section of code which requires that during its execution all other sites must be excluded from executing their critical sections) it must send a request message to $S_0$. $S_0$ will maintain a queue of requests, and when $S_i$'s turn comes around $S_0$ will send it a permission message allowing $S_i$ to execute its critical section. When $S_i$ is finished it must send a release message back to $S_0$.

This solution is highly efficient in that it requires only three messages per critical
section execution, regardless of the size of the system or the density of requests. However, it has several drawbacks. First, the arbiter site, $S_0$, is unfairly burdened with responsibility and with message traffic. This solution might even be seen as not truly “distributed”. Second, the algorithm is highly vulnerable to failures at $S_0$. Indeed, the failure of a single site, $S_0$, causes the entire system to go down, as far as accessing critical sections goes, despite the fact that every other site in the system is functioning. This defeats one of the main purposes of having a distributed system in the first place.

A third drawback with this algorithm is subtle, but illustrative. Notice that between any two critical section executions two messages must be sent consecutively. If we assume that the time required to execute a critical section is negligible when compared with the time required to transmit a message, and that each message takes one time unit to transmit, then the throughput of the system, which is defined as the number of critical sections executed per unit time, cannot exceed 0.5. We’ll see later that in some algorithms throughput approaches 1.0. Simply put, this means that in the primary site algorithm sites tend to wait longer for their critical section requests to be granted.

### 1.2.2 Permission Based Algorithms

Sanders [20] has developed a general formalism to describe the permission based algorithms [1, 9, 10, 18, 20, 21]. These typically require that a site which wants to
execute its critical section send request messages to some subset of the sites in the system—its request set. The requesting site may enter its critical section as soon as it receives a reply to each one of its requests, and must send a release message to each site in its request set when it is finished, allowing them to permit other sites to enter the critical section. Typically, each request will have a sequence number, or timestamp as described by Lamport [9], and this information will be used by sites to create a logical, consistent, ordering of the requests. The timestamp concept assumes that events within each process are ordered in terms of time, but the order of two events occurring at separate sites may not be known, since there is no shared clock. Lamport’s timestamping technique alleviates this problem; see [9] for more details.

In the permission based algorithms, a site which is not currently requesting will send replies to all requests that it receives; a site which is requesting will immediately send replies to those requests which are timestamped before its own request, and will delay replies to other sites until after it finishes with its own critical section.

These algorithms can be characterized by the nature of the request sets. (Note that the primary site approach is essentially a permission based algorithm in which every request set contains exactly one site, the arbiter site.) Lamport [9] described an algorithm in which each request set is the entire network. Then, if $N$ is the number of sites in the system, and if self-messages are not counted, the algorithm requires $N - 1$ each of requests, replies, and releases; or $3(N - 1)$ messages per critical section execution. Ricart and Agrawala [18] realized that if all sites must grant permission
by sending replies, then the release messages are superfluous, since a reply involves an implicit release. Therefore, they have reduced the number of messages in Lamport's algorithm to $2(N - 1)$. Others [1, 10, 21] have found ways to group the sites so that each request set only needs to be a proper subset of the sites in the system. An algorithm which defines request sets in such a way that every set intersects with at least one other set can be a valid mutual exclusion algorithm. Sanders [20], and Garcia-Molina and Barbara [8], have generalized and formalized this idea. An example algorithm is that of Maekawa [10], which uses finite projective planes to group the sites. This algorithm requires between $3\sqrt{N}$ and $5\sqrt{N}$ messages per critical section entry. This algorithm is not deadlock free; after the occurrence of a deadlock the algorithm must send more messages in order to recover from the deadlock. This problem was later corrected by Sanders [20] and by Chang and Singhal [4].

1.2.3 Token Based Algorithms

In the token based algorithms [5, 11, 12, 14, 15, 16, 23] the token is a unique and singular message which circulates among the sites. Only the site which possesses the token may enter its critical section. The various token based algorithms are distinguished by the methods for determining how a site obtains the token, and where a site sends the token when it is finished with its critical section. In recent years some very efficient token based algorithms have been proposed; and, although the question is still open, it now appears that the best token based algorithms are more efficient
than any of the permission based algorithms. The token based algorithms will be examined in detail in chapter 2.

1.2.4 Fault Tolerant Algorithms

The solutions discussed so far have all been designed to operate in an ideal environment in which sites do not fail. Making an algorithm resilient to site failures is certainly a non-trivial assignment. In the permission based algorithms, when a site is down, sites which have the failed site in their request sets will also become non-functioning, in so far as their critical section needs are concerned, for the duration of the failure. In the token based algorithms the token may get lost if it is sent to a failed site, or if the site holding it fails.

In contrast to the plethora of mutual exclusion algorithms that have been proposed, there is a paucity of algorithms that are truly fault tolerant, in the sense that the system can continue to function while some sites are down. These algorithms employ dynamically changeable request sets, in the case of permission based algorithms [1]; and token regeneration schemes, or elections, in the case of token based algorithms [7, 15]. Specifics about these approaches will be discussed in chapter 3.

1.3 Thesis Organization

The remainder of this thesis is organized as follows. The representative token based, non-fault-tolerant algorithms are described in chapter 2. Two important fault tolerant
algorithms are described in chapter 3. Descriptions of the model distributed system and simulation program, are given in chapter 4. Chapter 5 contains an exposition of incorrectness of two published algorithms. The results of the performance experiments are in chapter 6. The final chapter contains some concluding remarks.
Chapter 2

Representative Token Based, Non-Fault Tolerant, Distributed Mutual Exclusion Algorithms

Four of the token based algorithms analyzed in this thesis include a queue as part of the token message. This queue contains the identifiers of the sites which will be receiving the token in the near future. When a site is finished with the token it passes it to the next site on the queue. A site captures the token by getting its id put on the token queue; and it does that by sending a message to a site which either has the token or will have it in the near future. The final algorithm, CSL-2, also has a kind of token queue, but it is distributed over the sites in the network. This chapter describes the five well known token based distributed mutual exclusion algorithms.
2.1 Algorithm SK

Algorithm SK, proposed by Suzuki and Kasami [23], specifies that a site, \( S_i \), which desires the token, must send requests to all other sites in the system. This guarantees that the site possessing the token will receive a request message and will enqueue \( S_i \) onto the token queue. Each request is labeled with a sequence number, and the token message also contains an array with the sequence numbers of the last request satisfied at each site. This is so that late arriving messages, i.e. requests which arrive after the requesting site has been serviced, can be ignored. For each critical section entry \( N - 1 \) request messages are sent and the token is passed once. Thus, algorithm SK requires \( N \) messages per critical section execution.

2.2 Algorithm CSL

Algorithm CSL, proposed by Chang, Singhal, and Liu [5], improves on algorithm SK by noticing that a requesting site does not need to send requests to every other site in the system in order to get itself onto the token queue. In this algorithm, each site maintains a request set which contains identifiers for those sites which should be receiving the token in the near future. A requesting site must, therefore, only send request messages to those sites in its request set. Initially every request set contains every other site in the system. However, when a site receives the token it reduces its request set to only those sites which are on the token queue. It then adds to
its request set those sites from which it receives request messages. Here, as with SK, sequence numbers are used to detect out of date requests. In the worst case of heavy traffic of token requests, the token-queue will usually be full, the request sets will usually be full, and the algorithm will not provide any message traffic savings over SK. However, when token requests are relatively infrequent, CSL may provide significant savings. In fact, previous experiments [4] have shown that this mutual exclusion algorithm can reduce message traffic of Suzuki et al.'s algorithm by 40% in light traffic, but needs $N$ messages in heavy traffic.

We'll now examine three algorithms which require a site to send only one request message to capture the token. Ideally, we would like to send only two messages per critical section entry—one for a site to request permission, and one for the permission to be granted. The next three algorithms attempt to achieve this optimal performance. Furthermore, since only one request message is sent, no request message may become superfluous or outdated, and the sequence numbers are not necessary.

### 2.3 Algorithm MBBO

Algorithm MBBO, proposed by Makki, Banta, Been, and Ogawa [12], has a requesting site send one request message to that site which it believes is the last site on the token-queue. When a site possesses the token it may simply examine the queue to see what the last site is, and store that information. Those sites which may not have had the token recently, or will not be getting it soon, must periodically be updated as
to which is the last site on the queue. It is these update messages which cause the
algorithm to perform suboptimally. When the updates are to be sent out, the site
with the token sends updates to every site that is not on the token queue. It then
flags the last site on the queue as the next site to send updates.

When token request traffic is heavy, the queue will generally be full and few
updates will have to be sent; in this case MBBO approaches optimal performance.
When token request traffic is light, however, the token queue will generally have one
or fewer sites on it, and updates must be sent to almost all the sites in the system
upon almost every critical section entry. In this case performance degrades to \( N \)
messages per critical section. Makki, et al.[12], give a theoretical argument that the
number of messages will be inversely proportional to the traffic of requests, and our
simulation confirms this result.

There is another difficulty with algorithm MBBO that must be examined here.
It is possible that a request message from \( S_i \) to its “last” site \( S_j \) will arrive after \( S_j \)
has had the token and sent it on. Then \( S_j \) will not be able to put \( S_i \) onto the queue.
This problem is solved by having \( S_j \), after sending out the update messages, wait for
an amount of time equal to twice the maximum message transmission time of the
system before sending on the token. This guarantees that all requests will catch the
token, but it requires that the maximum message transmission time of the system be
known.

In this thesis, in order to study the general case of unpredictable message times,
<table>
<thead>
<tr>
<th>Traffic Volume</th>
<th># forwards</th>
</tr>
</thead>
<tbody>
<tr>
<td>light request traffic</td>
<td>0.8 ± 0.1</td>
</tr>
<tr>
<td>medium request traffic</td>
<td>2.1 ± 0.2</td>
</tr>
<tr>
<td>heavy request traffic</td>
<td>1.0 ± 0.0</td>
</tr>
</tbody>
</table>

Table 2.1: Maximum Request Forwards, Algorithm MBBO, \( N = 21 \)

MBBO has been modified so that the “wait” site only waits for an amount of time equal to twice the average transmission time of the system. Now, any message which arrives at \( S_j \) after it has sent the token on will be forwarded to \( S_j \)’s “last” site if \( S_j \) is not itself requesting. This modified algorithm is more general since the maximum message time of the system need not be known, but it introduces the possibility for a request to be forwarded repeatedly and never catch the token. The simulation, however, indicates that the probability of a request being forwarded more than a few times is negligible. See Table 2.1 for some sample values of the expected maximum number of forwards for a given request in one run of the simulation.

### 2.4 Algorithm NT

Algorithm NT, proposed by Naimi and Trehel [14], organizes the sites into a dynamic, logical, rooted tree. Each site, \( S_i \), maintains only a portion of the tree—the id of its parent, which is the last site to have sent a request to \( S_i \). Initially, one site possesses the token, and it is the parent of all other sites. If an edge of the tree leads from \( S_i \) to
$S_j$, then $S_j$ was the last site to send a request to $S_i$, and $S_i$ will send its next request to $S_j$ since $S_i$ knows that $S_j$ will be getting the token soon. If $S_j$ is expecting the token it will save $S_i$'s request on a local queue—to be appended to the token queue when the token arrives. If $S_j$ is not expecting the token it will forward $S_i$'s request to its own parent. In either case $S_j$ will update its parent to be $S_i$, meaning that $S_j$ will send its next request to $S_i$.

In this algorithm the "update" messages which were required in MBBO are replaced by the dynamically changing logical tree. Here it is not guaranteed that a request will, in one step, reach a site that is expecting the token, but a request is guaranteed to reach the token in a finite amount of time. Performance of the algorithm will depend on the average height of the tree and on the extent to which requests need to be forwarded. These factors in turn will depend on the frequency of token requests in the system. In theory, their algorithm requires $O(\log N)$ messages per critical section execution so long as the tree remains relatively balanced; simulation results are required to determine how efficiently the algorithm will actually perform in practice.

### 2.5 Algorithm CSL-2

Like NT, algorithm CSL-2, proposed by Chang, et al.[6], organizes the sites into a dynamic, logical, rooted tree. Paths which lead toward the root lead toward the current location of the token, so that requests are sent along these paths. Each site
knows only its parent in the tree, and that site whose parent is itself is the root. When
a site receives a request it sets its parent pointer to the requesting site, since that site
will be possessing the token in the future. A site initiates a critical section request by
sending a request message to its parent, and then setting its parent pointer to itself,
making itself the new root of the tree. During the time that request messages are
in transit, several sites may consider themselves to be roots, so the network will, in
general, be a forest. When no messages are in transit, however, only one site should
consider itself to be the root, and the network will be a tree.

CSL-2 differs from NT by simplifying the data structures stored at each site and
in the token. In addition to its parent pointer, each site maintains only a simple
variable next, which is the next site to receive the token. This can be thought of as a
queue of length 1. When site \( S_i \) receives a request from \( S_j \), \( S_i \) will forward the request
to its parent if it is not currently requesting or its next field is full; otherwise, \( S_i \) will
set its next field to point to \( S_j \). In any case, \( S_i \)'s parent will finally be updated to
point to \( S_j \). The token itself also does not contain a full queue, but only contains the
id of one site that is likely to be the root of the tree (or near it). Each site, before
passing on the token, sets this value to point to its own parent. A site receiving the
token updates its own parent pointer to point to the parent site, or newroot listed
in the token. (If the newroot listed in the token is the site itself, the site does not
change its own parent, since this information must be at least as recent as that in the
token.) In this way, the height of the tree is systematically reduced.
We'll see in chapter 5 that, in the network model used for the experiments in this thesis, CSL-2 cannot guarantee that a site will receive the token in a finite amount of time; i.e., it allows starvation. Therefore, performance analysis for this algorithm has not been included in the experimental results in chapter 6.
Chapter 3

Representative Fault Tolerant
Distributed Mutual Exclusion Algorithms

Fault tolerance, as applied to distributed mutual exclusion algorithms, is certainly a concept that is open to some interpretation. Claims about algorithms' tolerance to failures are often shaky, at best[1, 16]. Rarely is a rigorous demonstration (or proof) of fault tolerance attempted; and rarely does a published algorithm contain explicit instructions for what to do in case of failures¹. One of the goals of this thesis is to establish a clear definition and understanding of fault tolerance for distributed algorithms; one that can be used as a practical basis for developing and evaluating

¹for a good exception, see [15]
fault tolerant algorithms.

An extreme definition would require that (1) upon failure of an arbitrary number of sites and communication links, the algorithm continues to satisfy critical section requests from those sites that are still up; and that (2) recovering sites can reenter the system without human intervention. This strict interpretation, however, excludes every algorithm that has been proposed, generally by virtue of failure to meet the first condition. Therefore, a definition that will help in classifying and evaluating algorithms should recognize fault tolerance as a relative property, not an absolute one. It may even be possible to define fault tolerance as a measurable quantity, so that this would join message complexity and delay times as the primary measures of the value of an algorithm. Such a definition is beyond the scope of this thesis; instead, the following operational definition has been applied: An algorithm is fault tolerant if it provides mutual exclusion, without starvation, in a simulated environment in which sites occasionally go down for finite periods of time. Note that the “simulated environment” includes aspects not directly related to fault tolerance. For example, in the simulated network used in this thesis, messages between two sites might not arrive in the same order in which they are sent; this assumption has implications for all algorithms, not just fault tolerant ones.

The operational definition above is about as unrestrictive as possible. It does not, for example, mandate that sites be allowed to access critical sections during the time that one site is down. So the definition is satisfied by an algorithm which simply puts
the entire system "on hold" whenever all sites cannot be verified to be functional. Notice how the two definitions form two extremes of a continuum of fault tolerance. Most algorithms will probably fall somewhere in between; i.e., sites will continue to access critical sections despite *some* failures, but when simultaneous failures become too numerous, or when certain important sites go down, or when certain important messages get lost, the system will be put "on hold" (in terms of accessing critical sections) until certain conditions are satisfied.

### 3.1 Algorithm NLM

Algorithm NLM, proposed by Nishio, Li, and Manning[15], starts with a slightly modified version of SK, and incorporates a token regeneration scheme to make itself robust. In a network the token may occasionally get lost if, for example, the site holding it fails, or it gets sent to a site that is down. When sites request the token they each set a timer which, when it expires, indicates the possibility of a token loss. A site initiates the token regeneration protocol when its timer expires. If the old token was indeed lost, and not merely delayed, then the regeneration protocol will automatically replace it with a new token so that execution of the mutual exclusion algorithm may proceed.

Aside from the token regeneration scheme, the only difference between NLM and SK is in what order requesting sites receive the token. In SK, the token carries a queue with it, so the requests that reach it are serviced in a first-come-first-served
order. The token in NLM carries a simple array containing, for each site, the number of times that site has received the token. Each site also stores the sequence number of the last request that it has received from every other site. A site is currently requesting if its current request number, according to the site which holds the token, is greater than the number of times that it has used the token. The site, $S_i$, possessing the token, will next send it to that unique requesting site $S_j$ such that $(j - i) \mod N$ is minimum. In other words, the token continually circles around a ring, but skipping sites which are not requesting.

Like SK, this basic portion of the algorithm requires $N$ messages per critical section entry: $(N - 1)$ requests and 1 token pass. The total message complexity of NLM depends additionally on the token regeneration scheme.

