1

PERFORMANCE OF CONCURRENCY CONTROL ALGORITHMS IN DISTRIBUTED SYSTEMS

by

SAMUEL LYCURGUS GRIER JR, B.S., M.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1989

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	1b. RESTRICTIVE MARKINGS NONE
2a. SECURITY CLASSIFICATION AUTHORITY	3. DISTRIBUTION/AVAILABILITY OF REPORT APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE	
4. PERFORMING ORGANIZATION REPORT NUMBER(S)	5. MONITORING ORGANIZATION REPORT NUMBER(S) AFIT/CI/CIA- 89-076
6a. NAME OF PERFORMING ORGANIZATION AFIT STUDENT AT UNIV OF TEXAS AT AUSTIN 6b. OFFICE SYMBOL *(If applicable)*	7a. NAME OF MONITORING ORGANIZATION AFIT/CIA
6c. ADDRESS *(City, State, and ZIP Code)*	7b. ADDRESS *(City, State, and ZIP Code)* Wright-Patterson AFB OH 45433-6583
8a. NAME OF FUNDING/SPONSORING ORGANIZATION 8b. OFFICE SYMBOL *(If applicable)*	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER
8c. ADDRESS *(City, State, and ZIP Code)*	10. SOURCE OF FUNDING NUMBERS PROGRAM ELEMENT NO. / PROJECT NO. / TASK NO. / WORK UNIT ACCESSION NO.

11. TITLE *(Include Security Classification)* (UNCLASSIFIED)
Performance of Concurrency Control Algorithms in Distributed Systems

12. PERSONAL AUTHOR(S)
Samuel Lycurgus Grier, Jr.

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT *(Year, Month, Day)*	15. PAGE COUNT
THESIS/DISSERTATION	FROM ______ TO ______	1989	177

16. SUPPLEMENTARY NOTATION
APPROVED FOR PUBLIC RELEASE IAW AFR 190-1
ERNEST A. HAYGOOD, 1st Lt, USAF
Executive Officer, Civilian Institution Programs

17. COSATI CODES			18. SUBJECT TERMS *(Continue on reverse if necessary and identify by block number)*
FIELD	GROUP	SUB-GROUP	

19. ABSTRACT *(Continue on reverse if necessary and identify by block number)*

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT [✓ UNCLASSIFIED/UNLIMITED] ☐ SAME AS RPT. ☐ DTIC USERS	21. ABSTRACT SECURITY CLASSIFICATION UNCLASSIFIED
22a. NAME OF RESPONSIBLE INDIVIDUAL ERNEST A. HAYGOOD, 1st Lt, USAF	22b. TELEPHONE *(Include Area Code)* (513) 255-2259 22c. OFFICE SYMBOL AFIT/CI

DD Form 1473, JUN 86 *Previous editions are obsolete.* SECURITY CLASSIFICATION OF THIS PAGE

AFIT/CI "OVERPRINT"

PERFORMANCE OF CONCURRENCY CONTROL ALGORITHMS IN DISTRIBUTED SYSTEMS

APPROVED BY

SUPERVISORY COMMITTEE:

Jeffrey A. Brumfield

[illegible]

[illegible]

[illegible]

[illegible]

[illegible]

To Veronica, Benjamin and Emily — for all the pain and joy we have experienced together as a family these last 3 years I have been in school.

Acknowledgments

I want to acknowledge Avi Silberschatz for his crucial role in guiding me through the graduate school maze. His wisdom and insights were invaluable. Jeff Brumfield gave me the gift of his time as I struggled to put together thoughts and ideas. Avi's wit and Jeff's good nature provided the uplifting encouragement I needed when I wondered if graduation would ever come to pass. To the rest of my committee - J.C. Browne, Gael Buckley, Vipin Kumar and Herb Schwetman — thank you.

I also want to thank Hank Korth whose reaction to my need to graduate in 3 years was, "Miracles do happen." As my initial advisor, he outlined the milestones and the time frames I had to meet in order to graduate within the constraints set by the Air Force.

To Gloria Ramirez and Karen Shaffer, the C.S. Department couldn't do it without you! To Sara Strandtman, thanks for all of your help.

To Dad — thanks for all of your support and encouragement.

To Jerry Beverly — thanks for your many words of wisdom.

To Archie, Paul, Chris, Dirk, Sarah, Vicki, Bill The Lasher, Mark, Ed, the DSDG, Anish, Sumit, David, Nick, Martha and my other fellow grad students, I leave my office (ha! ha!), my box in the basement of Taylor Hall (you can keep the spider) and my portion of the low self-esteem shared by all graduate students. Thank you for your friendship.

One of the dangers that faces many of us trying to complete such a rigorous program is losing touch with our families. I thank my wife who consistently reminded me of my responsibilities as a husband and dad; she was convinced I could be both a student and a father, and she was right.

Finally, I want to acknowledge God's faithfulness through this entire process. Through Dick Flaten, the men in Bible Study Fellowship and Bill Hobart, He provided encouragement, humor and wisdom to persevere in the face of adversity. Through their prayers, He brought success out of what sometimes seemed certain disaster. When I floundered, He made me soar on the wings of eagles. When I felt as though I could only walk, He gave me the strength to run. And when I felt as though weariness might overcome me, He gave me the ability to press forward...

Samuel Lycurgus Grier Jr

The University of Texas at Austin
August, 1989

PERFORMANCE OF CONCURRENCY CONTROL ALGORITHMS IN DISTRIBUTED SYSTEMS

Publication No.____________

Samuel Lycurgus Grier Jr, Ph.D.

The University of Texas at Austin, 1989

Supervising Professors: Abraham Silberschatz and Jeffrey A. Brumfield

The performance of concurrency control algorithms is explored in the context of pessimistic distributed mutual exclusion algorithms. A detailed model of the underlying distributed system and its parameters are defined for fail-free systems and systems where processes may fail. New performance metrics are proposed as a means to measure the efficiency of distributed mutual exclusion algorithms. A model of distributed mutual exclusion algorithms and analytical approximations of their performance are developed. Simulation results of five selected distributed mutual exclusion algorithms on fully-connected, star and ring topologies are cataloged. Finally, simulation results of four selected distributed mutual exclusion algorithms on a fully-connected topology where processes may fail are cataloged. Conclusions are drawn as to which algorithm is most appropriate for particular environments.

Table of Contents

Chapter 1

Introduction

The oft-quoted proverb "two heads are better than one" often seems to have its counterpart in computing sciences: "two processors are better than one; and furthermore, the more processors, the better." One particular multiprocessing environment is that of asynchronous distributed systems. In asynchronous distributed systems, nodes are connected to one another by some communications medium, and nodes do not share memory or have a common clock. Processes at different nodes communicate with one another by exchanging messages [MOO, PS].

Many algorithms can be designed for asynchronous distributed systems that will result in the following improvements over similar algorithms written for single processors [MS]:

1. faster execution due to parallelism
2. improved resiliency to failures
3. absence of single site bottlenecks

As a result of these potential enhancements, much research is concentrating on the area of distributed systems.