Whenever a site suspects that the token has been lost (i.e., that site's timer has expired), that site sends messages to all other sites in the system proposing that a new token be generated. If any site has the token, it informs the requesting site and the regeneration attempt is abandoned. Otherwise, all sites inform the regenerating site of the number of times they have used the token, and the new token is generated with this information in its array. However, the token can only be regenerated if all sites respond. Thus, so long as one site is down, any regeneration attempt must fail. The site that is attempting the regeneration will repeatedly invoke the regeneration protocol until it is successful. Thus significant "thrashing" of messages can occur if the token gets lost.
Sequence numbers, along with the site numbers, are used to insure that if several sites attempt to regenerate a token concurrently, only one will be successful. Sequence numbers also insure that a token which was in transit during a regeneration attempt will be deleted.

2\((N - 1)\) messages are required for each token regeneration attempt, whether it is successful or not (assuming all sites are up). If some sites are down then fewer messages will be sent, since the down sites cannot respond; but if the token is indeed lost, the regeneration attempt will be repeated, and thrashing could ensue.

One important property of NLM is that it allows the possibility of “false token loss”. If a site \(S_i\) receives a token regeneration request while the token is in transit toward \(S_i\), \(S_i\) will give permission to generate a new token, and will delete the old token when it arrives. Since many messages and much time may subsequently be required for the token regeneration attempt, the false token loss phenomenon is a significant, presumably unnecessary, detriment to performance.

### 3.2 Algorithm AA

Algorithm AA, proposed by Agrawal and El Abbadi[1], is a permission based algorithm which uses dynamic request sets. In this approach, a site generates a request set every time it desires to access its critical section. This differs from Maekawa’s approach, which sets the request sets once and for all at system startup. In the dynamic approach, some sites being down need not prevent any site from attaining a
function GetQuorum( Tree: treetype): settype;

var  left, right: settype;
begin
  if Empty( Tree ) then
    return (0);
  else if  GrantsPermission( Tree.Site ) then
    return ({ Tree.Site } ∪  GetQuorum( Tree.LeftChild ));
  or
    return ({ Tree.Site } ∪  GetQuorum( Tree.RightChild ));
  else begin
    left := GetQuorum( Tree.LeftChild );
    right := GetQuorum( Tree.RightChild );
    if ( left = Ø or right= Ø ) then
      exit (error); (* no quorum available *)
    else
      return (left ∪  right);
  end;
end;

Figure 3.1: Attaining a Quorum in Algorithm AA

request set, since a request set may be formed from among those sites that are up.

In this algorithm, a logical, static, tree structure is imposed on the network. For
simplicity, we may assume a binary tree; trees in which nodes have degree greater
than 2 will require fewer messages, but will be less fault tolerant in the sense that
fewer failures can make forming a request set impossible[1]. A request set, or quorum,
consists of all sites on a path from the root of the tree to a leaf. If any site on the path
is down, then a path must be found from both of that site's children to the leaves.
The algorithm is naturally recursive, and can be expressed as shown in figure 3.1.
It is shown in [1] that any two quorums must intersect, so that mutual exclusion is guaranteed. The number of messages depends on the frequency and duration of failures. So long as all sites are up, $3 \log(N)$ messages are required per critical section, since the request set is of size $\log N$ and three messages are passed for each member of the request set. When sites are down, however, not only will the quorums be larger, but several attempts may be needed to establish a quorum. Agrawal and El Abbadi show that the largest quorum is $\lceil (N + 1)/2 \rceil$ if a binary tree is used. However, this does not represent a bound on the message complexity in any way since it does not place a limit on the number of unsuccessful quorum attempts.

### 3.3 Implementation Considerations

The two fault tolerant algorithms studied for this thesis are considerably more involved than the other algorithms. Many of the details necessary to make the algorithms work have not been discussed in the original publications. Thus, both algorithms require the implementor, or simulator, to make some decisions that can significantly affect performance. For the current research, these decisions have been guided by three goals:

1. preserve the essential character of the algorithm;

2. simulate real systems, as much as possible; and

3. optimize performance.
The remainder of this section describes and explains the adjustments to the fault tolerant algorithms necessary for implementation.

3.3.1 Timer Settings

Since sites do not possess "meta-information" about the system, one site can never know for sure if another site is down or if that site is just slow to respond to a query. Furthermore, a site cannot distinguish between a downed site and a failed communication link; the only knowledge sites can have about a particular site in the system must come from messages originating at that site. To approximate this knowledge, sites that are waiting for responses will generally set a timer, and when the timer goes off will abandon hope of receiving those responses and take appropriate action. Clearly, the timer settings are important, since if the timer is set for too short a time unnecessary corrective actions will be taken which may involve significant numbers of messages and significant deterioration in performance; and if the timer is set for too long a time then precious time is wasted while sites sit idle (in terms of critical section usage).

In the case of NLM, two different timer settings are used: $T_{cs}$ and $T_{resp}$. After requesting the critical section, if a site does not receive the token within $T_{cs}$ time then that site initiates token regeneration. Subsequently, if token regeneration has not completed, either by finding the old token or generating a new one, by $T_{resp}$ time, the regeneration attempt is restarted. Each token regeneration attempt can
involve up to $2(N - 1)$ messages (maximum $2(N - 1) - 1$ if it needs to be repeated), since it involves a site querying every other site in the system. If $T_{cs}$ is too short many unnecessary messages will be sent due to superfluous regeneration attempts; if it is too long then necessary token regeneration will be delayed. In addition to these consequences, an improper choice for $T_{resp}$ can have an even more catastrophic result. Suppose $T_{resp}$ is too short; then it is possible that during every regeneration attempt one response is late, so that token regeneration is postponed indefinitely and starvation results for the entire system. No value of $T_{resp}$ can guarantee that this will not happen because there is no limit on message transmission time. Nonetheless, a proper choice for $T_{resp}$ can make the probability of such a catastrophe negligible.

Given an estimate of the maximum message transmission time in the system, Nishio, et al.[15], stipulate $T_{cs}$ and $T_{resp}$ as follows.

$$T_{resp} = 2t_m + t_e$$

$$T_{cs} = (N + 1)t_m + (N - 1)t_e + t_d$$

where $t_m$ is the approximate maximum message transmission time; $t_e$ is the maximum time to process the token, including the critical section execution time; $t_e$ is the maximum time for a site to process a regeneration request; and $t_d$ is the maximum time for a site to process a critical section request. In most networks $t_c$, $t_d$, and $t_e$ will be quite small compared to $t_m$, so the problem comes down to determining $t_m$. In a physical network, $t_m$ will probably be estimated experimentally; in a simulated
network the distribution of message delay times should be known precisely. For both cases, the following analysis can provide some guidelines.

The time required for a site to get a response to a query is the sum of the transmission times of the query and the reply. Define a late message as one whose transmission time is greater than $t_m$, and define a late response as one in which the sum of the query and reply transmission times is greater than $T_{\text{resp}}$, where $t_e$ is assumed to be zero. Notice that if only one of the messages involved in the response is late, the response may or may not be late. Let $p$ be the probability that a single response is on time, and let $P_k$ be the probability that at least one out of $N - 1$ independent responses is late on each of $k$ consecutive token regeneration attempts. ($N$ is the number of sites in the network.) Then

$$P_k = P^k = (1 - p^N)^k, \quad k = 1, 2, \ldots$$

For example, if $p = 0.99$ and $N = 21$ then $P_1 \approx 0.18$, $P_2 \approx 0.033$, and $P_4 \approx 0.0011$. So, even if each response has a 99% chance of arriving on time, in a network of 21 sites there is still an 18% chance that a token regeneration attempt will fail even when all sites are up and the token is lost. However, the chance of failure four times in a row is less than 1%. Given the steep cost in messages of token regeneration, an implementation of this algorithm should probably strive for a $p$ value of at least 99%.

Now the question remains of what value to choose for $t_m$ to give an appropriate value for $p$. The relation between $t_m$ and $p$ is implementation dependent; for the simulation it has been determined as follows. Message transmission times are chosen
from the exponential distribution $e^{-x}$, which has an expected value of 1.0. If $S$ is the sum of two independent message times, the probability distribution of $S$ is given by

$$f_S(s) = \int_0^s e^{-x} e^{-(s-x)} dx = se^{-s},$$

and the probability that a response arrives on time is

$$P(S \leq 2t_m) = \int_0^{2t_m} se^{-s} ds.$$

Integrating by parts, we find

$$P(S \leq 2t_m) = p = 1 - (1 + 2t_m)e^{-2t_m}.$$ 

For $t_m = 3$, $p \approx 0.983$; for $t_m = 4$, $p \approx 0.997$. The argument in the preceding paragraph indicates that $t_m = 4$ is a good choice.

### 3.3.2 Implementing AA

Timer settings are just one example of a host of timing issues that are not considered by Agrawal and El Abaddi. Their research has focused solely on the existence of quorums, and has ignored issues of how a site would actually go about attaining one.

Figure 3.2 illustrates a network of 15 sites, logically organized into a binary tree. If all sites are up, then any path from the root to a leaf can be a quorum, such as \{1, 2, 4, 8\}. If site 2 is down, then \{1, 4, 5, 8, 11\} can be a quorum. The problem is that a particular site does not know which other sites are up, so how does a site decide which quorum to try for? At least three different algorithms are possible.
Algorithm 1: Send a request to all sites and wait for permission messages. When enough permission messages arrive to form a quorum, enter the critical section. Finally, send release messages to all sites.

Algorithm 2: Form a quorum one site at a time. First, send a request to the root of the tree. When permission is received [the timer goes off], recursively form a quorum starting at one [both] of the root's children, if the root is not a leaf.

Algorithm 3: Assume that all sites are up, and choose an arbitrary path from root to leaf to serve as quorum. If the timer goes off before a quorum is formed, then recursively form quorums at all remaining children of those sites that have not responded.

The three algorithms above each present advantages and disadvantages. Algo-
Algorithm 1 has the advantage of speed: in only one round of message exchanges the requesting site knows whether a quorum is currently available in the system. The disadvantage to this approach is message traffic; if all sites are usually up, this approach reduces to Lamport’s algorithm [9], which requires $3(N - 1)$ messages per critical section. Algorithm 2 most closely corresponds to the function GetQuorum described in section 3.2 and in [1]; it also is more efficient in terms of message traffic, since no superfluous requests are sent as in Algorithm 1. However, the delay that will result is quite large since permissions are obtained consecutively. Algorithm 3 combines some of the good aspects of both Algorithms 1 and 2. Like Algorithm 2, it only sends requests to sites that are down or will be participating in the quorum. In the general case when all sites are up it gets speedy response: a quorum in one round of message exchanges, as with Algorithm 1. If some of the sites in the attempted quorum are down, then some additional rounds of message exchanges are required, but the delay will still not be as bad as Algorithm 2. According to the philosophy that, since site failures are relatively rare, the common case of all sites up should be made efficient at the expense of increased complexity when failures occur [22], Algorithm 3 is the one to choose. The results presented in this thesis have been based on a simulation using Algorithm 3.
Chapter 4

Simulation Model

One of the toughest difficulties encountered in analyzing distributed mutual exclusion algorithms is that caused by the wide variety of environments in which the algorithm may be used. Distributed systems come in many shapes and sizes, and the performance of an algorithm, even its correctness, may depend on the distributed environment. It may not, therefore, be possible to make blanket statements like "Algorithm X is the best solution to problem Y." Instead, we can be specific about what environment is used for experimentation, and about how this choice affects the experimental results. In some instances speculation about how results would differ under different environments may be appropriate, but cannot replace actual experiments.

Results should also be understood as deriving in part from the simulation technique chosen. This chapter describes in detail first the distributed environment, and then the simulation technique.
4.1 Distributed Environment

Some of the parameters used in this simulation model are drawn from Chang, et al.[3, 5], whose model was in turn based partly on [17]. The current model is broader and more realistic than that in [3, 5] in four important ways. (1) The size of the network can be varied, so that performance can be related to number of sites as well as frequency of requests. (2) Message transmission times are unpredictable, as opposed to constant. (3) The network modeled is "heterogeneous", in the sense that some sites may be more frequent users of the critical section than others. (4) Site failures may occur, when applicable (i.e., when investigating algorithms that claim to be fault tolerant).

Table 4.1 summarizes the input parameters used for the model. The distributed system consists of autonomous sites that communicate with each other via message passing. Message transmission time is finite but unpredictable, and messages might not be delivered in the order in which they are sent. Any site can send a message directly to any other site in the system. Broadcast message facilities are not available, so that $N$ messages are required to send the same information to $N$ sites (as opposed to 1 message, if broadcast were available).

The average message transmission time is 1.0, and the critical section execution time is fixed at 0.0002. The time of transmission for each message is chosen from an exponential distribution with mean 1.0, independent of which two sites are involved in the message, and of any previous message. These figures reflect the realistic
<table>
<thead>
<tr>
<th>SCOPE</th>
<th>PARAMETER</th>
<th>[DISTRIBUTION,] VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>independent variables</td>
<td>network size ($N$)</td>
<td>5–75</td>
</tr>
<tr>
<td></td>
<td>request frequency ($\lambda$)</td>
<td>0.001–1.0</td>
</tr>
<tr>
<td></td>
<td>failure frequency ($\beta$)</td>
<td>1/10,000–1/500</td>
</tr>
<tr>
<td>set for each run</td>
<td>site request frequency ($\lambda_i$)</td>
<td>exponential, mean $\lambda$</td>
</tr>
<tr>
<td></td>
<td>site request distribution</td>
<td>Poisson, parameter $\lambda_i$</td>
</tr>
<tr>
<td></td>
<td>site failure distribution</td>
<td>none or Poisson, parameter $\beta$</td>
</tr>
<tr>
<td>set for each instance</td>
<td>message transmission time</td>
<td>exponential, mean 1.0</td>
</tr>
<tr>
<td>set for entire experiment</td>
<td>c.s execution time</td>
<td>0.0002</td>
</tr>
<tr>
<td></td>
<td>length of run</td>
<td>10,000</td>
</tr>
<tr>
<td></td>
<td>number of runs for each</td>
<td>50</td>
</tr>
<tr>
<td></td>
<td>($N, \lambda, \beta$) combination</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.1: Simulation Model Parameters
assumption that critical section execution time is non-zero but several orders of magnitude smaller than message transmission time. Each run of the simulation lasts for 10,000 time units, and the simulation was run 50 times for each value of the input parameters. All message processing times are taken to be zero.

In order for the model to be as general as possible, so that it will have applicability to many different real implementations, requests for critical section execution have been randomly distributed in time in such a way that an average rate of request arrivals can be maintained without having any correlation between requests; i.e., the number of requests in a given time interval does not depend on the number of requests in any previous time interval. Such a distribution is generated by the Poisson process [19]. A homogeneous Poisson process, wherein the average rate of arrival of requests remains constant over one run of the simulation, allows isolation of specific degrees of request traffic for investigation. Furthermore, the modeled distributed system is a heterogeneous one in that different sites may request the critical section at different frequencies. Specifically, at site \( S_i \) requests arrive according to a Poisson distribution with mean \( \lambda_i \); i.e., the site averages \( \lambda_i \) requests per unit time. The \( \lambda_i \)'s are themselves chosen from an exponential distribution with mean \( \lambda \), where \( \lambda \) is specified by the experimenter. By this method, the general situation in which some sites may be heavier users of the token than others can be studied, while at the same time experimentally controlling the overall frequency of token requests in the system.

For those experiments in which network failures will be considered, no "human"
element is introduced into the picture. What this means is that a site can detect a failure in another site only via the messages it receives, or does not receive; no site has access to "meta-information" about the network state. Furthermore, only site failures are considered, and not communication link failures. This is reasonable since a site $S_i$ which fails to communicate with $S_j$ cannot know if $S_j$ is down or if the link connecting them is down. Nishio, et al.[15], also argue that it is illogical for a site to be capable of detecting the failure and recovery of communication links.

For those experiments in which sites may fail, the failures arrive at each site according to a Poisson distribution with parameter $\beta$. $1/\beta$, the average amount of time between breakdowns, was varied from 500 to 10,000. Those experiments which varied the other independent variables held $1/\beta$ at 2000. At this level, failures are rare, but it is expected that most sites will fail at least once during a run of the simulation (10,000 time units). The average frequency of arrival of breakdowns is equal at all sites. The duration of each failure is chosen from an exponential distribution with mean 50; and, again, the average duration is equal at all sites. These average values are controlled by the experimenter, so that performance can be studied under a range of network reliability.

In general the exponential and Poisson distributions have been chosen for their simplicity and generality. This thesis does not deal with specific distributions which may occur in specialized applications; rather, it provides a general basis for analyzing performance under a wide variety of practical applications.
4.2 Simulation Program

The complete source code for the simulation program can be found in the appendix. The program encompasses 11 files. The header file, `mutex.h`, contains declarations of constants, data structures, and functions that are used in the other files. `main.c` contains the main driver routine which reads input parameters from the command line, calls the algorithm simulators, and compiles the statistics. This file also contains some global function and variable definitions. `rand.c` is the random number generator. `heap.c` and `queue.c` contain data structure definitions. The remaining files implement the various algorithms.

The technique used is “discrete event simulation”. In this approach time is viewed as a sequence of discrete events: “site failure” and “receipt of a message” are examples of discrete events. The process being simulated conforms to some rules for event causation, so that an event $E_1$ along with a particular state of the system at the time of $E_1$, uniquely determines a set of future events $E_2, E_3, \ldots, E_n$. The exact timing of events $E_2, E_3, \ldots, E_n$ may, however, be non-deterministic. For example, in algorithm SK, the event “need for critical section arises at site 3” causes the future events “receipt of request message from 3 at $x$”, for every $x$ a site in the network, $x \neq 3$. The time of occurrence of these future events will depend on assumptions made about message transmission time, and might be randomly determined.

When a future event is scheduled, a record for that event is entered in a heap, or priority queue. In a heap, every record has a key field, and the keys are drawn from
an ordered set. Items may be added to the heap in any order, but the item removed
must be that item with the lowest key value currently in the heap. In discrete event
simulation, the key for an event is the time at which that event occurs, so the event
with the lowest time in the heap is the next event to occur.

Each algorithm is simulated by scheduling some initial events, and then removing
events one by one and scheduling new events as required, until the "end of simulation"
event is removed. The initial events include the first critical section need at each site,
the first failure at each site (if applicable), and the end of the simulation. As each
event is processed, any necessary statistics are updated, such as number of messages if
the event is the receipt of a message. When the end of simulation event is encountered,
the heap is flushed, the final statistics for the run compiled, and, generally, the process
restarted.
Chapter 5

Problematic Algorithms

This chapter, even more than chapter 6, illustrates the importance of modeling and computer simulation in the design of distributed algorithms. Although the simulation is not specifically discussed in this chapter, each of the results presented here were discovered with the aid of the simulation. Both algorithms CSL-2 and AA have been published with proofs of certain properties that seem to indicate that those algorithms correctly solve the distributed mutual exclusion problem. Nonetheless, complications involved in the design of distributed mutual exclusion algorithms some factors to be overlooked. These factors are discussed in this chapter.

5.1 Algorithm CSL-2

In addition to their correctness proofs, Chang, et al.[6], have performed simulation experiments on CSL-2 and have not found any problems. The reason for this is
that their simulation model assumed constant message transmission times, so that all messages are delivered in the same order as that in which they are sent. The problem with this algorithm arises when two messages between two sites get switched in transit, so that they are delivered in reverse order from the way they were sent.

Figure 5.1 illustrates the problem. Figure 5.1a shows an initial network configuration in which no messages are in transit; site A holds the token. (The algorithm startup configuration has site A as the parent of all other sites, and holding the token. The configuration in figure 5.1a can be obtained from startup by the following sequence of critical section requests: F, E, C, A, D, C, D, B, A.) In figure 5.1b sites B, C, and E have all initiated requests and their request messages are in transit. In figure 5.1c the request messages from figure 5.1b have all been received, the token is on its way to site B, and site D has initiated a request. Between figures 5.1c and 5.1d site B has received D's request and, since B's next field was full, forwarded the request to B's parent, C. B then made D its parent.

The core of the problem can be seen in figures 5.1e and 5.1f. In figure 5.1e the token has been received and used by B, and then forwarded to C, since B's next field contained C. The newroot field in the token now contains D, since D is the parent of B. The request of D, which had been forwarded through B, is still in transit, so that two messages between two sites are in transit simultaneously. Figure 5.1f shows what happens when the token message arrives first: C becomes the child of D, forwards the token to its next site (E), while the request from D is yet to arrive at C. In figure
Figure 5.1: An Example of Starvation with CSL-2
5.1g the request from D has finally arrived at C, and since C is not now requesting, D’s request is forwarded to C’s parent, D itself! Also in figure 5.1g, E has received the token, changed its parent according to the information in it, and, since E’s next field is empty, has not forwarded it.

The situation in figure 5.1g shows that a problem has developed since D is the root of the tree yet E has no intention of sending it the token. The problem becomes clearer if the example is taken one step further, as in figure 5.1h. Here, F has requested the token from E, and become the parent of E. From this point on, only E and F can use the token; the other sites will starve.

5.2 Algorithm AA

As discussed in section 3.3.2, the method used to find a quorum under algorithm AA is as follows. Choose a path from root to leaf and send request messages to all sites on the path. If all sites respond, then this becomes the quorum; if some sites are down, then additional requests are sent, according to the algorithm in figure 3.1. The important thing to notice is that not all requests for a particular critical section instance are sent at the same time. This is also a characteristic of algorithm 2 in section 3.3.2, and because of it deadlock may result.

Figure 5.2 illustrates the problem with AA. The network configuration is shown in figure 5.2a; site A is down. First, site E desires the critical section, and chooses the request set \{A, C, G\} (figure 5.2b). E receives grant messages from C and G,
Figure 5.2: An Example of Deadlock with AA
and waits for a response from A. While E is waiting, F chooses request set \{A, B, D\} (figure 5.2c), and receives grants from B and D. Since site A is down, timers go off at both sites E and F, and they both must extend their request sets to include another child of A (figure 5.2d). Now we can see the problem in figure 5.2e. Site B thinks that F requested first and has priority over E; site C thinks that E requested first and has priority over F. At this point a deadlock has formed and no site in the network can use the critical section. Eventually, timers will go off at sites E and F, and both sites will choose new request sets. However, there is no guarantee that the same problem won't arise repeatedly, causing these sites to starve.
Chapter 6

Performance Analysis

The experiments which comprise the core of this thesis can be divided into two halves, depending on whether site failures are allowed. The first half disallows failures in order to examine algorithms SK, CSL, MBBO, and NT, and compare their respective performances. The second half examines the performance of algorithm NLM in a network in which sites may occasionally fail.

Each reported result value is a mean value attained from 50 runs of the simulation, each lasting 10,000 time units. Error bars indicate 95% confidence intervals.

6.1 Measures

The simulation output includes the following factors, which are then used in the performance analysis:

1. expected number of messages per critical section execution;
Table 6.1: Statistics Measured

<table>
<thead>
<tr>
<th>STATISTIC</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>messages per critical section</td>
<td>total messages / total critical sections</td>
</tr>
<tr>
<td>average waiting time</td>
<td>total time waiting for all sites / total critical sections</td>
</tr>
<tr>
<td>expected maximum waiting time</td>
<td>longest wait for a single critical section</td>
</tr>
<tr>
<td>throughput</td>
<td>total critical sections / length (in time) of a run</td>
</tr>
</tbody>
</table>

2. expected delay a site experiences in receiving the token;

3. expected value of the maximum delay a site may experience;

4. expected throughput: the mean number of critical sections executed per unit time.

Table 6.1 summarizes the desired measures. Each statistic has been computed for various values of $\lambda$, representing the frequency of critical section requests at each site; $N$, representing the number of sites in the network; and $\beta$, where applicable, representing the frequency of failures at each site.

The first statistic, messages per critical section, represents the total number of messages passed in a run of the simulation, divided by the total number of critical sections executed during that run of the simulation. Average waiting time is the sum, over all sites and all critical sections, of the time between requesting critical section entry and receiving the token, divided by the total number of critical sections executed. The maximum waiting time statistic is formed by looking at the longest
amount of time that any site had to wait for the token during one run of the simulation, and averaging this value over all runs. Throughput is the total number of critical sections executed divided by the length of the simulation (in time). The time units are generic; 1.0 time unit is the expected message transmission time.

6.2 Performance Without Failures

The first half of this experiment, in which site failures do not occur, can be divided into two parts: one which measures performance as a function of the request traffic, keeping the network size fixed; and one which measures performance as a function of the network size, keeping request traffic fixed. Specifically,

1. Network size fixed at 21, $\lambda$ varies from 0.001 to 1.0;
2. $\lambda$ set to 0.001, 0.06, or 0.2, and network size varies from 5 to 75.

This section provides a more detailed description and analysis of the results of these two experiments.

6.2.1 Varying Request Frequencies

Figures 6.1–6.4 comprise the first experiment, in which the size of the network was fixed at 21 sites, and $\lambda$, the request traffic, was changed. The network size of 21 was chosen for consistency with previous work [3, 5]. $\lambda$ ranged from 0.001 to 1.0. At the low end for $\lambda$ conflicts in requesting the critical section are rare; at the high end most
of the time most sites are requesting. In fact, the graphs show that for $\lambda > 0.3$ the algorithms studied have nearly reached their saturation level performance.

Figure 6.1 shows the message traffic plotted as a function of the request traffic. NT stands out as providing the most consistently efficient performance, showing excellent results at all levels of request traffic; it essentially achieves the optimal two messages per critical section in very heavy request traffic. The exceptional performance of NT indicates that the tree tends to remain balanced, and this is especially impressive in an environment of heterogeneous request traffic. The theoretical analysis of Makki, et al.[12], which predicts an inverse relationship between message traffic and request traffic for MBBO, has been verified by the simulation. This algorithm also comes close to optimal performance in heavy request traffic. CSL does provide some savings
over SK in light traffic, but it still does not compare with NT. SK, which is the simplest algorithm to analyze theoretically, predicts $N (21)$ messages per critical section, regardless of the level of request traffic. The glitch found in light traffic is probably due to the possibility of a site using the token twice in a row, and thus not needing to send requests the second time. This can only happen when the token queue occasionally becomes empty, and sits idle at one site for some period of time; and this, in turn, is only likely to happen in light request traffic.

One important consideration to keep in mind when studying these results is that the network modeled here does not support broadcast facilities. In a network that supports broadcasting, such as ETHERNET, it costs only one message transmission to send the same message from one site to all others. In that case the waiting time would become the most important statistic, since all four algorithms would be highly efficient in message traffic. (In fact, SK and CSL should never exceed two messages per critical section).

The average waiting time for the critical section is plotted in figure 6.2. This statistic indicates how quickly the system typically responds to a request for critical section. Here SK and CSL are the best performers, particularly in light request traffic. MBBO’s slight sluggishness is probably due to the time which that algorithm requires sites to “wait” with the token. The throughput, plotted in figure 6.3 as a function of request traffic, is most meaningful in heavy request traffic, when it represents a kind of “inverse” of the average waiting time. Here again MBBO is seen to be somewhat
slower than the others.

Figure 6.4 plots the expected maximum waiting time against request traffic. While this measure certainly does not indicate any sort of "guarantee" that a site will not wait longer, it does give an indication of an algorithm's "worst case" performance; information that cannot be deduced from the average wait time. MBBO is somewhat slower than the rest, although none of the algorithms show unusually poor worst case performance.

6.2.2 Varying Network Size

In the second part of the experiment performance is seen as a function of network size, with \( \lambda \) held constant. Each group of runs, covering a range of \( N \)'s, was performed for
Figure 6.3: Throughput vs. $\lambda$ ($N = 21$)

Figure 6.4: Expected Maximum Waiting Time vs. $\lambda$ ($N = 21$)
three separate $\lambda$ values, representing light, moderate, and (relatively) heavy traffic of critical section requests. These results are plotted in figures 6.5–6.16.

Figures 6.5–6.7 plot message complexity against $N$. In even moderate and heavy traffic the constant performance of NT and MBBO is apparent, with NT performing slightly better. SK and CSL vary linearly with $N$. In light traffic the story is quite different. NT continues to show constant performance, but MBBO is virtually indistinguishable from SK in requiring $N$ messages per critical section. CSL still appears to be proportional to $N$ in light traffic, but performs better than SK and MBBO by some constant factor (roughly 0.4). Clearly, if message traffic is the most important measure, then NT is the best choice since it outperforms the other algorithms for every $(N, \lambda)$ combination.
Figure 6.6: Messages per Critical Section vs. $N$, $\lambda = 0.06$

Figure 6.7: Messages per Critical Section vs. $N$, $\lambda = 0.001$
Figure 6.8: Average Waiting Time vs. N, $\lambda = 0.2$

Figure 6.9: Average Waiting Time vs. N, $\lambda = 0.06$
The average waiting time for the critical section is plotted as a function of network size in figures 6.8–6.10. The heavy and moderate traffic plots reveal the effect of the extra waiting time that MBBO requires sites to do with the token. Otherwise, these graphs generally show a linear relation between \( N \) and waiting time for each algorithm. This linear relationship is not surprising, especially in heavy traffic, since if most sites are generally requesting, then a site typically will have to wait for all other sites in the system to complete before it gets the token. The light traffic case is somewhat more difficult to explain. Three of the plots level off, or at least grow quite slowly, when the number of sites exceeds about 20. Of these three, MBBO again shows the effects of the extra wait time. Average waiting time for NT, on the other hand, grows relatively rapidly, passing that for MBBO in large networks. This may
indicate that in light traffic of requests, the dynamic tree that NT maintains tends to get stretched out, and behaves more like a linear list.

The throughput of the system is plotted in figures 6.11-6.13. In heavy and moderate traffic the sluggishness of MBBO shows up quite clearly, while in light traffic the algorithms are virtually indistinguishable.

Figures 6.14-6.16 show the expected maximum waiting time as a function of \( N \). As described above, this statistic is only a rough indication of the worst case performance in terms of waiting time. Although all four algorithms perform quite similarly in networks smaller than 30 sites, NT stands out as allowing longer waits than the other algorithms in large networks. The other three algorithms show a linear correspondence between worst case waiting time and network size in moderate and heavy
Figure 6.12: Throughput vs. N, $\lambda = 0.06$

Figure 6.13: Throughput vs. N, $\lambda = 0.001$
Figure 6.14: Expected Maximum Waiting Time vs. N, $\lambda = 0.2$

Figure 6.15: Expected Maximum Waiting Time vs. N, $\lambda = 0.06$
Figure 6.16: Expected Maximum Waiting Time vs. N, $\lambda = 0.001$

request traffic, which indicates that in the worst case every site in the network uses the token between a particular site’s request and receipt of the token. The fact that NT’s worst case performance grows faster than linearly with network size indicates that, in some instances, a site may use the token repeatedly while another site waits. Since this phenomenon does not appear in the average waiting time, we may conclude that this worst case scenario does not occur frequently in practice. In light traffic, the maximum waiting time for each algorithm grows only very slowly with $N$. Here, SK and CSL again show significantly quicker response times than MBBO and NT.

6.2.3 Final Analysis, No Failures
Table 6.2 summarizes the performance results, indicating which algorithms perform well under which conditions. NT appears in every category for few messages, and SK and CSL appear in every category for low waiting time. Thus, the primary conclusions to be drawn from these experiments are the following three: (1) Whenever message traffic complexity is the primary evaluation criterion, NT is the most efficient algorithm to use; (2) Whenever delay time is the primary evaluation criterion, CSL is the best algorithm (SK is just as fast, but requires more messages); and (3) Whenever both message traffic and waiting time are important criteria, the situation becomes somewhat murkier, although NT is the only algorithm which provides reasonably good performance according to both criteria.

Although table 6.2 summarizes the high points of the simulation results, it does
not capture the whole story. In particular,

- **NT** is not listed as having good waiting times in large networks with heavy traffic. This is because of its poor worst case performance as seen in figure 6.14. However, **NT**'s average waiting time is virtually indistinguishable from **SK** and **CSL**, so that its worst case performance is quite rare.

- **NT** is also not listed for low waiting time in large networks with light traffic. This is because its average wait is more than twice that of **SK** and **CSL** in the largest networks tested, and its maximum wait was the highest of the algorithms tested. On closer examination of figure 6.10, however, we can see that even in networks of 75 sites, under **NT** sites generally wait only about four message transmission times for the token. This does not seem unreasonable, especially considering how many fewer messages **NT** requires than the other algorithms.

- **MBBO** is nowhere listed for low waiting time because it is consistently slower than the others. However, the difference between **MBBO** and the fastest algorithms, **SK** and **CSL**, is consistently a small constant: in light traffic **NT** requires approximately two more time units than **SK**, regardless of network size. Furthermore, **MBBO** never varies from this consistent performance, as **NT** does in large networks.

One final note: In a network which supports broadcast facilities, **SK** provides optimal performance in request traffic (2 messages per critical section) and the quickest
response time of the algorithms studied. In such a network, then, SK would be the best choice. (CSL is as fast as SK, but does not provide any message traffic savings over SK in such an environment, so that SK is preferable because of its simplicity.)

6.3 Performance With Failures

When sites are allowed to fail, a third independent variable is introduced into the model: $\beta$, the frequency of failures at each site. Thus, the results for algorithm NLM, which is the only functional algorithm that handles site failures, encompasses three experiments.

1. Network size fixed at 21, $\beta$ fixed at 1/2000, and $\lambda$ varies from 0.001 to 1.0;

2. $\beta$ fixed at 1/2000, $\lambda$ set to 0.001, 0.06, or 0.2, and network size varies from 5 to 75;

3. Network size set to 21 or 50, $\lambda$ set to 0.001 or 0.2, and $\beta$ varies from 1/10,000 to 1/500.

6.3.1 Varying Request Traffic

The results from experiment 1 are plotted in figures 6.17–6.20. Interestingly, the effects of the site failures show up primarily in the waiting time, not in message traffic. The message traffic, plotted in figure 6.17, shows only a small increase over the $N$ (= 21) messages required by SK. (Since NLM is based on SK, it cannot require
fewer messages, and will in general require more.) This, despite the fact that once
the token is lost one site must repeatedly "pump" messages out in attempting to
regenerate it. The fact that only one site will be doing this (only in rare cases, when
messages get delayed, will two sites attempt to regenerate several times in succession,
concurrently) and that the token rarely gets lost, keeps the message count down.