1.1 The Problem

An algorithm that distributes its computation among asynchronous processes at multiple sites is a distributed algorithm. Often times, the distributed computations contend for a common resource. Concurrency control algorithms are required to efficiently manage the contention. This necessity of controlling concurrency in distributed systems gives rise to a "layered" system, where there is an underlying computation and a separate algorithm to synchronize the underlying computation [CM87a]. We will not concern ourselves with the underlying computation. We will assume that the process executing the underlying computation participates in the concurrency control algorithm as necessary to obtain and release resources.

One important class of concurrency control algorithms, and the one that is the focus of this dissertation, is that of distributed mutual exclusion. In distributed mutual exclusion algorithms, a process requests entry to a critical section. The process waits for entry to the critical section in a mutual exclusion queue and then executes the critical section when it is acquired. Upon completion of the critical section, the process releases it back to the algorithm.

Distributed mutual exclusion algorithms can be characterized as centralized. fully distributed, or a hybrid of these two paradigms.

A centralized algorithm designates a single process as a central controller (sometimes referred to as an arbitrator). This controller coordinates the activity of every other process in the system. A centralized algorithm has the following characteristics [MOO]:

1. There exists a single central controller.

2. The controller maintains all necessary information to enforce the desired algorithmic behavior.[1]

3. The central controller makes decisions based on global state information.

4. The central controller bears responsibility for ensuring that the actions taken by other processes in the system result in the desired algorithmic behavior.

A fully distributed algorithm requires every process to do an equal amount of coordination to enforce the desired behavior. Specifically, a fully distributed algorithm has the following characteristics [MOO]:

1. Processes possess an equal amount of partial state information.[2]

2. Processes make decisions solely on partial state information.

3. Processes bear equal responsibility for enforcing the desired algorithmic behavior.

4. Processes expend an equal amount of effort to enforce the desired algorithmic behavior.

A hybrid algorithm may lie anywhere in the spectrum between centralized and fully distributed. One particular hybrid algorithm is the mutual exclusion algorithm of Maekawa [Mae]:

1. The controller is replicated some number of times in the system.

[1] Redundant information may be maintained by other processes.

[2] In mutual exclusion, for example, partial state information implies that a process does not have complete knowledge of the priority queue.

2. Each controller is responsible for the behavior of some subset of processes in the system.

3. Conflicts among processes in different subsets are resolved by the responsible controller.

Another important area of investigation within distributed systems is that of fault tolerance. In the presence of failures, concurrency control algorithms must allow processes to detect failure, compensate for it, and continue computation. If a process fails and then recovers, the algorithm must allow the process to reacquire the state lost at failure and rejoin the algorithm in a state consistent with the execution of the algorithm. Protocols for failure detection and recovery vary according to whether a system employs a centralized, fully distributed, or hybrid algorithm.

A centralized implementation must employ the following failure protocols when processes fail:

- **failure detection**

    1. Detection of controller failure.
    2. Detection of process failure.

- **failure compensation**

    1. When the controller fails, a new controller must be elected and the explicit mutual exclusion queue reconstructed.
    2. When an arbitrary process fails, the controller must update the mutual exclusion queue if necessary and ensure the critical section does not remain occupied by a failed process.

- **failure recovery**

  1. When a process rejoins the algorithm, it must determine which process is currently serving as the controller.

Fully distributed implementations must employ the following protocols to continue execution in the presence of process failures:

- **failure detection**

  1. Detection of process failure.

- **failure compensation**

  1. The process detecting the failure of another process must update its local state and notify the rest of the system of the failure.

- **failure recovery**

  1. When a process rejoins the algorithm, it must obtain a list of active processes in the system, notify other active processes that it has rejoined the algorithm, and obtain a timestamp consistent with the execution of the algorithm.

A hybrid implementation will employ protocols tailored to the peculiar characteristics of the algorithm. In the *Maekawa* algorithm, for example, there is logical replacement of failed controllers rather than the use of an election protocol [Mae]. In the *Schneider* algorithm, global state must be reacquired by a recovering process [Sch].

What metrics are available to quantify faster execution, improved failure resiliency, and absence of bottlenecks in distributed systems? The following criteria have been proposed as a means to assess the performance of a distributed algorithm with no failures [Sil84]:

- The amount of message traffic generated to implement the algorithm.
- The amount of local processing required by a process to carry out its designated function.
- The amount of local data required by a process to carry out its designated function.

When the notion of process failure is added to a distributed system, the following additional criteria are proposed [Sil84]:

- The amount of redundant data that must be maintained in order for the system to recover in the event of failure.
- The efficiency of an algorithm after it recovers from failure.
- The ability of an algorithm to continue computation in the presence of process failure.
- The similarity of an algorithm's characteristics after failure to its characteristics prior to failure.

These criteria are primarily a subjective basis of quantification. What does it mean to say the "efficiency of an algorithm after it recovers from failure"? Without a well-defined basis for describing efficiency prior to failure, there can be no basis for describing efficiency after failure. It appears that new, well-defined metrics are necessary to properly evaluate the performance of distributed algorithms.

1.2 The Mutual Exclusion Problem

Many implementations of distributed mutual exclusion algorithms have been proposed. It is widely held that fully distributed implementations of mutual exclusion are in some way more efficient than their centralized counterparts [Bag, CM87a, RA]. Vulnerability of the central controller and the belief that the central controller becomes a bottleneck are the primary reasons for this assertion.

Mutual exclusion is a special case of distributed semaphores. In both mutual exclusion and distributed semaphore algorithms, some process is chosen from among the other processes; in the case of distributed semaphores, more than one process may be chosen. Since one or more processes can be granted special status in the algorithm for distributed semaphores, an algorithm for distributed semaphores is a more general mechanism than necessary for mutual exclusion. Regardless, how one chooses these processes and how these processes then discover that they are chosen is a dilemma shared with mutual exclusion algorithms.

Leader election is a special case of mutual exclusion. In both leader election and mutual exclusion algorithms, some single process is chosen from among the other processes in the system. This process is then granted special status: in mutual exclusion, the chosen process enters the critical section; in leader election, the chosen process becomes a central controller. How one chooses the process and how the process discovers its special status are basic to both algorithms.

Hence, if we can make observations about the performance of mutual exclusion algorithms, they may help determine how leader election and distributed semaphore algorithms perform in their various implementations. Finally, any new performance metrics that we discover for mutual exclusion algorithms may be useful for measuring the performance of all distributed systems that use shared resources.

1.3 Definition of Mutual Exclusion

The critical section is a shared resource for which processes throughout a distributed system contend. An implementation of mutual exclusion must guarantee every requesting process in the distributed system exclusive access to the critical section within a finite period of time. The following definition of mutual exclusion is used as the basis for discussion throughout this dissertation [CM87a, Lam86b, MOO, PS, Ray].

Each process in a distributed system contains a noncritical section statement and a critical section statement. These statements are executed alternately and generate the following sequence of executions for some process i:

$$NCS_i \rightarrow waiting_i \rightarrow CS_i \rightarrow NCS_i \rightarrow waiting_i \rightarrow CS_i \rightarrow \cdots$$

where NCS_i denotes the execution of process i's noncritical section, CS_i denotes the execution of process i's critical section, and $waiting_i$ denotes that process i has submitted a request for the critical section which has not been satisfied.

For any process i, CS_i is finite; that is, process i never halts in the critical section. This assumption also holds in a fail-prone environment in the sense that failure in the critical section is detected using a timeout mechanism.

allowing computation to continue. On the other hand, for any process i, NCS_i is not necessarily finite; that is, process i may halt in its non-critical section.

Any correct solution to the mutual exclusion problem must guarantee the following three properties.[3]

Mutual Exclusion: For any pair of processes i and j, CS_i and CS_j are not concurrent:

$$(\exists \text{ a process } i :: CS_i) \Rightarrow (\neg\exists \text{ a process } j : j \neq i :: CS_j)$$

Progress: At least one of the processes wishing to reach its critical section must be capable of reaching it:

$$(\exists \text{ a process } i :: \mathit{waiting}_i \wedge (\neg\exists \text{ a process } j : j \neq i :: CS_j) \mapsto CS_i)$$

Bounded Waiting: Any process attempting to reach its critical section must reach it in finite time:

$$(\exists \text{ a process } i :: \mathit{waiting}_i \mapsto CS_i)$$

1.4 Goals and Strategies

In this dissertation, we intend to examine the assertion that fully distributed implementations of mutual exclusion are superior to their centralized counterparts. We will characterize the performance of fully distributed and centralized implementations in the context of new performance metrics for distributed systems.

[3]See [CM87a] for an explanation of notation.

This research is also aimed at quantifying the overhead inherent in distributed mutual exclusion algorithms. Looking at the spectrum of algorithms, which ranges from centralized to fully distributed, we propose to rank the efficiency of these algorithms in fail-free and fail-prone environments. In the fail-prone environment, we have limited our research to systems where only processes fail. In both environments, we will look at systems with and without broadcast capability.

Adding the assumption of process failure, we examine different techniques for the reconstruction of control information after failure. It is not our intention to address how computational data is recovered in a fail-prone environment. Control information necessary to effect mutual exclusion consists of the priority queue and a list of active processes; the priority queue may be either explicit (as in a centralized implementation) or implicit (as in a fully distributed implementation). Control information can be easily reconstructed when lost.

We define a model of computation for the underlying distributed system. A model of sufficient detail for the simulation of distributed mutual exclusion algorithms is currently not available in the literature.

We also define a model of computation for the spectrum of distributed mutual exclusion algorithms. This model allows us to make a reasonable estimate of the efficiency of a particular algorithm on a particular network topology. These estimates are verified using simulation.

The message-based approach to discrete-event simulation of distributed systems is the simulation strategy employed in this research [Bag, BCM, Mis86].

POSSUM[4] is the simulation facility used to collect all simulation results [Bru].

The rest of this dissertation is arranged as follows. In Chapter 2, we define the model of the underlying distributed system which forms the basis for our results. We also define our model for distributed mutual exclusion algorithms and develop our approximation methods for predicting their performance on varying topologies. Chapter 3 argues the need for new metrics, and it also presents the analytical results that are derivable from the new metrics that we define. In Chapter 4, fail-free simulation parameters are defined and results cataloged. Chapter 5 defines fail-prone simulation parameters and catalogs the simulation results. Finally, Chapter 6 provides conclusions based on our results and outlines future research.

[4]POSSUM: Process Oriented Simulation System Using Modula-2, ©1986 by Dr. Jeffrey Brumfield at the University of Texas at Austin.

Chapter 2

Algorithms and Models

In this chapter, we claim that the *Maekawa* algorithm is incorrectly implemented in [MOO], which reflects an incorrect proof of deadlock freedom in [Mae]. We present an example where the algorithm deadlocks and identify the erroneous assertion in the proof. We then provide a correction to the algorithm.

Our next task is to define a model of the underlying distributed system. We present a model with detail sufficient such that one can simulate distributed mutual exclusion algorithms. We define parameters for both fail-free and fail-prone environments. We are unaware of any comparably detailed description in the literature.

We also present a model of mutual exclusion algorithms which utilizes the notion of system state and the application of a function to that state [AB]. We discuss the ramifications of maintaining global state, partial state and null state on the performance of a distributed system. We also look at the consequences of applying a distributed or centralized function to the state. We then rank three distributed mutual exclusion algorithms, in light of these observations, according to their efficiency in a system where message processing time is negligible. This model provides the basis for our analytical results where message processing time and the resulting contention are significant; these results are derived in Chapter 3 for varying topologies.

2.1 Algorithm Description

We provide the following brief algorithmic descriptions to refresh the reader's mind. It is assumed that the reader is familiar with the mutual exclusion algorithms studied in this dissertation.

2.1.1 Buckley and Silberschatz [BS84]

The *BS* algorithm is a straightforward implementation of the centralized mutual exclusion algorithm. A single controller receives requests for mutual exclusion, and orders them in a FIFO priority queue. Upon receiving a release from the process exiting its critical section, the controller grants mutual exclusion to the head of the queue. Priority in the FIFO queue is based on when the request is received at the controller.

The *BS* algorithm detects process failure by having the controller periodically poll the critical section process. Likewise, processes waiting for the critical section periodically poll the controller. In both cases, a timeout mechanism detects failure. What makes the *BS* algorithm unique is its recovery mechanism; the *BS* algorithm effectively detects and recovers from controller failure in a centralized implementation.

2.1.2 Maekawa [Mae]

The *Maekawa* algorithm is a complex hybrid implementation of mutual exclusion. Based on what *Maekawa* calls the "pairwise non-null intersection property," N sets are formed of size K, where $K \approx \sqrt{N}$.[1] Each process

[1]Each subset S of size K intersects with every other subset of size K; or more formally, $\forall i \forall j : 1 \leq i, j \leq N :: S_i \cap S_j \neq \emptyset$.

occurs in K sets, and each process acts as a controller. With the pairwise non-null intersection property, a process which is granted mutual exclusion within its own subset is guaranteed to have mutual exclusion among all processes. A process requesting mutual exclusion sends a request to all processes within its own subset (process number p is in subset p). If a conflict exists with a requesting process from another subset, the conflict is resolved by the process which occurs in both subsets. When all processes in subset p grant mutual exclusion to p, it acquires the critical section. Upon exiting its critical section, $(K - 1)$ releases are sent to the other processes in p. Priority in the mutual exclusion queue may be either of the priorities described for *BS* and *RA* (see Section 2.1.4); a process may receive priority based on when a request is received at the controller for its subset as in *BS*, or it may cause another process to relinquish its lock, which then gives it priority as in the *RA* algorithm.

The *Maekawa* algorithm is not investigated in a fail-prone environment in this dissertation. More on fail-detection and recovery can be found in [Mae].

There is an error in the *Maekawa* algorithm as published in [MOO], which reflects an incorrect proof of deadlock freedom in [Mae]. It is a subtle error that will go undetected until $K > 5$. A hypothetical sequence of events is proposed, which will cause deadlock. Assume that $K = 6$ and that the 31 processes in the system each have an identical current sequence number. Let *CS* be shorthand for the critical section:

1. Process 27 requests the *CS* from its subset $[1, 27, 28, 29, 30, 31]$. It receives a *locked* message from every member of its subset and enters the *CS*.

2. Process 19 requests the *CS* from its subset $[4, 8, 16, 19, 22, 30]$. It receives a *locked* message from every member of its subset except process 30.

Process 30 sends an *inquire* message off to process 27 to see if process 27 will relinquish 30. Process 27 is currently in the *CS*, so it ignores the *inquire* message. Meanwhile, process 19 never receives a *failed* message.