The degradation in performance in terms of waiting time shows up most poignantly
in figure 6.19, the plot of expected maximum waiting time. While the wait under SK
never exceeded 40, under NLM it exceeded 600 on occasion. This is certainly due to
token loss; once the token is lost it cannot be regenerated until all sites are up. Thus,
even though each site individually recovers within an average of 50 time units, other
Figure 6.18: Average Waiting Time vs. $\lambda$, Algorithm NLM, $N = 21$, $\beta = 1/2000$

Figure 6.19: Expected Maximum Waiting Time vs. $\lambda$, Algorithm NLM, $N = 21$, $\beta = 1/2000$
sites may go down in the meantime, so that considerably more time is needed to re-generate the token. This pathological case, despite occurring rarely, is bad enough to cause significant deterioration in overall performance, as seen in figures 6.18 and 6.20. Recall, furthermore, that under NLM “false token loss” is possible, as described in section 3.1. The results presented here indicate that a method for token regeneration which does not risk false token loss could significantly improve performance.

6.3.2 Varying Network Size

Results from experiment 2, that for which the network size was varied, are plotted in figures 6.21–6.24. Here again, message traffic does not appear to be significantly affected by site failures, as the number of messages grows only slightly faster than
Figure 6.21: Messages per Critical Section vs. N, Algorithm NLM, $\beta = 1/2000$

Figure 6.22: Average Waiting Time vs. N, Algorithm NLM, $\beta = 1/2000$
Figure 6.23: Expected Maximum Waiting Time vs. N, Algorithm NLM, $\beta = 1/2000$

Figure 6.24: Throughput vs. N, Algorithm NLM, $\beta = 1/2000$
linearly with network size. The deterioration in performance in terms of waiting
time and throughput is, however, unmistakable. For networks larger than about 20
the maximum wait becomes intolerable when critical section requests are frequent.
Considering average wait time, the same can be said for networks larger than 40 sites.
A likely explanation for this rise in wait times in large networks is probably the token
loss problem, as discussed above. Whenever the token gets lost, the more sites there
are in the network, the less likely it is that all of them will be up simultaneously so
that a new token can be generated.

6.3.3 Varying Breakdown Frequency

Experiment 3 provides the most striking results. In terms of messages per critical
section and average waiting time (figures 6.25–6.28) performance stays fairly con­
sistent as frequency of breakdowns increases (1/β decreases), until β exceeds 1/1000,
at which point the algorithm breaks down. This threshold value is the same whether
N = 21 or N = 50; it may indicate a point at which breakdowns are frequent enough
that, when the token gets lost the probability is negligible that it will ever be regen­
erated. The maximum waiting time plots (figures 6.29–6.30) also show an explosion
around β = 1/1000, although these also show some performance deterioration at
higher values of 1/β. As expected, the throughput (figures 6.31–6.32) is quite low
when breakdowns are frequent, and approaches 1, the performance of SK for high
request traffic, as breakdowns become rare.
Figure 6.25: Messages per Critical Section vs. $1/\beta$, Algorithm NLM, $N = 21$

Figure 6.26: Messages per Critical Section vs. $1/\beta$, Algorithm NLM, $N = 50$
Figure 6.27: Average Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 21$ 

Figure 6.28: Average Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 50$
Figure 6.29: Expected Maximum Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 21$

Figure 6.30: Expected Maximum Waiting Time vs. $1/\beta$, Algorithm NLM, $N = 50$
Figure 6.31: Throughput vs. \(1/\beta\), Algorithm NLM, \(N = 21\)

Figure 6.32: Throughput vs. \(1/\beta\), Algorithm NLM, \(N = 50\)
6.3.4 Final Analysis, With Failures

Since NLM is the only truly distributed fault tolerant algorithm available, in the sense of chapter 3, conclusions drawn from performance experiments must be somewhat limited; we cannot compare it with other approaches. The only exception is the primary site approach, which in some sense is not truly distributed since the failure of one site causes the entire system to go down (at least as far as attaining the critical section.

What can be said is that in some environments the algorithm can handle site failures without showing a significant deterioration for the general case over SK, which cannot handle site failures, while in other environments the algorithm is not practical. Specifically, when breakdowns occur more often than about once every 1000 message transmission times at each site, the algorithm cannot be used. Furthermore, even when breakdowns are infrequent, if the network is large (greater than about 30 sites) and critical section requests are frequent, then the algorithm still cannot be used. In other situations the algorithm handles site failures quite well. One caveat: even when the algorithm performs well in general, it still has a tendency to cause exceptionally long delays in granting critical section on occasion. Therefore in time critical applications this algorithm might not be preferable to the primary site approach, despite its fault tolerance capability.
Chapter 7

Conclusions and Future Directions

This thesis has described an experimental model for evaluating distributed mutual exclusion algorithms. This model is sophisticated enough to model realistic networks, to accommodate a variety of algorithms, and to provide useful results. The current research significantly extends previous work covering distributed mutual exclusion performance analysis in several ways: a wide variety of algorithms has been studied; the simulated network supports realistic features, including site failures and unpredictable message transmission times; and the effects of varying network size and failure frequency, as well as request traffic, have been studied.

The model has been used to simulate seven algorithms, two of which are designed for networks in which sites may fail. These algorithms have been evaluated and compared according to two performance criteria: message traffic complexity and delay in accessing the critical section. The results of these investigations can be summarized
as follows.

- Two algorithms, CSL-2 and AA, have not correctly avoided starvation and deadlock, and therefore should not be considered for implementation.

- The algorithm proposed by Naimi and Trehel[14] (NT) performs most efficiently in terms of message traffic complexity in a general network without failures.

- Two other algorithms, SK and CSL[5, 23], are most efficient in terms of delay times in a network without failures.

- Considering both criteria, NT is probably the best in a general network, while SK is clearly the best in a network that supports broadcast messages.

- When site failures are taken into account, algorithm NLM performs quite well in the usual case, so long as the network is not too large and breakdowns are not too frequent.

- Even when NLM operates in a friendly network, it still allows occasional long delay times. This is due to the fact that a lost token cannot be regenerated unless all sites in the network are up simultaneously. This is the main drawback of NLM.

Much work remains to be done in this field. Most importantly, this thesis demonstrates that current methods for verifying distributed algorithms are inadequate. Ei-
ther all algorithms will need to be tested experimentally in the future, or more complete and rigorous analysis techniques for distributed algorithms must be developed.

Further research may include extensive performance analyses for permission based algorithms, including [9, 10, 18, 20, 21]. Some interesting open questions are "could there be a permission based algorithm which approaches the optimal performance of two messages per critical section, as NT and MBBO do?" and "is it possible for any algorithm to provide optimal performance in all cases of request traffic?".

Finally, future work must include development of fault tolerant algorithms. Token based mutual exclusion algorithms are seen as having poor failure resiliency [1] because of the complex token regeneration or election [7] algorithms. Current wisdom has it that because of these election schemes, permission based algorithms will prove to be more acceptable when fault tolerance is taken into account [1]. The current research was undertaken in part to test this assumption. Since the only permission based, fault tolerant algorithm was found to have a problem with deadlock, it remains to be seen how fault tolerant the permission based algorithms really are.

In the current research environment in this field, developing a clear consensus for evaluating algorithms is as important as developing new algorithms. Today, new distributed mutual exclusion algorithms tend to get lost in the crowd. It is within the context of this goal that the current research has been undertaken.
Appendix A

Simulation Source Code

/*************************************************************
 mutex.h
 ************************************************************/
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define MAXNET 100
#define ENDTIME 10000.0
#define BREAKTIME expdis(breakt)
#define DOWNTIME expdis(downt)
#define CSTIME 0.0002
#define AVG_MSG_TIME 1.0
#define TRANS ((*msg_time)())
#define max(x, y) (((x) > (y)) ? (x) : (y))
#define min(x, y) (((x) < (y)) ? (x) : (y))
#define my_abs(x) (((x) >= 0) ? (x) : -(x))
#define sq(x) ((x)*(x))

/* ---- enumerated types ---- */

enum eventype {NeedArises, ReceiveRequest, ReceiveToken,
    ReceiveUpdate, EndWait, ReceiveMissing, ReceiveAck, ReceiveNack,
    ReceiveCandidate, ReceiveRefused, ReceiveInquire, ReceiveFail, ReceiveRelease, ReceiveYield,
    ReceiveGrant, BreakDown, Recovery, TimeOut, ExitCS, EndSim};
typedef enum eventype eventype;

76
enum boolean {false, true};
typedef enum boolean boolean;

/* ---- structure types ---- */
typedef struct heap {
    int top;
    int size;
    void **list;
    int (**comp)(void *, void *);
    void (**outnode)(void *);
} heap;
typedef struct request {
    int site;
    long int timestamp;
    long int seqnum;
} request;

struct eventnode {
    double time;
    eventtype code;
    int nearsite; /* site at which this event takes place */
    int farsite; /* other site which may be of interest */
    long int seqnum1;
    long int seqnum2;
};
typedef struct eventnode event;

struct Qnode { /* member of linked-list/queue */
    int site;
    struct Qnode *next;
};
typedef struct Qnode Qnode;

struct Qhdr { /* ptr to linked-list/queue */
    Qnode *head;
    Qnode *tail;
};
typedef struct Qhdr Qhdr;

struct statnode { /* vital statistics of system*/
double now;      /*current time*/
int messages;    /*running totals...*/
int cs_uses;
double total_wait;   /*total waiting time over all sites*/
double start_wait[MAXNET];  /*when ith site started wait*/
double start_big_wait;
int big_wait_site;
double max_wait;
int active;        /*# of sites currently requesting*/
double last_change; /*last time # of sites requesting */
/*changed*/
double L;          /*avg # requesting * total time*/
int maxact;        /*Max # requesting at any time */
int maxforward;
};
typedef struct statnode simst;

double expdis(double alpha);
double ranf(void);

void createQ(Qhdr *Q);
boolean emptyQ(Qhdr Q);
int firstQ(Qhdr Q);
int lastQ(Qhdr Q);
void deQ(Qhdr *Q);
void enQ(int site, Qhdr *Q);
void append(Qhdr *back, Qhdr *front);
void search(Qhdr Q, boolean onQ[]);

void createHp(heap *Hp, int size, int (*comp)(void *, void *),
        void (*outnode)(void *));
boolean emptyHp(heap *Hp);
void closeHp(heap *Hp);
void deHp(heap *Hp, void **temp);
void enHp(heap *Hp, void *temp);
boolean reposition(heap *Hp, void *temp);
boolean delete_evt(heap *Hp, void *temp);
void outheap(heap *Hp);

double trans(void);
void out_evt(void *p);
int comp_evt(void *a, void *b);
void out_req(void *p);
int comp_reqs(void *a, void *b);
event *alloc_event(eventtype code, int nearsite, double time);
void *mycalloc(int number, int size);
void *mymalloc(int size);

/**
 * main.c
 */
#include "mutex.h"
#include <ctype.h>

#define MAXTRIALS 50
#define ALGNUM 7
#define CONFIDENCE 1.96

struct finals {
    double mgpercs[MAXTRIALS];
    double waitime[MAXTRIALS];
    double maxwait[MAXTRIALS];
    double thrput[MAXTRIALS];
    double mgpercsmean;
    double mgpercssd;
    double waitimesmean;
    double waitimesd;
    double maxwaitmean;
    double maxwaitsd;
    double thrputmean;
    double thrputsd;
};
typedef struct finals finals;

void banta(void);
void liu(void);
void naimi(void);
void suzuki(void);
void nishio(void);
void agrab(void);
void tn(void);
void tnozig(void);
void maekawa(void);
void chang(void);

/* global variables; "extern" in the files that need them */

unsigned int seed;        /* for random #'s and seed*/
int N;       /* network size */
double lambda[MAXNET];  /* rate of requests at each site */
double breakt, downt;  /* rate of breakdowns & repairs */
double (*msg_time)(void);  /* pointer to message time function */
heap eventHp;  /* main simulation event heap */
simst sim;  /* record of simulation statistics */

/* common routines used by all algorithms */
/* comp_evt:     
   compare two events by time */
int comp_evt(void *a, void *b)
{
    int answer;

    if (!a)
        answer = -1;
    else if (!b)
        answer = 1;
    else if (((event *)a)->time > ((event *)b)->time)
        answer = 1;
    else if (((event *)a)->time < ((event *)b)->time)
        answer = -1;
    else
        answer = 0;

    return answer;
}

/* mainly for debugging */
void out_evt(void *p)
{
    event *q;
    char *codes[] = {"NeedArises", "ReceiveRequest",
                     "ReceiveToken", "ReceiveUpdate",
                     "EndWait", "ReceiveMissing", "ReceiveAck",
                     "ReceiveNack", "ReceiveCandidat",
                     "ReceiveRefused", "ReceiveInquire",
                     "ReceiveFail", "ReceiveRelease",
                     "ReceiveYield", "ReceiveGrant",
                     "BreakDown", "Recovery", "TimeOut",
                     "ExitCS", "EndSim"};

    q = (event *)p;
    printf("%s\ttt: %9.2f near: %3d"
" far:%3d s1:%4d s2:%4d\n",
codes[q->code], q->code!=TimeOut?"\t":",
qu->time, q->nearsite, q->farsite,
qu->seqnum1, q->seqnum2);
}

int comp_reqs(void *a, void *b)
{
    int answer;
    if (a == NULL)
        answer = -1;
    else if (b == NULL)
        answer = 1;
    else if (((request *)a)->timestamp != ((request *)b)->timestamp)
        answer = ((request *)a)->timestamp - ((request *)b)->timestamp;
    else /* timestamps are equal; go by site number */
        answer = ((request *)a)->site - ((request *)b)->site;

    return answer;
}

void out_req(void *p)
{
    request *q;
    q = (request *)p;
    printf(" (%d,%ld,%ld)", q->site, q->timestamp, q->seqnum);
}

void *mymalloc(int size)
{
    void *p;

    if (!(p = malloc(size))) {
        printf("MEMORY ALLOCATION FAILED; PROGRAM ABORTING.\n");
        exit(1);
    }

    return p;
}

void *mycalloc(int number, int size)
{
void *p;

if (!(p = calloc(number, size))) {
    printf("MEMORY ALLOCATION FAILED; PROGRAM ABORTING.\n");
    exit(1);
}

return p;

/* alloc_event:
   allocate memory for an event, set the code,
   nearsite, and time fields */
event *alloc_event(eventtype code, int nearsite, double time)
{
    event *temp;

    temp = (event *)mymalloc(sizeof(event));
    temp->code = code;
    temp->nearsite = nearsite;
    temp->time = time;

    return temp;
}

/* init_sim()
   initializes statistical variables */
void init_sim()
{
    int i;

    sim.now = sim.total_wait = sim.max_wait
        = sim.last_change = sim.L = 0.0;

    sim.messages = sim.cs_uses = sim.active
        = sim.maxact = sim.maxforward = 0;
}

/* trans:
   message transmission time
   will be constant or variable depending on command line arg */
double const_trans(void)
{
return AVG_MSG_TIME;
}

double var_trans(void)
{
    return expdis(AVG_MSG_TIME);
}

main(int argc, char *argv[])
{
    int i, j, k;
    int trials;
    char *names[] = {"BANTA", "LIU", "NAIMI", "SUZUKI",
                     "NISHIO", "AGRAB", "CHANG"};
    char *p;
    char line[80];
    int algs[ALGNUM] = {0};
    double rate, denom;
    double summsgpercs, sumwaitime, summaxwait, sumthrput;
    double smsgpercs, sswaitime, ssmaxwait, ssththrput;
    finals res[ALGNUM];

    /* check for proper number of command line args, and get seed for */
    /* random number generation */
    if ( (argc != 10) || !(seed = (unsigned int)atol(argv[1])) )
    {
        printf("\nusage: * /,s seed lb ub trials alg-digits"
               " breakd out var_msg_flag net-size\n",
               argv[0]);
        printf("\nalg-digit codes:\n"");
        for (i = 0; i < ALGNUM; i++)
            printf("%2d\t%5s\n", i, names[i]);
        exit(1);
    }

    /* number of trials on which to base confidence interval */
    trials = atoi(argv[4]);
    if (trials > MAXTRIALS || trials < 0) {
        fprintf(stderr, "trials>MAXTRIALS or neg.; rerun or recompile\n");
        exit(1);
    }

    /* alg-digits: string of digits; if the digit for an algorithm is */
    /* in the string then run that algorithm; e.g. a string 04 means */
    /* run MBB0 and SK */
for (p = argv[5]; *p; p++)
    if (isdigit(*p) && *p - '0' < ALGNUM)
        algs[*p - '0'] = 1;

/* breakt is average time between breakdowns at each site */
breakt = 1.0/atof(argv[6]);

/* downt is average amount of time it takes for a breakdown to */
/* be fixed */
downt = 1.0/atof(argv[7]);