3. Process 2 now requests the *CS* from its subset $[2, 7, 12, 17, 22, 27]$. It receives a *locked* message from every member of its subset except processes 22 and 27. Process 27 will send an *inquire* message to itself, but since it is still in the *CS*, the *inquire* is ignored. Process 22 will send an *inquire* message off to process 19. Since process 19 has never received a *failed* message, it places the inquire from process 22 into its outstanding inquiry queue. Meanwhile, process 2 never receives a *failed* message.

4. Finally, process 13 requests the *CS* from its subset $[3, 9, 13, 17, 26, 30]$. It receives a locked message from every process except processes 17 and 30. Process 17 is already locked by process 2, which has never received a *failed* message. Process 2 remains the locking process and a *failed* message is returned to process 13. At process 30, process 13 finds it locked by process 27. Process 13 will not be sent a *failed* message, because it has priority over process 19, which is currently at the head of the queue. Since an *inquire* message has already been sent, process 13 is merely inserted into process 30's waiting queue. However, since process 13 has priority over process 19, process 13 is placed at the head of the queue, *in front of* process 19.

5. When process 27 finally exits the *CS*, a *release* message is sent to process 30, and a *locked* message is relayed to process 13. The result: process 13 is waiting for process 2 at 17, process 2 is waiting for process 19 at 22, and process 19 is waiting for process 13 at 30. The system deadlocks.

Maekawa's proof that deadlock cannot occur makes the erroneous assertion that a process involved in a cycle, whose *request* timestamp is preceded by both of its waiting neighbors, must have had at least one of its *request* messages arrive later than a *request* message from one of its waiting neighbors. The preceding scenario shows that this assertion may not be true: process 19's *request* messages arrived prior to those of process 2 and process 13, yet the *request* messages of process 19 had later timestamps.

When process 13 is placed at the head of the queue at process 30, where an *inquire* message has already been sent, a *failed* message must be sent to process 19. Process 19 will then be treated as if its request had actually arrived later than that of process 13; this will cause process 19 to relinquish 22 to process 2, and the cycle will be broken.

This error points to the continued difficulty of programming asynchronous distributed systems and supports the argument against operational proofs of algorithms written for distributed systems. Operational proofs tend to be long and unconvincing; and not surprisingly, sometimes wrong.

The algorithm has been changed so that it will not deadlock under these circumstances.

2.1.3 Schneider [Sch]

The *Schneider* algorithm is not a true implementation of mutual exclusion, but is an implementation of distributed semaphores; it is of a more general nature than the others. By initializing the semaphore with a single *V-phase* transition, the *Schneider* algorithm becomes an implementation of mutual exclusion, where every process maintains global state. The *Schneider* algorithm does not meet our definition of a fully distributed algorithm — within a fully distributed algorithm, processes maintain only partial state.

To maintain global state, every process must acknowledge every message sent to it by every other process. When a process seeks mutual exclusion, it sends a P message to every other process in the system. Every process which receives the P message acknowledges it by sending an *ack* message to every other process. The P message is inserted into a local FIFO queue that is ordered according to when a message is sent. When a P message becomes fully acknowledged, and it is at the head of the FIFO queue, the process which sent the P message has authority to enter the critical section, provided a corresponding V message is in the queue.

Upon exiting the critical section, a process sends a V message to every other process. Every process which receives the V message acknowledges it by sending an *ack* message to every other process. The V message does not have to be fully acknowledged for the process at the head of the priority queue to enter the critical section. However, the V message must still be fully acknowledged to ensure that every process has received the message.

Failure in the *Schneider* algorithm can be discovered in two ways. Since every message must be fully acknowledged, failure can be detected when a message is not fully acknowledged within some timeout period. Failure can also be detected when a process in the critical section fails to release it.

Since each process maintains global state, a recovering process must obtain the message queue from a sponsor process. Rather than using the published protocol, a recovering process sends a restart message to every process in the system. Every process in the system will acknowledge the restart message by sending an *ack* message to every other process. The sponsoring process will send the message queue to the restarting process once it receives an acknowledgement from every other process. The restarting process will save all messages sent to it by a particular process once that particular process has acknowledged its restart. When it finally receives the message queue from its sponsor, the restarting process will update the queue and discard duplicate messages that are already in the queue.

2.1.4 Ricart and Agrawala [RA]

The *RA* algorithm is a fully distributed implementation of mutual exclusion. When a process seeks mutual exclusion, it sends a *request* message to all active processes. Any active process, which has sent a *request* message with an earlier timestamp than the requesting process, defers its reply; otherwise, a *reply* message is returned immediately. When a process exits its critical section, a *reply* message is sent to all deferred processes. A process which has received replies from all active processes in the system has authority to enter its critical section. Priority in the implicit mutual exclusion queue is FIFO, based on the time that a process sends its request.

The *RA* failure detection scheme requires each process to use a timer, which is reset whenever a *reply* message is received by a process waiting for the critical section. If the timer expires and a process has not received a *reply* message, it probes all processes that have not yet responded to its request; those processes not acknowledging the probe are considered to have failed.

One difference, between the published *RA* algorithm and the implementation in our simulations, is the restart mechanism after failure has occurred. We are interested in studying rapid recovery from failure; the published protocol is not an aggressive one. Therefore, rather than use some other process as a sponsor, a restarting process probes all other processes in the system. After a timeout period, processes that have acknowledged the probe are inserted into the request-queue of the restarting process; all other processes are assumed to have failed.

2.1.5 Carvalho and Roucairol [CR]

The *CR* algorithm is a variation of the *RA* algorithm; it uses the notion of implicit authorization. This notion implies that a process p, which has received permission from a process q to use the critical section, retains permission from q to use the critical section until a request is received from q. The advantage of this protocol is that the number of request messages that a process must send to obtain permission to enter the critical section can be greatly diminished; in fact, a process, which has retained permission from all other processes in the system, can immediately enter the critical section without sending a single request.

Failure within the *CR* algorithm is handled in an identical fashion as the *RA* algorithm. However, there is the opportunity for greater efficiency in the *CR* algorithm. If an active process has attained permission from a process which subsequently fails, the active process will not have to compensate for the failed process. The implicit authorization protocol allows the active process to assume that the failed process has simply granted it permission to continue using the critical section.

Algorithm	Implementation	ME Determined	State Kept
BS	Centralized	FIFO (receipt of req)	Global*
Maekawa	Hybrid	FIFO (time or receipt of req)	Partial
Schneider	Distributed	FIFO (time of req)	Global
RA	Distributed	FIFO (time of req)	Partial
CR	Distributed	FIFO (time of req)	Partial

*controller only — all other processes know only their own state

Table 2.1: Summary of Distributed Mutual Exclusion Algorithms

2.2 Model of the Underlying Distributed System

The algorithms we have described must be fitted to a model of an underlying distributed system. We will study loosely-coupled networks where message transmission time is significant and messages can be queued at the destination.

2.2.1 System Structure

- The underlying distributed system is modeled by a finite directed graph.
- Each vertex in the graph is comprised of two parts:
    1. a single non-terminating applications process running on a dedicated processor
    2. a communications interface processor (**CIP**)
- Applications processes are uniquely numbered from 1 to N.
- Each directed edge represents a direct, one-way communications channel between a pair of processes.
- Figure 2.1 shows an example of a fully-connected distributed system.