/* var_msg_flag: 1 means use variable message transmission */
/* times, */
/* 0 means use constant (1.0) time */
if (atoi(argv[8]))
    msg_time = var_trans;
else
    msg_time = const_trans;

/* number of sites in the network */
N = atoi(argv[9]);

/* which levels of request traffic to use? go from lb to ub; */
/* e.g if lb is 0 and ub is 2, then use 0.001, 0.03 and 0.06 */
for (k = atoi(argv[2]); k <= atoi(argv[3]); k++) {
    switch (k) {
        case 0: rate = 0.001; break;
        case 1: rate = 0.03; break;
        case 2: rate = 0.06; break;
        case 3: rate = 0.09; break;
        case 4: rate = 0.10; break;
        case 5: rate = 0.12; break;
        case 6: rate = 0.15; break;
        case 7: rate = 0.20; break;
        case 8: rate = 0.50; break;
        case 9: rate = 1.0; break;
    }
    for (i = 0; i < trials; i++) {
        /* set request freq. at each site */
        for (j=0; j<N; j++)
            lambda[j] = expdis(1.0/rate);
for (j = 0; j < ALGNUM; j++)
    if (algs[j]) {
        switch (j) {
            case 0: banta(); break;
            case 1: liu(); break;
            case 2: naimi(); break;
            case 3: suzuaki(); break;
            case 4: nishio(); break;
            case 5: agrab(); break;
            case 6: chang(); break;
            default:
                printf("Bad number in algs\n");
                exit(1);
                break;
        }
    }

    res[j].mgpercs[i] =
        (double)sim.messages/(double)sim.cs_uses;
    res[j].waitime[i] =
        sim.total_wait/(double)sim.cs_uses;
    res[j].maxwait[i] = sim.max_wait;
    res[j].thrput[i] = (double)sim.cs_uses/ENDTIME;
}
}

 denom = (double)(trials*(trials-1));

for (j = 0; j < ALGNUM; j++)
    if (algs[j]) {
        summpercs = sumwaitime = summaxwait = sumthrput = 0.0;
        ssmperecs = sswaitime = ssmaxwait = ssthrput = 0.0;
        for (i = 0; i < trials; i++) {
            summpercs += res[j].mgpercs[i];
            sumwaitime += res[j].waitime[i];
            summaxwait += res[j].maxwait[i];
            sumthrput += res[j].thrput[i];

            ssmperecs += sqr(res[j].mgpercs[i]);
            sswaitime += sqr(res[j].waitime[i]);
            ssmaxwait += sqr(res[j].maxwait[i]);
            ssthrput += sqr(res[j].thrput[i]);
        }
        res[j].mgpercsmean = summpercs/(double)trials;
        res[j].waitimemean = sumwaitime/(double)trials;
res[j].maxwaitmean = summaxwait/(double)trials;
res[j].thrputmean = sumthrput/(double)trials;
res[j].mgpercssd =
sqrt((trials*ssmgpercs - sqrr(summgpercs))/denom);
res[j].waitime =
sqrt((trials*sswaitime - sqrr(sumwaitime))/denom);
res[j].maxwait =
sqrt((trials*ssmaxwait - sqrr(summaxwait))/denom);
res[j].thrputs =
sqrt((trials*ssthrput - sqrr(sumthrput))/denom);
}
printf("\nN is %d, lambda is %g, breakt is %g, %g\n\n\n\t",
N, rate, breakt, 1.0/breakt);
for (i=0;i<ALGNUM;i++)
  if (algs[i])
    printf("7.9s	", names[i]);
printf("\nmgpercs:");
for (i=0;i<ALGNUM;i++)
  if (algs[i])
    printf("7.6.3f	" , res[i].mgpercsmean,
            res[i].mgpercssd*CONFIDENCE/sqrt((double)trials));
printf("\nwaitime:");
for (i=0;i<ALGNUM;i++)
  if (algs[i])
    printf("7.6.3f	" , res[i].waitime,
            res[i].waitime =CONFIDENCE/sqrt((double)trials));
printf("\nmaxwait:");
for (i=0;i<ALGNUM;i++)
  if (algs[i])
    printf("7.6.3f	" , res[i].maxwait,
            res[i].maxwait =CONFIDENCE/sqrt((double)trials));
printf("\n\nthrput:	");
for (i=0;i<ALGNUM;i++)
  if (algs[i])
    printf("7.6.3f	" , res[i].thrput,
            res[i].thrput =CONFIDENCE/sqrt((double)trials));
printf("\n\n\n

*********************************
");
extern simst sim;
void outheap(heap *Hp);

/* createHp:
   initializes order of heap -- heap[0] to -infinity, rest of heap to
   +infinity. heap[0] will never change; top points to last element
   of array in active use. */
void createHp(heap *Hp, int size, int (*comp)(void *, void *),
              void (*outnode)(void *))
{
    Hp->list = (void **)myalloc(size+1, sizeof(void *));
    Hp->size = size;
    Hp->comp = comp;
    Hp->outnode = outnode;
    Hp->list[0] = NULL;
    Hp->top = 0;
}

void closeHp(heap *Hp)
{
    int i;
    for (i = 0; i <= Hp->top; i++)
        free(Hp->list[i]);
    free(Hp->list);
}

/* emptyHp(top)
   if top == 0 then no elements of array are in active use */
boolean emptyHp(heap *Hp)
{
    return Hp->top == 0;
}

/* swap(&a, &b) */
void swap(void **a, void **b)
{
    void *temp;
```c
temp = *a;
*a = *b;
*b = temp;
}

/* deHp(event)
   copy first element in heap order back to caller, remove that
   element from the heap, and restore heap order */
void deHp(heap *Hp, void **temp)
{
    int curr, child;

    if (emptyHp(Hp)) {
        printf("attempt to deHp empty list, program aborted\n");
        exit(1);
    }

    /*copy first element back to caller*/
    *temp = Hp->list[1];

    /*remove first element from heap by copying bottom element
    over it and decrementing top*/
    Hp->list[1] = Hp->list[Hp->top--];

    /*restore heap order*/
    curr = 1;
    child = 2;
    while (child <= Hp->top) {
        if (child < Hp->top &&
            (*Hp->comp)(Hp->list[child+1], Hp->list[child]) < 0)
            child++;
        if (((*Hp->comp)(Hp->list[curr], Hp->list[child]) > 0) {
            swap(&Hp->list[curr], &Hp->list[child]);
            curr = child;
            child = curr + curr;
        }
        else break;
    }
}

/* enHp(event)
   adds an event to the event-heap and restores heap order */
void enHp(heap *Hp, void *temp)
{```
int curr, par;

if ((Hp->top >= Hp->size) {
    outheap(Hp);
    printf("HEAP OVERFLOW, PROGRAM ABORTED at \n", sim.now);
    exit(1);
}

/* add new event as leaf */
curr = ++Hp->top;
Hp->list[curr] = temp;

par = curr/2;
while ((*Hp->comp)(Hp->list[par], Hp->list[curr]) > 0) {
    swap(&Hp->list[par], &Hp->list[curr]);
    curr = par;
    par = curr/2;
}

/* reposition: the value that temp points to has been changed, */
/* and heap order must be restored */
boolean reposition(heap *Hp, void *temp) {
    int curr, par, child;
    boolean success;
    for (curr = 1; curr <= Hp->top && Hp->list[curr] != temp; curr++)
        ;
    success = curr <= Hp->top;
    if (success) {
        par = curr/2;
        child = 2*curr;
        if ((*Hp->comp)(Hp->list[par], Hp->list[curr]) > 0)
            do {
                swap(&Hp->list[par], &Hp->list[curr]);
                curr = par;
                par = curr/2;
            } while ((*Hp->comp)(Hp->list[par], Hp->list[curr]) > 0);
        else
            while (child <= Hp->top) {
                if (child < Hp->top & &
boolean deleteEvt(heap *Hp, void *temp)
{
    int curr, par, child;
    boolean success;

    for (curr = 1; curr <= Hp->top && Hp->list[curr] != temp; curr++)
    {
        success = curr <= Hp->top;  
        if (success) {
            Hp->list[curr] = Hp->list[Hp->top--];
            par = curr/2;
            child = 2*curr;
            if ((*(Hp->comp)(Hp->list[par], Hp->list[curr]) > 0)
                do {
                swap(&Hp->list[par], &Hp->list[curr]);
                curr = par;
                par = curr/2;
            } while ((*(Hp->comp)(Hp->list[par], Hp->list[curr]) > 0);
        else
            while (child <= Hp->top) {
                if (child < Hp->top &&
                    (*(Hp->comp)(Hp->list[child+1], Hp->list[child]) < 0)  
                    child++;
                if ((*(Hp->comp)(Hp->list[curr], Hp->list[child]) > 0) {
                    swap(&Hp->list[curr], &Hp->list[child]);
                    curr = child;
                    child = curr + curr;
                }
            else break;
        }
    return success;
}
return success;
}

/* for debugging, mainly */
void outheap(heap *Hp)
{
  int i;
  void **p;

  p = Hp->list;

  for (i = 1; i<=Hp->top; i++)
    (*Hp->outnode)(p[i]);
}

/* queue.c */

#include "mutex.h"

extern int N;

/* createQC&Q */
void createQ(Qhdr *Q)
{
  Q->head = Q->tail = NULL;
}

/* emptyQ(Q)
  returns true if Q is empty */
boolean emptyQ(Qhdr Q)
{
  return (Q.head == NULL);
}

/* returns key of first element in Q */
int firstQ(Qhdr Q)
{
  if (!emptyQ(Q))
    return Q.head->site;
  else
    return -1;
}
return -1;
}

/* returns key of last element in Q */
int lastQ(Qhdr Q)
{
    if (!emptyQ(Q))
        return Q.tail->site;
    else
        return -1;
}

/* deQ(&Q)
remove first element of Q and discard it; do not send it back to caller */
void deQ(Qhdr *Q)
{
    Qnode *temp;

    if (emptyQ(*Q)) {
        printf("ERROR - attempt to deQ empty Q\n");
        exit(1);
    }

    temp = Q->head;
    Q->head = Q->head->next;
    if (Q->head == NULL)
        Q->tail = NULL;
    free(temp);
}

/* enQ(site, &Q)
create new Qnode and add it to back of Q */
void enQ(int site, Qhdr *Q)
{
    Qnode *temp;

    temp = (Qnode *)malloc(sizeof(Qnode));
    temp->site = site;

    if (emptyQ(*Q)) {
        temp->next = NULL;
        Q->head = Q->tail = temp;
    }
else {
    temp->next = NULL;
    Q->tail->next = temp;
    Q->tail = temp;
}
}

/* append(&localQ, &tokenQ)
    adds back Q to front Q, and leaves back Q empty */
void append(Qhdr *back, Qhdr *front)
{
    if (emptyQ(*back))
        return;
    if (emptyQ(*front))
        front->head = back->head;
    else
        front->tail->next = back->head;

    front->tail = back->tail;

    back->head = back->tail = NULL;
}

/* search(Q, onQ)
    set onQ[i] = true if site i is in Q; otherwise false */
void search(Qhdr Q, boolean onQ[])
{
    Qnode *temp;
    int i;

    for (i=0; i<N; i++)
        onQ[i] = false;
    temp = Q.head;
    while (temp != NULL) {
        onQ[temp->site] = true;
        temp = temp->next;
    }
}

 rand.c

#include <stdio.h>
#include <math.h>
#include <limits.h>
#include <float.h>

#define MULT 65539

/* for random #'s and seed */
extern unsigned int seed;

/* ranf() 
   returns U(0,1) random number */
double ranf(void)
{
    float u;

    seed = seed*MULT;
    u = (float)seed/(float)UINT_MAX;
    return (double)u;
}

/* expdis(alpha) 
   returns random number from exponential distribution 
   with parameter alpha */
double expdis(double alpha)
{
    double u;

    u = ranf();
    if (u != 0.0)
        u = (-1.0/alpha)*log(u);
    else
        u = DBL_MAX;

    return u;
}

banta.c

algorithm MBBO
Makki, Banta, Been and Ogawa, "Two algorithms for mutual exclusion in a distributed system", _Proceedings of the International Conference on Parallel Processing_, pp. 1460-1466, August 1991
#include "mutex.h"

#define MSGDEL 0.0
#define WAITIME 2.0

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double breakt, downt;
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

/* for Banta, et. al.; local globals */
enum statetype {Idle, Requesting, Using, Done, Holding};
typedef enum statetype statetype;

static int good[MAXNET];
static statetype state[MAXNET];
static Qhdr tokenQ, localQ[MAXNET];
static int mark;

static void schedule_end_sim(double time)
{
    event *temp;

    temp = alloc_event(EndSim, 0, time);
    enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(NeedArises, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(ExitCS, nearsite, sim.now + time);
enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite, double time)
{
  event *temp;
  temp = alloc_event(ReceiveToken, nearsite, sim.now + time);
  enHp(&eventHp, temp);
}

static void schedule_end_wait(int nearsite, double time)
{
  event *temp;
  temp = alloc_event(EndWait, nearsite, sim.now + time);
  enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite, long int seqnum, double time)
{
  event *temp;
  temp = alloc_event(ReceiveRequest, nearsite, sim.now + time);
  temp->farsite = farsite;
  temp->seqnum1 = seqnum;
  enHp(&eventHp, temp);
}

static void schedule_receive_update(int nearsite, int farsite, double time)
{
  event *temp;
  temp = alloc_event(ReceiveUpdate, nearsite, sim.now + time);
  temp->farsite = farsite;
  enHp(&eventHp, temp);
}

/* init_banta(&sim)
   initializes arrays and queues according to algorithm and
   schedules first events */
void init_banta(void)
97

```c
{
    int i;

    createQ(&tokenQ);
    createHp(&eventHp, 3*N, comp_evt, out_evt);
    init_sim();

    for (i=0; i<N; i++) {
        good[i] = 0;
        state[i] = Idle;
        createQ(&(localQ[i]));
        schedule_need_arises(i, expdis(lambda[i]));
    }

    state[0] = Done;
    mark = 0;

    schedule_end_sim(ENDTIME);
}

/* send_token(me, heap, &top)
   called upon exiting CS, and if receive request while sitting
   idle with token; algorithm described by Banta, et. al. */
static void send_token(int me) {
    int i;
    boolean onQ[MAXNET];
    double time;

    if (!emptyQ(localQ[me])) {
        append(&(localQ[me]), &tokenQ);
        search(tokenQ, onQ);
        time = 0.0;
        for (i=0; i<N; i++)
            if (!onQ[i] && (i != me)) {
                schedule_receive_update(i, lastQ(tokenQ), time+TRANS);
                time += MSGDEL;
            }
        schedule_end_wait(me, time+WAITIME);
        mark = good[me] = lastQ(tokenQ);
        state[me] = Holding;
    }
    else if (!emptyQ(tokenQ)) {
        if (mark == me) {
            