2.2.2 Communications

- There is no shared memory; no process can directly access the memory of another process.
- Processes communicate solely by sending and receiving messages. When a process transmits the same message to multiple destinations, the message is transmitted to other processes in order from lowest unique number to

highest. For example, if process 1 sends a *request* message to all other processes in the system, it first sends the message to process 2, then process 3, and so on until the last copy of the message is sent to process N.

- Each CIP has a buffer in which all messages from incoming channels and all messages from its applications process are queued. This buffer is assumed to be of sufficient capacity to hold all incoming messages.
- A CIP can process only one message at a time. A CIP can send a single message onto a communications path or broadcast the same message on any number of outgoing channels.
- Message processing consists of a CIP removing a message from an incoming communications path and examining its address, or examining a message's address and then placing it onto an outgoing communications path. Processing of a message occurs at the CIP of the sender, receiver, and any intermediate vertices.
- Message processing time is denoted by t_p. t_p is a random variable having a specified distribution.
- Messages transmitted from a given sender to a given receiver are received in the order they are sent.
- Message transmission time between adjacent processes on a channel is denoted by t_{msg}. t_{msg} is a random variable having a specified distribution.
- Messages are never lost or corrupted by the communications network.
- Messages that are part of the mutual exclusion algorithm are assigned a higher priority than those of the underlying computation.

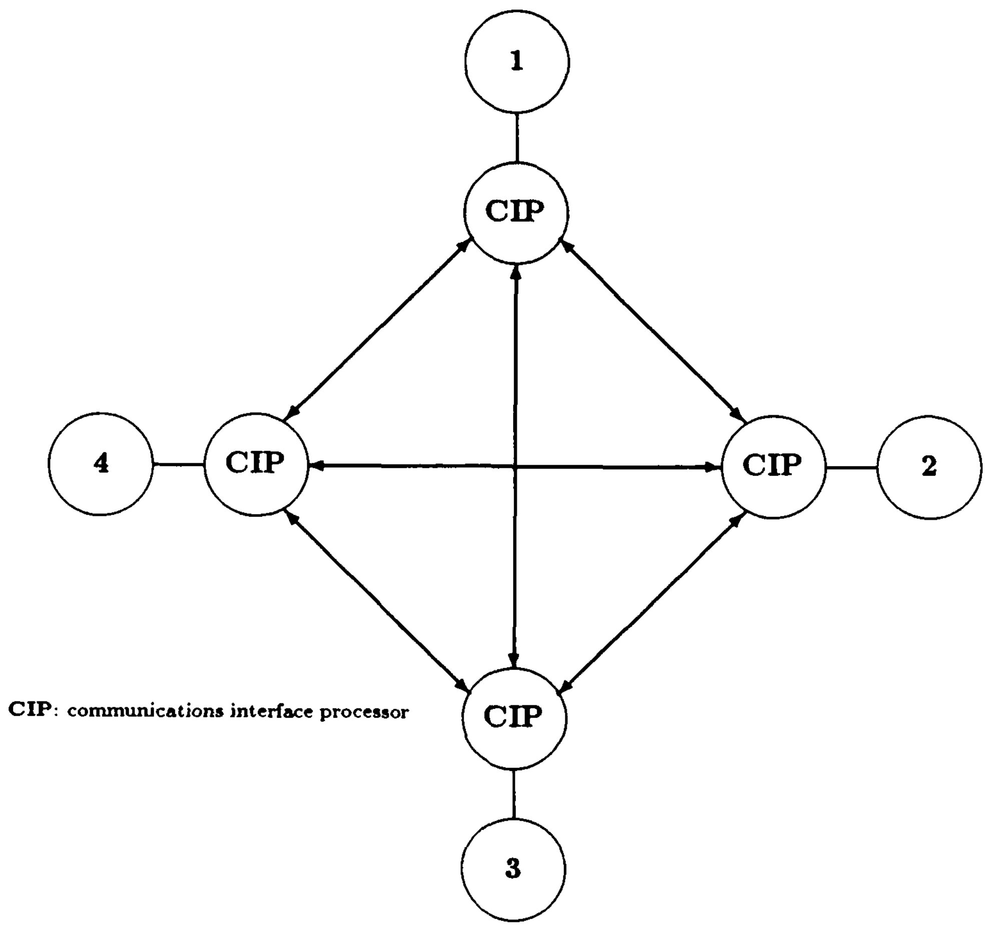


Figure 2.1: An Example of a Fully-connected Distributed System

2.2.3 Process Behavior

- Processes execute asynchronously; there is no common clock.
- All processes share a critical section which can be occupied by at most one process at a time.
- A process exists in one of three discrete states:

1. executing outside the critical section (**NCS**)
2. awaiting entry to the critical section (**waiting**)
3. executing within the critical section (**CS**)

- The initial state of a process is the state in which it is executing outside the critical section (**NCS**).
- Figure 2.2 shows all valid state transitions.
- State transitions are instantaneous.
- The time a process spends outside the critical section is denoted by t_{ncs}. t_{ncs} is a random variable having a specified distribution.
- The time a process spends within the critical section is denoted by t_{cs}. t_{cs} is a random variable having a specified distribution.
- The time a process spends waiting for the critical section is a function of the particular mutual exclusion algorithm.
- A process may send and receive messages in any state.

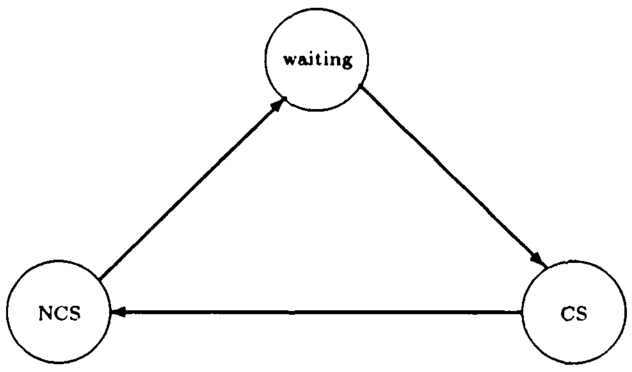


Figure 2.2: Process Behavior in a Fail-free Environment

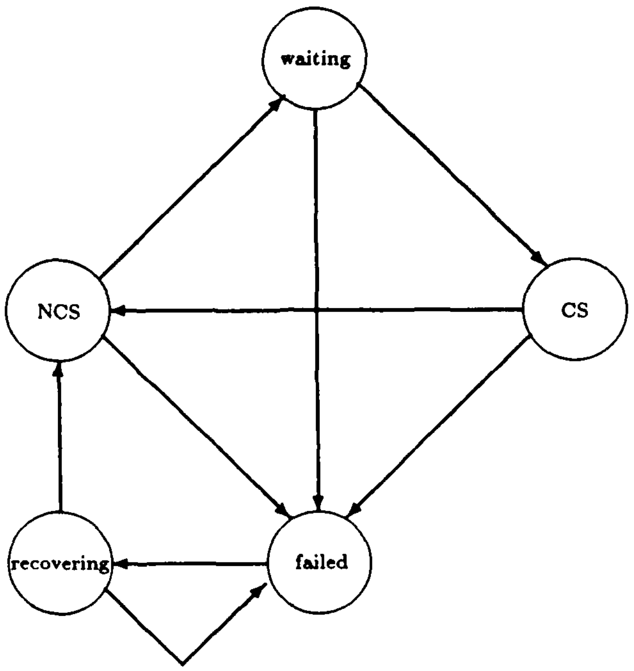


Figure 2.3: Process Behavior in a Fail-prone Environment

2.2.4 Failure and Recovery

- Applications processes may fail; the communications network, which includes the communications interface processors, does not fail.
- Failure of an applications process will not interfere with the communications network.
- Process failure is fail-fast and non-Byzantine [LSP].
- Process failures occur one at a time. All processes are equally likely to fail.
- The interval between process failures is denoted in the system by t_{fail}. t_{fail} is a random variable having a specified distribution.