```
search(tokenQ, onQ);
time = 0.0;
for (i=0; i<N; i++)
    if (!onQ[i] && (i != me)) {
        schedule_receive_update(i, lastQ(tokenQ), time+TRANS);
        time += MSGDEL;
    }
schedule_end_wait(me, time+WAITIME);
mark = good[me] = lastQ(tokenQ);
state[me] = Holding;
} else {
    good[me] = mark;
schedule_receive_token(firstQ(tokenQ), TRANS);
    state[me] = Idle;
}
else {
    state[me] = Done;
}
}

static void need_arises(int me)
{
    sim.L += (sim.now - sim.last_change)*sim.active;
sim.last_change = sim.now;
++(sim.active);
if (sim.maxact < sim.active)
    sim.maxact = sim.active;
if (state[me] == Done) { /* I already have token */
    state[me] = Using;
schedule_exit_cs(me, CSTIME);
    return;
}
if (state[me] == Holding)
    enQ(me, &(localQ[me]));
else
    schedule_receive_request(good[me], me, 0, TRANS);
state[me] = Requesting;
sim.start_wait[me] = sim.now;
}
static void receive_request(int me, int sender, long int seqnum)
{
    ++(sim.messages);

    if (state[me] == Idle) {
        schedule_receive_request(good[me], sender, seqnum+1, TRANS);
        sim.maxforward = max(seqnum+1,sim.maxforward);
    }
    
    else {
        enQ(sender, &(localQ[me]));
        if (state[me] == Done)
            send_token(me);
    }
}

static void receive_token(int me)
{
    double wait;

    ++(sim.messages);

    deQ(&tokenQ);
    state[me] = Using;

    wait = sim.now - sim.start_wait[me];
    sim.total_wait += wait;
    sim.max_wait = max(wait, sim.max_wait);

    schedule_exit_cs(me, CSTIME);
}

static void receive_update(int me, int goodsite)
{
    ++(sim.messages);
    if (state[me] == Idle)
        good[me] = goodsite;
}

static void exit_cs(int me)
{
    schedule_need_arises(me, expdis(lambda[me]));
    ++(sim.cs_uses);
--(sim.active);

send_token(me);
}

static void end_wait(int me)
{
    append(&(localQ[me]), &tokenQ);
    schedule_receive_token(firstQ(tokenQ), TRANS);
    if (state[me] != Requesting)
        state[me] = Idle;
}

static void end_sim(void)
{
    int i;
    double wait;

    sim.L += (sim.now - sim.last_change)*sim.active;
    for (i=0; i<N; i++)
        if (state[i] == Requesting) {
            wait = sim.now - sim.start_wait[i];
            sim.total_wait += wait;
            sim.max_wait = max(wait, sim.max_wait);
        }

    while (!emptyQ(tokenQ))
        deQ(&tokenQ);
    for (i=0; i<N; i++)
        while (!emptyQ(localQ[i]))
            deQ(&(localQ[i]));
    closeHp(&eventHp);
}

void banta(void)
{
    event *temp;
    boolean done = false;

    init_banta();

    do {
deHp(&eventHp, (void **)(temp));

sim.now = temp->time;
switch (temp->code) {
    case NeedArises:
        need_arises(temp->nearsite);
        break;
    case ReceiveRequest:
        receive_request(temp->nearsite, temp->farsite,
                        temp->seqnum);
        break;
    case ReceiveToken:
        receive_token(temp->nearsite);
        break;
    case ReceiveUpdate:
        receive_update(temp->nearsite, temp->farsite);
        break;
    case ExitCS:
        exit_cs(temp->nearsite);
        break;
    case EndWait:
        end_wait(temp->nearsite);
        break;
    case EndSim:
        end_sim();
        done = true;
        break;
    default:
        printf("Strange number in banta.\n");
        exit(1);
        break;
    }
    free(temp);
} while (!done);

/********************
liu.c

algorithm CSL
Ye In Chang and Mukesh Singhal and Ming T. Liu
"A Dynamic Token-Based Mutual Exclusion Algorithm"
_Proceedings of the 10th Annual International International
Phoenix Conference on Computers and Communications_
include "mutex.h"

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

/* local globals */
static boolean R[MAXNET][MAXNET], RequestingToken[MAXNET], Executing[MAXNET], HaveToken[MAXNET];
static long int Seq[MAXNET][MAXNET], TSeq[MAXNET];
static Qhdr TQ;

static void schedule_end_sim(double time)
{
    event *temp;

    temp = alloc_event(EndSim, 0, time);
    enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(NeedArises, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(ExitCS, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite, double time)
{
    event *temp;
temp = alloc_event(ReceiveToken, nearsite, sim.now + time);
enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite,
        long int seqnum, double time)
{
    event *temp;
    temp = alloc_event(ReceiveRequest, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum = seqnum;
    enHp(&eventHp, temp);
}

void init_liu(void)
{
    int i, j;

    init_sim();
    createHp(&eventHp, N*(N+1), comp_evt, out_evt);
    createQ(&TQ);

    for (i=0;i<N;i++) {
        for (j=0;j<N;j++) {
            R[i][j] = true;
            Seq[i][j] = 0;
        }
        RequestingToken[i] = Executing[i] = HaveToken[i] = false;
        schedule_need_arises(i, expdis(lambda[i]));
    }

    HaveToken[0] = true;
    for (j=0;j<N;j++) {
        R[0][j] = false;
        TSeq[j] = 0;
    }

    schedule_end_sim(ENDTIME);
}

static void need_arises(int me)
{
int i;
double time;

sim.L += (sim.now - sim.last_change)*sim.active;
sim.last_change = sim.now;
++(sim.active);
if (sim.maxact < sim.active)
    sim.maxact = sim.active;
sim.start_wait[me] = sim.now;

RequestingToken[me] = true;
if (!HaveToken[me]) {
    ++Seq[me][me];
time = 0.0;
    for (i=0;i<N;i++)
        if (R[me][i])
            schedule_receive_request(i, me, Seq[me][me], TRANS);
}
else {
    RequestingToken[me] = false;
    Executing[me] = true;
    schedule_exit_cs(me, CSTIME);
}
}

static void receive_token(int me)
{
    int i;
double wait;

    ++sim.messages;
    wait = sim.now - sim.start_wait[me];
sim.total_wait += wait;
sim.max_wait = max(wait, sim.max_wait);

    for (i=0;i<N;i++)
        R[me][i] = false;
    HaveToken[me] = true;
    RequestingToken[me] = false;
    Executing[me] = true;
    schedule_exit_cs(me, CSTIME);
}

static void exit_cs(int me)
{  
  int i;
  boolean onQ[MAXNET];

  schedule_need_arises(me, expdis(lambda[me]));
  ++sim.cs_uses;
  --sim.active;

  Executing[me] = false;
  TSeq[me] = Seq[me][me];

  search(TQ, onQ);
  for (i=0;i<N;i++)
    if (!onQ[i] && (Seq[me][i] == TSeq[i] + 1))
      enQ(i, &TQ);

  search(TQ, onQ);
  for (i=0;i<N;i++)
    if (onQ[i])  R[me][i] = true;

  if (!emptyQ(TQ)) {
    HaveToken[me] = false;
    i = firstQ(TQ);
    deQ(&TQ);
    schedule_receive_token(i, TRANS);
  }
}

static void receive_request(int me, int X, long int XSeq)
{
  ++sim.messages;

  if (XSeq > Seq[me][X]) {
    Seq[me][X] = XSeq;
    if (HaveToken[me] && !Executing[me] &&
        (Seq[me][X] == TSeq[X] + 1)) {
      HaveToken[me] = false;
      schedule_receive_token(X, TRANS);
    }
    else if (RequestingToken[me] && !R[me][X])
      schedule_receive_request(X, me, Seq[me][me], TRANS);
    R[me][X] = true;
  }
}
static void end_sim(void)
{
    int i;
    double wait;

    sim.L += (sim.now - sim.last_change)*sim.active;
    for (i=0; i<N; i++)
        if (RequestingToken[i] && !HaveToken[i]) {
            wait = sim.now - sim.start_wait[i];
            sim.total_wait += wait;
            sim.max_wait = max(wait, sim.max_wait);
        }
    while (!emptyQ(TQ))
        deQ(&TQ);
    closeHp(&eventHp);
}

static void out_dbg(void)
{
    /* outheap(&eventHp); */
}

void liu(void)
{
    event *temp;
    boolean done = false;

    init_liu();

    do {

        deHp(&eventHp, (void **)(temp));
        sim.now = temp->time;

        switch (temp->code) {
        case NeedArises:
            need_arises(temp->nearsite);
            break;
        case ReceiveRequest:
            receive_request(temp->nearsite, temp->farsite, temp->seqnum1);
            break;
        case ReceiveToken:
            receive_token(temp->nearsite);
        }
break;
case ExitCS:
    exit_cs(temp->nearsite);
    break;
case EndSim:
    end_sim();
    done = true;
    break;
default:
    printf("Strange number in liu.\n");
    exit(1);
    break;
}
free(temp);
} while (!done);

Algorithm NT
Mohamed Naimi and Michel Trehel
"An Improvement of the log N Distributed Algorithm for Mutual Exclusion"
Proceedings of the 7th International Conference on Distributed Computing Systems,
1987, pp. 371-375

#include "mutex.h"

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double breakt, downt;
extern double (*msg_time)(void);
extern heap_eventHp;
extern simst sim;

/* local globals */
static Qhdr tokenQ, localQ[MAXNET];
static int last[MAXNET];
static boolean requesting_cs[MAXNET], tokenpresent[MAXNET];

static void schedule_end_sim(double time)


```c
{
    event *temp;
    temp = alloc_event(EndSim, 0, time);
    enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;
    temp = alloc_event(NeedArises, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, double time)
{
    event *temp;
    temp = alloc_event(ExitCS, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite, double time)
{
    event *temp;
    temp = alloc_event(ReceiveToken, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite, double time)
{
    event *temp;
    temp = alloc_event(ReceiveRequest, nearsite, sim.now + time);
    temp->farsite = farsite;
    enHp(&eventHp, temp);
}

void init_naiimi(void)
{
    int i;
}```
createQ(&tokenQ);
createHp(&eventHp, 2*N, comp_evt, out_evt);
init_sim();

for (i=0;i<N;i++) {
  createQ(&localQ[i]);
  last[i] = 0;
  requesting_cs[i] = false;
  tokenpresent[i] = (last[i] == i);
  schedule_need_arises(i, expdis(lambda[i]));
}
schedule_end_sim(ENDTIME);

static void need_arises(int me)
{
  sim.L += (sim.now - sim.last_change)*sim.active;
  sim.last_change = sim.now;
  ++sim.active;
  if (sim.maxact < sim.active)
    sim.maxact = sim.active;
  sim.start_wait[me] = sim.now;

  requesting_cs[me] = true;
  if (last[me] != me) {
    schedule_receive_request(last[me], me, TRANS);
    last[me] = me;
  } else
    schedule_exit_cs(me, CSTIME);
}

static void receive_token(int me)
{
  double wait;

  ++sim.messages;
  wait = sim.now - sim.start_wait[me];
  sim.total_wait += wait;
  sim.max_wait = max(wait, sim.max_wait);
  if (sim.max_wait == wait) {
    sim.start_big_wait = sim.start_wait[me];
    sim.big_wait_site = me;
  }
}
append(&localQ[me], &tokenQ);
tokenpresent[me] = true;
schedule_exit_cs(me, CSTIME);
}

static void exit_cs(int me)
{
    int next;
    schedule_need_arises(me, expdis(lambdame[me]));
    ++(sim.cs_uses);
    --(sim.active);

    requesting_cs[me] = false;
    append(&localQ[me], &tokenQ);
    if (!emptyQ(tokenQ)) {
        next = firstQ(tokenQ);
        last[me] = lastQ(tokenQ);
        deQ(&tokenQ);
        tokenpresent[me] = false;
        schedule_receive_token(next, TRANS);
    }
}

static void receive_request(int me, int sender)
{
    ++sim.messages;

    if (requesting_cs[me])
        enQ(sender, &localQ[me]);
    else {
        if (tokenpresent[me]) {
            tokenpresent[me] = false;
            schedule_receive_token(sender, TRANS);
        } else
            schedule_receive_request(last[me], sender, TRANS);
    last[me] = sender;
}
}

static void end_sim(void)
{ 
  int i;
  double wait;

  sim.L += (sim.now - sim.last_change)*sim.active;
  for (i=0; i<N; i++)
    if (requesting_cs[i] && !tokenpresent[i]) {
      wait = sim.now - sim.start_wait[i];
      sim.max_wait = max(wait, sim.max_wait);
      if (sim.max_wait == wait) {
        sim.start_big_wait = sim.start_wait[i];
        sim.big_wait_site = i;
      }
      sim.total_wait += wait;
    }
  while (!emptyQ(tokenQ))
    deQ(&tokenQ);
  for (i=0; i<N; i++)
    while (!emptyQ(localQ[i]))
      deQ(&(localQ[i]));
  closeHp(&eventHp);
}

void naimi(void)
{
  event *temp;
  boolean done = false;

  init_naimi();

  do {
    deHp(&eventHp, (void **)(&temp));
    sim.now = temp->time;

    switch (temp->code) {
    case NeedArises:
      need_arises(temp->nearsite);
      break;
    case ReceiveRequest:
      receive_request(temp->nearsite, temp->farsite);
      break;
    case ReceiveToken:
      receive_token(temp->nearsite);
      break;
    }"
case ExitCS:
    exit_cs(temp->nearsite);
    break;

case EndSim:
    end_sim();
    done = true;
    break;

default:
    printf("Strange number in naimi.
")
    exit(1);
    break;
}
free(temp);
} while (!done);

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

/* global globals */
extern int N;
extern double lambda[MAXNODE];
extern double breakt, downt;
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

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

static void schedule_end_sim(double time)
{
    event *temp;

temp = alloc_event(EndSim, 0, time);
enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(NeedArises, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(ExitCS, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(ReceiveToken, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite,
    long int seqnum, double time)
{
    event *temp;

    temp = alloc_event(ReceiveRequest, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum1 = seqnum;
    enHp(&eventHp, temp);
}

void init_sk(void)
{
    int i, j;

    init_sim();
createHp(&eventHp, N*(N+1), comp_evt, out_evt);
createQ(&tokenQ);

for (i=0; i<N; i++) {
    HaveToken[i] = RequestingToken[i] = false;
    LN[i] = -1;
    for (j=0; j<N; j++)
        RN[i][j] = -1;
    schedule_need_arises(i, expdis(lambda[i]));
}

HaveToken[0] = true;

schedule_end_sim(ENDTIME);
}

static void need_arises(int me) {
    int i;
    double time;

    sim.L += (sim.now - sim.last_change)*sim.active;
    sim.last_change = sim.now;
    ++(sim.active);
    if (sim.maxact < sim.active)
        sim.maxact = sim.active;
    sim.start_wait[me] = sim.now;

    RequestingToken[me] = true;

    if (!HaveToken[me]) {
        ++RN[me][me];
        for (i=0; i<N; i++)
            if (i != me)
                schedule_receive_request(i, me, RN[me][me], TRANS);
    } else
        schedule_exit_cs(me, CSTIME);
}

static void receive_token(int me) {
    double wait;


```java
++(sim.messages);
wait = sim.now - sim.start_wait[me];
sim.total_wait += wait;
sim.max_wait = max(wait, sim.max_wait);

HaveToken[me] = true;
schedule_exit_cs(me, CSTIME);
}

static void exit_cs(int me)
{
    int i;
    boolean onQ[MAXNET];

    schedule_need_arises(me, expdis(lambda[me]));
    ++(sim.cs_uses);
    --(sim.active);

    LN[me] = RN[me][me];

    search(tokenQ, onQ);

    for(i=0; i<N; i++)
        if ((!onQ[i]) && (RN[me][i] == LN[i]+1))
            enQ(i, &tokenQ);

    if (!emptyQ(tokenQ)) {
        HaveToken[me] = false;
        i = firstQ(tokenQ);
        deQ(&tokenQ);
        schedule_receive_token(i, TRANS);
    }

    RequestingToken[me] = false;
}

static void receive_request(int me, int sender, int seqnum)
{
    ++(sim.messages);

    RN[me][sender] = max(RN[me][sender], seqnum);

    if (HaveToken[me] && !RequestingToken[me] &&
```
(RN[me][sender] == LN[sender]+1)) {
    HaveToken[me] = false;
    schedule_receive_token(sender, TRANS);
}
}

static void end_sim(void)
{
    int i;
    double wait;

    sim.L += (sim.now - sim.last_change)*sim.active;
    for (i=0; i<N; i++)
        if (RequestingToken[i] && !HaveToken[i]) {
            wait = sim.now - sim.start_wait[i];
            sim.total_wait += wait;
            sim.max_wait = max(wait, sim.max_wait);
        }
    while (!emptyQ(tokenQ))
        deQ(&tokenQ);
    closeHp(&eventHp);
}

void suzuki(void)
{
    event *temp;
    boolean done = false;

    init_sk();

    do {
        deHp(&eventHp, (void **)&temp);
        sim.now = temp->time;
        switch (temp->code) {
            case NeedArises:
                need_arises(temp->nearsite);
                break;
            case ReceiveRequest:
                receive_request(temp->nearsite, temp->farsite, temp->seqnum1);
                break;
            case ReceiveToken:
                receive_token(temp->nearsite);
                break;
        }
    }
case ExitCS:
    exit_cs(temp->nearsite);
    break;
case EndSim:
    end_sim();
    done = true;
    break;
default:
    printf("Strange number in suzuki.\n");
    exit(1);
    break;
}
free(temp);
} while (!done);

/******************************
chang.c

algorithm CSL-2
Chang, Singhal and Liu, "An improved $O(\log N)$ mutual exclusion
algorithm for distributed systems", _Proceedings of the
International Conference on Parallel Processing_, pp.
III-295--III-302, 1990
******************************/
#include "mutex.h"

#define NULLSITE -1

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

/* for chang et. al. */
static int NewRoot[MAXNET], Next[MAXNET];
static boolean RequestingToken[MAXNET], HaveToken[MAXNET];

static void schedule_end_sim(double time)
{
    event *temp;
temp = alloc_event(EndSim, 0, time);
    enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(NeedArises, nearsite, sim.now+time);
    enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite, int farsite,
                                    double time)
{
    event *temp;

    temp = alloc_event(ReceiveToken, nearsite, sim.now+time);
    temp->farsite = farsite;
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(ExitCS, nearsite, sim.now+time);
    enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite,
                                      double time)
{
    event *temp;

    temp = alloc_event(ReceiveRequest, nearsite, sim.now+time);
    temp->farsite = farsite;
    enHp(&eventHp, temp);
}

static void init_chang()
int i;

init_sim();
createHp(&eventHp, 2*N, comp_evt, out_evt);

for (i=0; i<N; i++) {
    HaveToken[i] = false;
    RequestingToken[i] = false;
    NewRoot[i] = 0;
    Next[i] = NULLSITE;
    schedule_need_arises(i, expdis(lambda[i]));
}
HaveToken[0] = true;

schedule_end_sim(ENDTIME);
}

/* note -- in this algorithm, for "ReceiveToken" events, the "farsite" field in the event contains the NewRoot information which is passed with the token */
static void need_arises(int me)
{
    sim.L += (sim.now - sim.last_change)*sim.active;
    sim.last_change = sim.now;
    ++(sim.active);
    if (sim.maxact < sim.active)
        sim.maxact = sim.active;
    sim.start_wait[me] = sim.now;

    RequestingToken[me] = true;
    if (!HaveToken[me]) {
        schedule_receive_request(NewRoot[me], me, TRANS);
        NewRoot[me] = me;
    }
    else
        schedule_exit_cs(me, CSTIME);
}

static void receive_token(int me, int newroot)
{
    double wait;

    ++(sim.messages);
    wait = sim.now - sim.start_wait[me];
sim.total_wait += wait;
sim.max_wait = max(wait, sim.max_wait);

HaveToken[me] = true;
if (newroot != me)
    NewRoot[me] = newroot;
schedule_exit_cs(me, CSTIME);
}

static void exit_cs(int me)
{
    schedule_need_arises(me, expdis(lambda[me]));
    ++(sim.cs_uses);
    --(sim.active);

    if (Next[me] != NULLSITE) {
        HaveToken[me] = false;
        schedule_receive_token(Next[me], NewRoot[me], TRANS);
        Next[me] = NULLSITE;
    }
    RequestingToken[me] = false;
}

static void receive_request(int me, int sender)
{
    ++(sim.messages);

    if (RequestingToken[me]) {
        if (Next[me] == NULLSITE)
            Next[me] = sender;
        else
            schedule_receive_request(NewRoot[me], sender, TRANS);
    }
    else
        if (HaveToken[me]) {
            HaveToken[me] = false;
            schedule_receive_token(sender, sender, TRANS);
        }
        else
            schedule_receive_request(NewRoot[me], sender, TRANS);
    NewRoot[me] = sender;
}

static void end_sim(void)
{  
  int i;
  double wait;

  sim.L += (sim.now - sim.last_change)*sim.active;
  for (i=0; i<N; i++)
    if (RequestingToken[i] && !HaveToken[i]) {
      wait = sim.now - sim.start_wait[i];
      sim.total_wait += wait;
      sim.max_wait = max(wait, sim.max_wait);
    }
  closeHp(&eventHp);
}

void chang(void)
{
  event *temp;
  boolean done = false;

  init_chang();

  do {
    deHp(&eventHp, (void **)(&temp));
    sim.now = temp->time;

    switch (temp->code) {
      case NeedArises:
        need_arises(temp->nearsite);
        break;
      case ReceiveRequest:
        receive_request(temp->nearsite, temp->farsite);
        break;
      case ReceiveToken:
        receive_token(temp->nearsite, temp->farsite);
        break;
      case ExitCS:
        exit_cs(temp->nearsite);
        break;
      case EndSim:
        end_sim();
        done = true;
        break;
    }
  } free(temp);
while (!done);
}

algorithm NLM
Shojiro Nishio and Kin F. Li and Eric G. Manning
"A Resilient Mutual Exclusion Algorithm for Computer Network"
_IEEE Transactions on Parallel and Distributed Systems_
July, 1990, pp. 344-355
******************************************************************************
#include "mutex.h"

#define _tm (4*AVG_MSG_TIME)
#define _tc CSTIME
#define _td 0.0
#define _te 0.0
#define _tresp (2*tm + _te)
#define _tmax ((N+1)*_tm + (N-1)*_tc + _td)

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double breakt, downt;
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

/* local globals */
/* these are part of the algorithm specification */
static long int CR_log[MAXNET], CR_request[MAXNET][MAXNET],
           curr_token_age, site_age[MAXNET];

/* these are for simulation bookkeeping */
static event *timer[MAXNET];
static long int cs_log[MAXNET], ack_rec[MAXNET][MAXNET],
           temp_crlog[MAXNET][MAXNET];
static boolean token_present[MAXNET], site_up[MAXNET];
static boolean cs_in_use;

static void schedule_end_sim(double time)
{  
event *temp;

temp = alloc_event(EndSim, 0, time);
enHp(&eventHp, temp);
}

static void schedule_need_arises(int nearsite, double time)
{
  event *temp;

  temp = alloc_event(NeedArises, nearsite, sim.now + time);
enHp(&eventHp, temp);
}

static void schedule_recovery(int nearsite, double time)
{
  event *temp;

  temp = alloc_event(Recovery, nearsite, sim.now + time);
enHp(&eventHp, temp);
}

static void schedule_break_down(int nearsite, double time)
{
  event *temp;

  temp = alloc_event(BreakDown, nearsite, sim.now + time);
enHp(&eventHp, temp);
}

static void schedule_receive_token(int nearsite,
                                    long int seqnum, double time)
{
  event *temp;

  temp = alloc_event(ReceiveToken, nearsite, sim.now + time);
  temp->seqnum1 = seqnum;
enHp(&eventHp, temp);
}

static void cancel_timer(int me)
{
  if (!delete_evt(&eventHp, (void *)&(timer[me]))) {

printf("ERROR -- timers are mixed up.\n");
exit(1);
}
free(timer[me]);
timer[me] = NULL;
}

static void schedule_time_out(int nearsite, double time) {
    if (!timer[nearsite]) {
        timer[nearsite] = alloc_event(TimeOut, nearsite, 
                                     sim.now + time);
        enHp(&eventHp, timer[nearsite]);
    }
    else {
        timer[nearsite]->time = sim.now + time;
        if (!reposition(&eventHp, (void *)(timer[nearsite]))) {
            printf("ERROR -- timers are mixed up!\n");
            exit(1);
        }
    }
}

static void schedule_exit_cs(int nearsite, 
                             long int seqnum, double time) {
    event *temp;
    temp = alloc_event(ExitCS, nearsite, sim.now + time);
    temp->seqnum1 = seqnum;
    enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite, 
                                     long int seqnum, double time) {
    event *temp;
    temp = alloc_event(ReceiveRequest, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum1 = seqnum;
    enHp(&eventHp, temp);
}
static void schedule_receive_missing(int nearsite, int farsite, long int seqnum, double time) {
    event *temp;
    temp = alloc_event(ReceiveMissing, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum1 = seqnum;
    enHp(&eventHp, temp);
}

static void schedule_receive_nack(int nearsite, int farsite, long int seqnum, double time) {
    event *temp;
    temp = alloc_event(ReceiveNack, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum1 = seqnum;
    enHp(&eventHp, temp);
}

static void schedule_receive_ack(int nearsite, int farsite, long int seqnum1, long int seqnum2, double time) {
    event *temp;
    temp = alloc_event(ReceiveAck, nearsite, sim.now + time);
    temp->farsite = farsite;
    temp->seqnum1 = seqnum1;
    temp->seqnum2 = seqnum2;
    enHp(&eventHp, temp);
}

static void init_nishio(void) {
    int i, j;

    init_sim();
    createHp(&eventHp, 2*N*N, comp_evt, out_evt);
    curr_token_age = 0;
    cs_in_use = false;
for (i = 0; i < N; i++) {
    CR_log[i] = 0;
    timer[i] = NULL;
    site_up[i] = true;
    site_age[i] = 0;
    token_present[i] = (i == 0);
    cs_log[i] = 0;
    schedule_need_arises(i, expdis(lambda[i]));
    schedule_break_down(i, BREAKTIME);
    for (j = 0; j < N; j++) {
        CR_request[i][j] = 0;
        ack_rec[i][j] = 0;
        temp_crlog[i][j] = 0;
    }
    schedule_end_sim(ENDTIME);
}

static void need_arises(int me)
{
    int j;

    if(site_up[me]) {
        sim.L += (sim.now - sim.last_change)*sim.active;
        sim.last_change = sim.now;
        ++sim.active;
        if (sim.maxact < sim.active)
            sim.maxact = sim.active;
        sim.start_wait[me] = sim.now;

        ++CR_request[me][me];
        if (!token_present[me]) {
            for (j = 0; j < N; j++)
                if (j != me)
                    schedule_receive_request(j, me, CR_request[me][me], TRANS);
            schedule_time_out(me, _tmax);
        } else {
            schedule_exit_cs(me, ++cs_log[me], CSTIME);
            cs_in_use = true;
        }
    } else
        schedule_need_arises(me, expdis(lambda[me]));
static receive_request(int me, int sender, long int seqnum)
{
  ++sim.messages;
  if (site_up[me] && seqnum > CR_request[me][sender]) {
    CR_request[me][sender] = seqnum;
    if (token_present[me] && !cs_in_use &&
        CR_log[sender] < CR_request[me][sender]) {
      schedule_receive_token(sender, curr_token_age, TRANS);
      token_present[me] = false;
    }
  }
}

static void receive_ack(int me, int sender, long int propose_age,
                         long int lognum)
{
  boolean regenerate, all_rec;
  int i;
  double wait;
  ++sim.messages;
  if (site_up[me] && propose_age == site_age[me]) {
    ack_rec[me][sender] = propose_age;
    temp_crlog[me][sender] = lognum;
    regenerate = true;
    all_rec = true;
    for (i = 0; i < N && (regenerate || all_rec); i++)
      if (ack_rec[me][i] < 0)
        regenerate = false;
      else if (my_abs(ack_rec[me][i]) < propose_age)
        all_rec = false;
    if (all_rec && regenerate) {
      curr_token_age = propose_age;
      cancel_timer(me);
      wait = sim.now - sim.start_wait[me];
      sim.total_wait += wait;
      sim.max_wait = max(wait, sim.max_wait);

      token_present[me] = true;
      cs_in_use = true;
      schedule_exit_cs(me, ++cs_log[me], CSTIME);
    }
  }
}
} else if (all_rec) {
    for (i = 0; i < N; i++)
        if (i != me)
            schedule_receive_request(i, me, CR_request[me][me], TRANS);
    schedule_time_out(me, _tmax);
}
}

static void receive_nack(int me, int sender, long int propose_age)
{
    boolean all_rec;
    int i;

    ++sim.messages;

    if (site_up[me] && propose_age == site_age[me]) {
        ack_rec[me][sender] = -propose_age;
        all_rec = true;
        for (i = 0; i < N && all_rec; i++)
            if (my_abs(ack_rec[me][i]) < propose_age)
                all_rec = false;
        if (all_rec) {
            for (i = 0; i < N; i++)
                if (i != me)
                    schedule_receive_request(i, me, CR_request[me][me], TRANS);
            schedule_time_out(me, _tmax);
        }
    }
}

static void time_out(int me)
{
    int i;
    long int propose_age;

    timer[me] = NULL;
    if (site_up[me]) {
        propose_age = (site_age[me]/N)*N + me;
        if (propose_age <= site_age[me])
            propose_age += N;
        site_age[me] = propose_age;
        for (i = 0; i < N; i++)
            }
if (i != me)
    schedule_receive_missing(i, me, propose_age, TRANS);
schedule_time_out(me, _tresp);
ack_rec[me][me] = propose_age;
}

static void receive_missing(int me, int sender, long int propose_age)
{
    ++sim.messages;

    if (site_up[me])
        if (propose_age <= site_age[me])
            schedule_receive_nack(sender, me, propose_age, TRANS);
        else if (token_present[me]) {
            site_age[me] = propose_age;
curr_token_age = propose_age;
schedule_receive_nack(sender, me, propose_age, TRANS);
        } 
        else {
            if (CR_request[me][me] > CR_log[me])
                schedule_time_out(me, _tmax);
            site_age[me] = propose_age;
schedule_receive_ack(sender, me, propose_age, CR_log[me], TRANS);
        }
}

static void receive_token(int me, long int this_token_age)
{
    double wait;
    ++sim.messages;

    if (site_up[me] && this_token_age >= site_age[me]) {
        site_age[me] = this_token_age;
cancel_timer(me);
        wait = sim.now - sim.start_wait[me];
sim.total_wait += wait;
sim.max_wait = max(wait, sim.max_wait);

token_present[me] = true;
cs_in_use = true;
schedule_exit_cs(me, ++cs_log[me], CSTIME);
}
static void exit_cs(int me, long int seqnum)
{
    int j;
    boolean token_sent;

    if (seqnum == cs_log[me]) {
        ++sim.cs_uses;
        sim.L += (sim.now - sim.last_change)*sim.active;
        sim.last_change = sim.now;
        --sim.active;
        schedule_needs_arises(me, expdis(lambda[me]));
        cs_in_use = false;
        ++CR_log[me];
        token_sent = false;
        j = (me + 1)%N;
        while (!token_sent && (j-me)%N) {
            if (CR_log[j] < CR_request[me][j]) {
                schedule_receive_token(j, curr_token_age, TRANS);
                token_sent = true;
                token_present[me] = false;
            }
            j = (j+1) % N;
        }
    }
}

static void recover(int me)
{
    int i;

    schedule_break_down(me, BREAKTIME);

    site_up[me] = true;
    if (CR_request[me][me] > CR_log[me]) {
        for (i = 0; i < N; i++)
            if (i != me)
                schedule_receive_request(i, me, CR_request[me][me],
                 TRANS);
        schedule_time_out(me, _tmax);
    }
}
static void break_down(int me)
{
    ++cs_log[me];
    site_up[me] = false;
    schedule_recovery(me, DOWNTIME);
    if (token_present[me]) {
        token_present[me] = false;
        cs_in_use = false;
    }
}

static void end_sim(void)
{
    int i;
    double wait;
    event *dummy;

    sim.L += (sim.now - sim.last_change)*sim.active;
    for (i = 0; i < N; i++)
    {
        if (CR.request[i][i] > CR.log[i]) {
            wait = sim.now - sim.start_wait[i];
            sim.max_wait = max(wait, sim.max_wait);
            sim.total_wait += wait;
        }
    }
    closeHp(&eventHp);
}

void nishio(void)
{
    event *temp;
    boolean done = false;

    init_nishio();
    
    do {
        deHp(&eventHp, (void **)(&temp));
        sim.now = temp->time;

        switch (temp->code) {
            case NeedArises:
                need_arises(temp->nearsite);
                break;
            case ReceiveRequest:
                receive_request(temp->nearsite, temp->farsite,


temp->seqnum1);  
    break;
  case ReceiveAck:
    receive_ack(temp->nearsite, temp->farsite, temp->seqnum1,
               temp->seqnum2);
    break;
  case ReceiveNack:
    receive_nack(temp->nearsite, temp->farsite, temp->seqnum1);
    break;
  case TimeOut:
    time_out(temp->nearsite);
    break;
  case ReceiveMissing:
    receive_missing(temp->nearsite, temp->farsite,
                     temp->seqnum1);
    break;
  case ReceiveToken:
    receive_token(temp->nearsite, temp->seqnum1);
    break;
  case ExitCS:
    exit_cs(temp->nearsite, temp->seqnum1);
    break;
  case Recovery:
    recover(temp->nearsite);
    break;
  case BreakDown:
    break_down(temp->nearsite);
    break;
  case EndSim:
    end_sim();
    done = true;
    break;
  default:
    printf("Strange number in nishio\n");
    exit(1);
    break;
  }
  free(temp);
} while (!done);

/************************************************************
 agrab.c
algorithm AA
Agrawal and El Abbadi, "An efficient and fault
tolerant solution for distributed mutual exclusion, _ACM
********************************************************************/
#include "mutex.h"

#define MAX_MSG_TIME (3*AVG_MSG_TIME)
#define Tresp (2*MAX_MSG_TIME)
#define Tc_s (N*MAX_MSG_TIME)
#define Tack (2*MAX_MSG_TIME)

#define msgtime(site1,site2) (((site1)==(site2))?0.0:(TRANS))

eenum clocks {resp, cs, ack};
eenum replies {none, grant, fail, inquire};
typedef struct reply_record {
    int code;
    long int timestamp;
    boolean in_req_set;
    boolean needs_release;
} reply_record;

/* global globals */
extern int N;
extern double lambda[MAXNET];
extern double breakt, downt;
extern double (*msg_time)(void);
extern heap eventHp;
extern sim st sim;