- Failure requires the introduction of two additional process states:

  1. **failed**
  2. **recovering**

- Figure 2.3 shows the extended state transition diagram. Note that a process may fail in any state except the failed state.

- The time a process remains failed is denoted by t_{dead}. t_{dead} is a random variable having a specified distribution.

- A process can send priority messages to determine if another process has failed, to elect a new controller or to rejoin the algorithm after fail recovery.

- The time from when a process sends a message to another process, until the sending process inquires whether or not the receiver has failed, is denoted by t_{delay}. t_{delay} is a random variable having a specified distribution.

- The time it takes a process to determine whether another process has failed (i.e., the interval from the time a process inquires about a process until failure is verified) is denoted by t_{probe}. t_{probe} is a random variable having a specified distribution.

- Failure of a process invalidates any local information pertaining to the use of the critical section.

- In the failed state, a process cannot send messages, receive messages, or execute instructions. Messages delivered to a failed process are lost.

- During recovery, a process reacquires information pertaining to the management of the critical section.

- The time a process spends recovering is dependent on the particular recovery protocol chosen for the algorithm.
- A process always recovers to the state where it is executing outside the critical section (**NCS**).
- To facilitate failure detection, processes can set local timers that expire after some pre-determined interval.

2.3 Model of Mutual Exclusion Algorithms

In order to study the effects of varying topologies on distributed algorithms, we must have a basis from which to make our computations. We have chosen a state model, because system state is common to all distributed algorithms [AB]. We can view a mutual exclusion algorithm in terms of its state and how this state impacts its execution. Since this view can take place independently of system topology, it is an ideal model for our purposes. We can then overlay the model onto a particular topology to gather our results.

2.3.1 General Structure

Every mutual exclusion algorithm consists of two distinct parts:

1. How the next process is chosen to enter the critical section from among the waiting processes when the critical section becomes available.
2. How the chosen process recognizes that it has authority to enter the critical section.

We can observe that every leader election algorithm consists of two very similar parts [Gar]:

1. How the next process is chosen to be leader.

2. How the system is informed that a new leader has been chosen.

However, leader election algorithms contain two important differences from mutual exclusion algorithms [Gar]:

1. Leader election algorithms must inform every process in the system of the identity of the new leader.

2. Fairness is not a requirement of leader election algorithms; i.e., some processes may never become leader.

We can think of the first part of every mutual exclusion algorithm as choosing a leader; this leader will be the next process to enter the critical section. The second part consists of the newly chosen leader being able to determine that it has been chosen to enter the critical section.

Definition: The local state of some process i is denoted by s_i:

$$s_i \in \{CS_i, NCS_i, waiting_i\}$$

Definition: The state of the distributed system, S, is the set of states consisting of the state of every process in the system:

$$S = (s_1, s_2, s_3, \ldots, s_n) = \cup s_i$$

Definition: A valid system state, S, is a state which has at most one occurrence of CS:

$$(\exists \text{ a process } i :: s_i = CS_i) \Rightarrow (\neg\exists \text{ a process } j : j \neq i :: s_j = CS_j)$$

Any state transition that might take the system into an invalid state is not permitted.

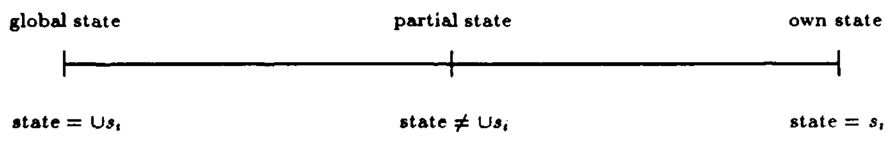


Figure 2.4: Spectrum of Local States

Figure 2.4 reveals that a process that maintains global state has a "snapshot" of the system state [CL].[2] A process that maintains partial state has something less than global state, but more than its own state; its partial state is not necessarily a subset of the global state. For example, a process may not know what state another process is in; however, it may know what state another process isn't in. Finally, a process may be concerned only with its own particular state, oblivious to the state of other processes within the system.

If every process in a distributed system maintains global state, then overhead is accrued due to the large number of messages that must be propagated throughout the system to maintain global state at each process. If every process maintains only its own state, then there is negligible overhead accrued; messages are only exchanged between a controller with global state and the active processes of the system seeking access to that global information. A process with global state has no need to consult a controller; in contrast, a process with no state will accrue overhead in the form of the message transmission time necessary to receive state from another source. An algorithm that

[2]This snapshot does not include the state of the channels.

falls somewhere between the two ends of the local state spectrum may attempt to decrease the number of messages that must be propagated to update local state, decrease message transmission time necessary to obtain state from another source, or decrease both.

In mutual exclusion, we can think of applying a function f to the system state, which chooses the next process to enter the critical section. f is applied to the state S of the computation, returning a decision d, which is the next process to enter the critical section:

$$f(S) \rightarrow d$$

f may be a distributed function, in which case all processes in the system apply f to their local states. If each process maintains global state, then f will tell every process which one has been chosen to enter the critical section. If each process maintains something less than global state, then f may tell only one process that it has been chosen to enter the critical section; f applied to the local states of the other processes will only tell them that they have not been chosen. If f is a centralized function, then f is applied only to the local state of the controller, which has global knowledge of the system. In this case, the controller learns which process is chosen to enter the critical section, and this knowledge is then relayed to the chosen process.

Measuring the efficiency of a mutual exclusion algorithm is dependent on:

1. How quickly messages can be propagated throughout the distributed system so that f can be applied to the updated state.

2. How quickly the decision reached by f can be relayed, if necessary, to the chosen process.

We have already noted that the speed with which a process updates local state depends on the amount of state maintained by each process in the system. When f is a distributed function, there is no need to relay the decision once f has been applied; every process derives a result. However, when f is a centralized function, the decision d must be relayed to the chosen process. Only upon receipt of d can the chosen process exit the *waiting* state and enter its critical section.

2.3.2 Application of the Model to Mutual Exclusion Algorithms

How efficiently messages can be propagated throughout the distributed system, so that local state is updated and f is applied, and the decision d then relayed to the chosen process depends on two properties of the distributed system:

- the state maintained by each process
- the topology of the system network

The two ends of our performance spectrum are centralized and fully distributed. A centralized implementation maintains no state other than its own. Each process is required to coordinate with a central controller, which maintains global knowledge of the system state, to determine if it has authority to enter the critical section. A process with a distributed implementation may maintain global or partial state; in either case, a process maintains sufficient local state to determine whether or not it has authority to enter the critical section. We find that if we can predict the performance of these two ends of our spectrum on a particular topology, then we can use the information gained from their results to predict the performance of a hybrid implementation as well.

For example, on a fully-connected topology with equivalent t_{msg} times between every node of the network and negligible t_p times, we can make the claim that $2t_{msg}$ is required between every execution of the critical section in a centralized algorithm with maximum contention for the critical section. The process exiting the critical section sends a *release* message to the controller. Upon receipt of the *release* message, the controller sends a *grant* message to the process at the head of the mutual exclusion queue. Likewise, we can claim that t_{msg} is required for a fully distributed algorithm. The process exiting the critical section sends a *reply* message to all deferred processes. Upon receipt of the *reply* message, the process at the head of the implicit mutual exclusion queue enters the critical section. Hence, we can expect twice as much time between every execution of the critical section in a centralized algorithm as in its fully distributed counterpart. Simulation results exactly confirm these observations. We can now predict how a hybrid algorithm, such as that of *Maekawa*, might perform on a fully-connected topology.

There are three possible scenarios that can occur in the *Maekawa* algorithm when a process exits the critical section and releases a node. First, the process that receives the release can lock the node and enter its critical section. Since there are $(K-1)$ processes which receive the release directly from a process, we can expect this to happen with probability $\frac{K-1}{N}$. Second, a process may release a node to itself, and a lock is then relayed to the locking process. We can expect this to happen with probability $\frac{1}{N}$. Finally, the process that receives the release may send a lock to the locking process, and that process then enters its critical section. We can expect this to happen the remainder of the time, or with probability $\frac{N-K}{N}$. In the first two cases, the interval between critical section executions is t_{msg}, as it would be in a fully distributed implementation. In the third case, the interval between critical section executions is $2t_{msg}$, as it would be in a centralized implementation. We can anticipate the time between each execution of the critical section for the *Maekawa* algorithm to be:

$$\frac{K}{N}t_{msg} + (\frac{N-K}{N})2t_{msg} = \frac{2N-K}{N}t_{msg}$$

For a system with 31 processes, we can expect $\frac{2N-K}{N}t_{msg} = 1.806t_{msg}$. Simulation results show an actual time of $1.727t_{msg}$; the relative error of our estimate is 4.6%.

We can now rank the performance of the three algorithms by the expected interval between successive executions of the critical section:

1. *RA* — t_{msg}
2. *Maekawa* — $\frac{2N-K}{N}t_{msg}$
3. *BS* — $2t_{msg}$

The interval between successive executions of the critical section captures our notions of updating state and, if necessary, relaying the decision as to which process has authority to enter the critical section. For example, the *RA* algorithm spends t_{msg} updating state; no relay is necessary, because f is a distributed function. In the *BS* algorithm, t_{msg} is required to update the controller's local state; because f is a centralized function applied only by the controller, another t_{msg} is required to relay the decision as to which process has been chosen to enter the critical section.

Utilization of the critical section is the percentage of observed time that the critical section is occupied by any process. The ranking just presented is consistent with utilization of the critical section. If we set the mean service time of the critical section S_{cs} equal to 1.0, then we can derive utilization using the utilization law from queueing theory:

$$U_{cs} = X_{cs} S_{cs} = X_{cs}$$

X is the throughput, or number of critical section usages per time unit. We can measure the throughput of the critical section by noting there is one execution of the critical section for each critical section usage. If we add the time spent in the critical section to the mean interval between successive executions of the critical section, we obtain throughput. Subsequently, we can compute utilization.

1. $U_{RA} = X_{RA} = \frac{1}{t_{cs} + t_{msg}}$
2. $U_{Maekawa} = X_{Maekawa} = \frac{1}{t_{cs} + \frac{2N-K}{N} t_{msg}}$
3. $U_{BS} = X_{BS} = \frac{1}{t_{cs} + 2t_{msg}}$

We will use this model to examine the performance of each algorithm on varying topologies when performance metrics are discussed in detail in Chapter 3. Our intention is to derive the interval between successive critical section executions. This interval is easily translated into utilization as just shown.

Chapter 3

Analytical Results for Fail-free Systems

We find that the metric of counting the number of messages sent for each use of the critical section is not by itself adequate to determine performance. We present a scenario where the algorithm that performs with the least efficiency sends the fewest messages, and the algorithm that performs most efficiently sends the most messages. Closer analysis of this metric leads us to examine which processes receive the messages and which processes send them for each use of the critical section. Additionally, we consider how these messages are distributed over time. An entirely different approach to measuring performance requires viewing the critical section as a shared resource for which processes throughout the distributed system contend. The critical section can be modeled using this approach by the classical "queue/server pair" from queueing theory. This view leads us to choose utilization as the metric of choice for ranking our algorithms.

We also find that, by counting the length of the chain of messages and using heuristics to measure the cost of message processing time and contention within the distributed system, we can accurately estimate the steady-state performance of our algorithms on fully-connected, ring and star topologies. Finally, we show that our approximation method accurately assesses the performance of a newly-published, tree-based distributed mutual exclusion algorithm.

3.1 Performance Metrics

The current metric in use to compare the performance of distributed mutual exclusion algorithms is the number of messages sent for each usage of the critical section. Supposedly, the fewer messages required, the better the algorithm. The widespread use of this metric is not surprising. The need to exchange state information via messages is an important factor in the performance of distributed systems. The interconnection network is a shared resource whose finite capacity should not be wasted. From a practical standpoint, the number of message transmissions per critical section usage is a well-defined and easily calculated quantity. But this metric only measures the cost of propagating messages throughout the distributed system to update the local state of each process and ignores the tradeoff that may exist if a process maintains minimal state. For example, if the decision as to which process has priority to enter the critical section must be relayed to the chosen process, that extra transmission may negate any gains from limiting the number of messages sent to exchange state information.

To illustrate that the current metric measures only the cost of updating local state, we will look at a fully-connected topology where message processing is not a factor in the efficiency of the network. Under this assumption, we can show that the metric of number of messages sent per critical section usage does not accurately assess the performance of distributed mutual exclusion algorithms. Rather, the time to relay the decision as to which process is chosen to enter the critical section is the determining factor in performance. With the following assumptions, we can derive some simple analytic results:

1. The time, t_{cs}, that any process spends in its critical section is finite and

constant:

$$\forall i : 1 \leq i \leq N :: t_{cs}[i] = t_{cs}$$

where $t_{cs}[i]$ is the time spent in the critical section by process i.

2. The time, t_{msg}, that a message takes to transit the network between any two adjacent processes is finite and constant:

$$\forall i \forall j : 1 \leq i, j \leq N :: t_{msg}[i, j] = t_{msg}$$

where $t_{msg}[i, j]$ is the time it takes a message to go from process i to process j.

3. Message processing time, t_p, is negligible.

4. When a process exits its critical section, there is always another process waiting to acquire the critical section (notice that this does not preclude a system with a single process, whereupon exiting its critical section, immediately attempts to acquire it again).

5. Processes never fail.

No assumptions are made about the length of time a process spends in its non-critical section nor how many processes are in the system.