/* local globals */
static reply_record request_set[MAXNET][MAXNET];
static request *next[MAXNET], *CSSTAT[MAXNET];
static heap waitQ[MAXNET];
static event *timer[MAXNET][3];
static boolean site_up[MAXNET], in_cs[MAXNET];
static long int tmstmp[MAXNET], time_curr_req[MAXNET], cs_log[MAXNET];
static boolean requesting_cs[MAXNET];

static void schedule_end_sim(double time)
{
    event *temp;
temp = alloc_event(EndSim, 0, time);
enHp(&eventHp, temp);

static void schedule_need_arises(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(NeedArises, nearsite, sim.now+time);
    enHp(&eventHp, temp);
}

static void schedule_exit_cs(int nearsite, long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ExitCS, nearsite, sim.now+time);
    temp->seqnum1 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_request(int nearsite, int farsite,
                                      long int timestamp,
                                      long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveRequest, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_inquire(int nearsite, int farsite,
                                      long int timestamp,
                                      long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveInquire, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
enHp(&eventHp, temp);
}

static void schedule_receive_fail(int nearsite, int farsite,
long int timestamp,
long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveFail, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_release(int nearsite, int farsite,
long int timestamp,
long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveRelease, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_yield(int nearsite, int farsite,
long int timestamp,
long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveYield, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_ack(int nearsite, int farsite,
long int timestamp,
long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveAck, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_receive_grant(int nearsite, int farsite,
    long int timestamp,
    long int cslog, double time)
{
    event *temp;

    temp = alloc_event(ReceiveGrant, nearsite, sim.now+time);
    temp->farsite = farsite;
    temp->seqnum1 = timestamp;
    temp->seqnum2 = cslog;
    enHp(&eventHp, temp);
}

static void schedule_recovery(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(Recovery, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void schedule_break_down(int nearsite, double time)
{
    event *temp;

    temp = alloc_event(BreakDown, nearsite, sim.now + time);
    enHp(&eventHp, temp);
}

static void cancel_timer(int me, int clock)
{
    if (!delete_evt(&eventHp, (void *) (timer[me][clock]))) {
        printf("ERROR -- timers are mixed up at %f.\n", sim.now);
    }
}
exit(1);
}
free(timer[me][clock]);
timer[me][clock] = NULL;
}

static void schedule_time_out(int nearsite, int clock, double time)
{
    if (!timer[nearsite][clock]) {
        timer[nearsite][clock] =
            alloc_event(TimeOut, nearsite, sim.now + time);
        timer[nearsite][clock]->seqnum1 = (long int)clock;
        enHp(&eventHp, timer[nearsite][clock]);
    } else {
        timer[nearsite][clock]->time = sim.now + time;
        if (!reposition(&eventHp, (void *) (timer[nearsite][clock]))) {
            printf("ERROR - timers are mixed up at %f!\n", sim.now);
            exit(1);
        }
    }
}

static boolean missing_leaf(int me)
{
    int i;
    boolean answer;

    answer = false;
    for (i = N-1; i >= N/2; i--) 
        if (request_set[me][i].in_req_set & request_set[me][i].code == none)
            answer = true;

    return answer;
}

static boolean have Resp(int me)
{
    boolean answer;
    int i;

    answer = true;
    for (i = 0; i < N; i++)
        if (request_set[me][i].in_req_set & request_set[me][i].code == none)
answer = false;

return answer;
}

static void choose_req_set(int me, int root)
{
    int permsite;

    permsite = root;
    while (permsite < N) {
        request_set[me][permsite].in_req_set = true;
        request_set[me][permsite].needs_release = true;
        schedule_receive_request(permsite, me, tmstamp[me],
                                   cs_log[me], msgtime(me, permsite));
        permsite = 2*permsite + (int)(1.5 + ranf());
    }
}

static void start_req_process(int me)
{
    int i;

    for (i = 0; i < N; i++) {
        request_set[me][i].in_req_set = false;
        request_set[me][i].needs_release = false;
        request_set[me][i].code = none;
    }
    choose_req_set(me, 0);
    schedule_time_out(me, resp, Tresp);
}

static void init_agrab(void)
{
    int i, j;

    init_sim();
    createHp(&eventHp, N*(N+1), comp_evt, out_evt);

    for (i = 0; i < N; i++) {
        site_up[i] = true;
        createHp(&waitQ[i], 2*N, comp_reqs, out_req);  
        next[i] = NULL;
        CSSTAT[i] = NULL;
for (j = 0; j < 3; j++)
    timer[i][j] = NULL;
in_cs[i] = false;
tmstmp[i] = 0;
for (j = 0; j < N; j++)
    request_set[i][j].timestamp = 0;
    cs_log[i] = 0;
    requesting_cs[i] = false;
schedule_need_arises(i, expdis(lambda[i]));
schedule_break_down(i, BREAKTIME);
}
schedule_end_sim(ENDTIME);
}

static void need_arises(int me)
{
    if (site_up[me]) {
        sim.L += (sim.now - sim.last_change)*sim.active;
        sim.last_change = sim.now;
        ++sim.active;
        if (sim.maxact < sim.active)
            sim.maxact = sim.active;
        sim.start_wait[me] = sim.now;

        requesting_cs[me] = true;
        ++tmstmp[me];
        start_req_process(me);
    }
    else
        schedule_need_arises(me, expdis(lambda[me]));
}

static void receive_request(int me, int sender,
                                 long int timestamp, long int cslog)
{
    request *req;

    if (me != sender)
        ++sim.messages;

    if (site_up[me]) {
        tmstmp[me] = max(tmstmp[me], timestamp);

        req = (request *)mymalloc(sizeof(request));

        req->sender = sender;
        req->timestamp = timestamp;
        req->cslog = cslog;

        request_set[me][sender].request = req;
        request_set[me][sender].timestamp = timestamp;
        request_set[me][sender].cslog = cslog;

        sim.sim[me].request = req;
        sim.sim[me].timestamp = timestamp;
        sim.sim[me].cslog = cslog;

        schedule_send_request(sim.sim[me].request, sim.sim[me].timestamp, sim.sim[me].cslog);

    }
req->site = sender;
req->timestamp = timestamp;
req->seqnum = cslog;
enHp(&waitQ[me], req);

if (CSSTAT[me])
    if (comp_reqs(CSSTAT[me], req) < 0)
        schedule_receive_fail(sender, me, ++tmstmp[me], cslog,
                              msgtime(sender, me));
    else if (!next[me]) {
        schedule_receive_inquire(CSSTAT[me]->site, me,
                                  ++tmstmp[me], CSSTAT[me]->seqnum,
                                  msgtime(me, CSSTAT[me]->site));
        next[me] = req;
    }
else if (comp_reqs(req, next[me]) < 0) {
    schedule_receive_fail(next[me]->site, me,
                          ++tmstmp[me], next[me]->seqnum,
                          msgtime(me, next[me]->site));
    next[me] = req;
}
else
    schedule_receive_fail(sender, me, ++tmstmp[me], cslog,
                          msgtime(me, sender));
else { /* CSSTAT is free */
    deHp(&waitQ[me], (void **)(&req));
    schedule_receive_grant(req->site, me, ++tmstmp[me], req->seqnum,
                           msgtime(req->site, me));
    schedule_time_out(me, ack, Tack);
    CSSTAT[me] = req;
}
}

static void receive_grant(int me, int sender, long int timestamp,
                           long int cslog)
{
    double wait;
    int i;
    boolean allrec;

    if (me != sender)
        ++sim.messages;
if (site_up[me]) {
    tmstmp[me] = max(tmstmp[me], timestamp);

    if (cslog == cs_log[me] && request_set[me][sender].in_req_set &&
        timestamp > request_set[me][sender].timestamp) {
        schedule_receive_ack(sender, me, ++tmstmp[me], cslog,
            msgtime(me, sender));
        request_set[me][sender].code = grant;
        request_set[me][sender].timestamp = timestamp;
    if (have_resp(me)) {
        if (timer[me][resp])
            cancel_timer(me, resp);
        allrec = true;
        for (i = 0; i < N; i++)
            if (request_set[me][i].in_req_set &&
                request_set[me][i].code != grant &&
                request_set[me][i].code != inquire)
                allrec = false;

        if (allrec) {
            if (timer[me][cs])
                cancel_timer(me, cs);
            wait = sim.now - sim.start_wait[me];
            sim.total_wait += wait;
            if (wait > sim.max_wait)
                sim.start_big_wait = sim.start_wait[me];
            sim.max_wait = max(wait, sim.max_wait);

            schedule_exit_cs(me, cslog, CSTIME);
        }
        else
            schedule_time_out(me, cs, Tc_s);
    }
}
else if (cslog != cs_log[me] ||
    !request_set[me][sender].in_req_set)
    schedule_receive_release(sender, me, ++tmstmp[me], cslog,
        msgtime(me, sender));
}

static void receive_ack(int me, int sender, long int timestamp,
    long int cslog)
if (me != sender)
  ++sim.messages;

if (site_up[me]) {
  tmstmp[me] = max(tmstmp[me], timestamp);
  if (CSSTAT[me] &&
      sender == CSSTAT[me]->site &&
      cslog == CSSTAT[me]->seqnum & & timer[me][ack])
    cancel_timer(me, ack);
}

static void receive_yield(int me, int sender, long int timestamp,
                           long int cslog)
{
    if (me != sender)
        ++sim.messages;

    if (site_up[me]) {
        tmstmp[me] = max(tmstmp[me], timestamp);

        if (CSSTAT[me] && sender == CSSTAT[me]->site) {
            if (cslog != CSSTAT[me]->seqnum) {
                printf("HEY!! wrong yield cslog at %f\n", sim.now);
                exit(1);
            }
            enHp(&waitQ[me], CSSTAT[me]);
            delp(&waitQ[me], (void **)(&CSSTAT[me]));
            schedule_receive_grant(CSSTAT[me]->site, me, ++tmstmp[me],
                                    CSSTAT[me]->seqnum,
                                    msgtime(me, CSSTAT[me]->site));
            schedule_time_out(me, ack, Tack);
            next[me] = NULL;
        }
    }
}

static void receive_release(int me, int sender,
                             long int timestamp, long int cslog)
{
    int i;
    boolean found;
    request *req;
if (me != sender)  
  ++sim.messages;

if (site_up[me]) {
  tmstmp[me] = max(tmstmp[me], timestamp);

  if (CSSTAT[me] &&
      sender == CSSTAT[me]->site &&
      cslog == CSSTAT[me]->seqnum) {
    if (timer[me][ack])
      cancel_timer(me, ack);
    free(CSSTAT[me]);
    CSSTAT[me] = NULL;
    next[me] = NULL;
    if (!emptyHp(&waitQ[me])) {
      deHp(&waitQ[me], (void **)(&CSSTAT[me]));
      schedule_receive_grant(CSSTAT[me]->site, me, ++tmstmp[me],
                              CSSTAT[me]->seqnum,
                              msgtime(me, CSSTAT[me]->site));
      schedule_time_out(me, ack, Tack);
    }
  }
  else {  /* kludge extraordinaire */
    i = 1;
    found = false;
    while (i <= waitQ[me].top && !found) {
      req = (request *)(waitQ[me].list[i]);
      if (req->site == sender && req->seqnum == cslog)
        found = true;
      else
        ++i;
    }
    if (found)
      delete_evt(&waitQ[me], (void *)req);
  }
}

static void receive_fail(int me, int sender, long int timestamp,
                          long int cslog)
{
  int i;
  long int currstamp;
if (me != sender)
  ++sim.messages;

if (site_up[me]) {
  tmstmp[me] = max(tmstmp[me], timestamp);
  if (cslog == cs_log[me] &&
      timestamp > request_set[me][sender].timestamp) {
    currstamp = ++tmstmp[me];
    request_set[me][sender].code = fail;
    request_set[me][sender].timestamp = timestamp;
    for (i = 0; i < N; i++)
      if (request_set[me][i].in_req_set &&
          request_set[me][i].code == inquire) {
        schedule_receive_yield(i, me, currstamp, cslog,
                                msgtime(me, i));
        request_set[me][i].code = fail;
      }
  }
  if (have_resp(me)) {
    if (timer[me][resp])
      cancel_timer(me, resp);
    schedule_time_out(me, cs, Tc_s);
  }
}
else if (cslog != cs_log[me] ||
         !request_set[me][sender].in_req_set)
  schedule_receive_release(sender, me, ++tmstmp[me], cslog,
                           msgtime(me, sender));

}

static void receive_inquire(int me, int sender, long int timestamp,
                             long int cslog)
{
  int i;
  boolean received_fail;

  if (me != sender)
    ++sim.messages;

  if (site_up[me]) {
    tmstmp[me] = max(tmstmp[me], timestamp);
    if (cslog == cs_log[me] &&

if (timestamp > request_set[me][sender].timestamp) {
    request_set[me][sender].timestamp = timestamp;
    if (!request_set[me][sender].in_req_set) {
        schedule_receive_release(sender, me, ++tmstmp[me], cslog,
                                 msgtime(me, sender));
        request_set[me][sender].needs_release = false;
    } else {
        received_fail = false;
        for (i = 0; i < N; i++)
            if (request_set[me][i].in_req_set &&
                request_set[me][i].code == fail)
                received_fail = true;
        if (received_FAIL) {
            request_set[me][sender].code = fail;
            schedule_receive_yield(sender, me, ++tmstmp[me],
                                    cslog, msgtime(me, sender));
        } else
            request_set[me][sender].code = inquire;
    }
}

static void exit_cs(int me, long int cslog)
{
    int i;
    long int currstamp;

    if (cslog == cs_log[me]) {
        ++sim.cs Uses;
        sim.L += (sim.now - sim.last_change)*sim.active;
        sim.last_change = sim.now;
        --sim.active;

        schedule_need_arises(me, expdis(lamba[me]));
        ++cs_log[me];
        in_cs[me] = false;
        requesting_cs[me] = false;
        currstamp = ++tmstmp[me];
        for (i = 0; i < N; i++)
            if (request_set[me][i].needs_release)
                schedule_receive_release(i, me, currstamp,
```java

cslog, msertime(i, me));
}
}

static void time_out(int me, int clock)
{
    int i;
    long int currstamp;

    timer[me][clock] = NULL;
    if (site_up[me]) {
        currstamp = ++tmstmp[me];
        if (clock == resp)
            if (missing_leaf(me)) {
                for (i = 0; i < N; i++)
                    if (request_set[me][i].needs_release)
                        schedule_receive_release(i, me, currstamp,
                                                  cs_log[me],
                                                  msctime(i, me));
                ++cs_log[me];
                start_req_process(me);
            }
        else if (clock == cs) {
            for (i = 0; i < N; i++)
                if (request_set[me][i].needs_release)
                    schedule_receive_release(i, me, currstamp, cs_log[me],
                                                msctime(me, i));
                ++cs_log[me];
                start_req_process(me);
        }
        else if (clock == Tresp) {
            for (i = 0; i >= 0; i--)
                if (request_set[me][i].in_req_set &&
                    request_set[me][i].code == none) {
                    request_set[me][i].in_req_set = false;
                    if (!request_set[me][2*i+1].needs_release)
                        choose_req_set(me, 2*i+1);
                    else if (!request_set[me][2*i+2].needs_release)
                        choose_req_set(me, 2*i+2);
            }
            schedule_time_out(me, resp, Tresp);
        }
        else if (clock == cs) {
            for (i = 0; i < N; i++)
                if (request_set[me][i].needs_release)
                    schedule_receive_release(i, me, currstamp, cs_log[me],
                                                msctime(me, i));
                ++cs_log[me];
                start_req_process(me);
            }
    }
    else /* clock == ack */ {
        schedule_receive_grant(CSSTAT[me]->site, me, currstamp,
```
CSSTAT[me]->seqnum, 
msgtime(me, CSSTAT[me]->site));
    schedule_time_out(me, ack, Tack);
}
}

static void break_down(int me)
{
    site_up[me] = false;
    schedule_recovery(me, DOWNTIME);
    in_cs[me] = false;
}

static void recover(int me)
{
    schedule_break_down(me, BREAKTIME);

    site_up[me] = true;
    if (requesting_cs[me])
        if (haveresp(me))
            schedule_time_out(me, cs, Tc_s);
        else
            schedule_time_out(me, resp, Tresp);
    if (CSSTAT[me]) {
        schedule_receive_grant(CSSTAT[me]->site, me, ++tmstmp[me],
                                CSSTAT[me]->seqnum,
                                msgtime(me, CSSTAT[me]->site));
        schedule_time_out(me, ack, Tack);
    }
}

static void end_sim(void)
{
    int i;
    double wait;

    sim.L += (sim.now - sim.last_change)*sim.active;
    for (i = 0; i < N; i++) {
        free(CSSTAT[i]);
        if (requesting_cs[i]) {
            wait = sim.now - sim.start_wait[i];
            sim.max_wait = max(wait, sim.max_wait);
            sim.total_wait += wait;
        }
    }
}
void agrab(void)
{
    event *temp;
    boolean done = false;

    init_agrab();

    do {
        deHp(&eventHp, (void **)(&temp));
        sim.now = temp->time;

        switch (temp->code) {
        case NeedArises:
            need_arises(temp->nearsite);
            break;
        case ReceiveRequest:
            receive_request(temp->nearsite, temp->farsite,
                             temp->seqnum1, temp->seqnum2);
            break;
        case ReceiveInquire:
            receive_inquire(temp->nearsite, temp->farsite,
                            temp->seqnum1, temp->seqnum2);
            break;
        case ReceiveFail:
            receive_fail(temp->nearsite, temp->farsite,
                         temp->seqnum1, temp->seqnum2);
            break;
        case ReceiveRelease:
            receive_release(temp->nearsite, temp->farsite,
                            temp->seqnum1, temp->seqnum2);
            break;
        case ReceiveYield:
            receive_yield(temp->nearsite, temp->farsite,
                          temp->seqnum1, temp->seqnum2);
            break;
        case ReceiveGrant:
            receive_grant(temp->nearsite, temp->farsite,
                           temp->seqnum1, temp->seqnum2);
        }
    } while (!done);
    closeHp(eventHp);
}
break;
case ReceiveAck:
    receive_ack(temp->nearsite, temp->farsite,
                temp->seqnum1, temp->seqnum2);
    break;
case ExitCS:
    exit_cs(temp->nearsite, temp->seqnum1);
    break;
case EndSim:
    end_sim();
    done = true;
    break;
case TimeOut:
    time_out(temp->nearsite, (int)temp->seqnum1);
    break;
case Recovery:
    recover(temp->nearsite);
    break;
case BreakDown:
    break_down(temp->nearsite);
    break;
default:
    printf("Strange number in agrab\n");
    exit(1);
    break;
} 
free(temp); 
} while (!done); 
}
Bibliography