When a process in the *BS* algorithm (recall that *BS* is a centralized implementation) exits the critical section, a *release* message is sent to the controller. The controller accepts the *release* message and then sends a *grant* message to the next process with priority to enter the critical section. The result is that time equal to $2t_{msg}$ is required to effect mutual exclusion between executions of the critical section. In the *RA* algorithm, (*RA* is a fully distributed implementation), a *reply* message is sent directly to the next process that will enter its critical section; this process receives the *reply* message, and immediately enters its critical section. The result is time equal to t_{msg} to effect mutual exclusion.

We can view this result more clearly with response time:

N — average number of processes awaiting mutual exclusion when a process joins the queue

R — response time: from when a process requests mutual exclusion until it is granted

Under our assumptions, R is equal to the time it takes for N processes to execute their critical sections plus the time it takes to grant permission for N processes to enter their critical sections. If we assume that the average number of processes awaiting mutual exclusion, N, is identical for the *BS* and *RA* algorithms, we can make the following comparison:

$$N_{BS} = N_{RA}$$

$$R_{BS} = (N)t_{cs} + 2(N)t_{msg} = (N)(t_{cs} + 2t_{msg})$$

$$R_{RA} = (N)t_{cs} + (N)t_{msg} = (N)(t_{cs} + t_{msg})$$

We can observe that R_{BS} is $(N * t_{msg})$ greater than R_{RA}. Actually, since the time between critical section executions is greater for the *BS* algorithm than for the *RA* algorithm, N_{BS} will be larger than N_{RA}; hence, the difference in expected mean response times will be even greater than just shown.

We can establish a similar result for the response time of the *Maekawa* algorithm. Our previous analysis in Section 2.3 assumed that N was equal to the number of processes in the system; this also established K as the number of processes in each subset awaiting mutual exclusion. We now expect N to be smaller than the number of system processes. But just as N will be smaller, K will also be proportionately smaller. Consequently, we can still use our previous analysis to observe:

$$R_{Mae} = (N)(t_{cs} + \frac{K}{N}t_{msg} + (\frac{N-K}{N})2t_{msg}) = (N)(t_{cs} + \frac{2N-K}{N}t_{msg})$$

We can conclude that the fully distributed algorithm of *RA* performs best, the centralized algorithm of *BS* performs worst, and the hybrid algorithm of *Maekawa* performs somewhere in between. Since *Maekawa* sends fewer messages than *RA* for each use of the critical section, and the centralized implementation of *BS* sends fewest of all, it is clear that other metrics, in addition to the classical "messages sent", must be found to compare performance. This observation has also been made in reference to centralized and fully distributed implementations of the committee coordination algorithm [Bag].

We can build on the idea of the number of messages sent per usage of the critical section with the addition of two new metrics. Because messages burden the sites in the distributed system that must transmit and receive them, we consider the number of messages that must be transmitted or received by each process in the system for each particular mutual exclusion algorithm. And, because mutual exclusion messages contend for the same resources as other messages, we consider how these messages are distributed over time.

A different approach to selecting performance metrics involves considering the critical section as a shared resource for which processes throughout the distributed system contend. The critical section can be modeled by the classical "queue/server pair" from queueing theory. The server represents the critical section which can "serve" only one process at a time. The queue represents the set of processes wishing to use the critical section.

Using this model, we immediately recognize four performance metrics: the number of critical section usages per time unit (throughput), the elapsed time between requesting the critical section and subsequently exiting the critical section after acquiring it (response time), the number of processes waiting to use the critical section (queue length), and the proportion of time the critical section is in use (utilization).

The utilization, U, and throughput, X, are related by the utilization law,

$$U = XS$$

where S is the mean time the critical section is held. Given S, this law allows us to study only one or the other of these metrics. Any conclusions obtained from studying one of the two measures can be immediately translated into the other using the utilization law. The algorithms will be ranked the same by these two metrics.

Similarly, the mean queue length, Q, and mean response time, R, are related by Little's Law:

$$Q = RX$$

Given X, either the queue length or response time, but not both, must be studied. However, we have elected not to study either of these metrics. We are not neglecting these metrics because they are unimportant. Rather, we can argue that if our particular algorithm has a greater utilization of the critical section than another algorithm, then that algorithm has a shorter queue length and response time as well. Since it is our intention to rank these algorithms, and studying Q and R lends no further insight into our ranking, we choose to ignore them. This decision is discussed in more detail later in this chapter in Section 3.4.

Finally, we will examine an algorithm's performance on varying topologies. Using our model of mutual exclusion algorithms, we will rank each algorithm according to its efficiency on fully-connected, ring and star topologies.

We propose the five following metrics to measure the efficiency of mutual exclusion algorithms:

1. spatial distribution of messages sent per use of the critical section
2. spatial distribution of messages received per use of the critical section
3. temporal distribution of messages sent per use of the critical section
4. utilization of the critical section
5. robustness (how an algorithm's performance is affected by topology)

3.2 Spatial Distribution of Messages

Two metrics which may be useful when message processing time is significant are the number of messages sent or received per use of the critical section. Table 3.1 and Table 3.2 reveal which processes do most of the work to achieve mutual exclusion.

Algorithm	Controller	User of CS	$N-1$ Other Processes	Total Msg
BS	1	2	0	3
Maekawa	—	$2\sqrt{N}$	$\frac{2\sqrt{N}}{(N-1)}$	$4\sqrt{N}$
Schneider	—	$2*(N-1)$	$2*(N-1)$	$2*N*(N-1)$
RA	—	$N-1$	1	$2*(N-1)$
CR	—	$0\cdots(N-1)$	1 or 0	$0\cdots 2*(N-1)$

Table 3.1: Spatial Distribution of Messages Sent for Mutual Exclusion

Algorithm	Controller	User of CS	$N - 1$ Other Processes	Total Msg
BS	2	1	0	3
Maekawa	—	$2\sqrt{N}$	$\frac{2\sqrt{N}}{(N-1)}$	$4\sqrt{N}$
Schneider	—	$2 * (N - 1)$	$2 * (N - 1)$	$2 * N * (N - 1)$
RA	—	$N - 1$	1	$2 * (N - 1)$
CR	—	$0 \cdots (N - 1)$	1 or 0	$0 \cdots 2 * (N - 1)$

Table 3.2: Spatial Distribution of Messages Received for Mutual Exclusion

The tables show us that in the *BS* algorithm, processes that do not seek access to the critical section process no messages. This is the ideal implementation of mutual exclusion: a process that does not seek access to the critical section never processes a single control message. We can observe that the *CR* algorithm is next best. In fact, once a process in the *CR* algorithm has replied to all other processes that seek access to the critical section, it will not process another control message until it desires access to the critical section for itself. The *RA* algorithm also measures up well; a process that is not gaining access to the critical section sends and receives only one message for each entry to the critical section. In contrast, the *Schneider* algorithm equally burdens every process, even those that never seek access to the critical section. A process that never seeks entry to the critical section will find itself processing a number of messages equal to the process that gains entry.

When we look at processes that seek access to the critical section, we can draw similar conclusions. In the *BS* algorithm, the process that seeks access to the critical section sends two messages and receives one. Hence, the user of the critical section does minimal message processing. The controller is required to send a single message and receive two. The *Maekawa* algorithm requires the process seeking entry to the critical section to send and receive fewer messages than the *RA* algorithm any time N exceeds 4. The *CR* algorithm will require, at most, the same number of messages as the *RA* algorithm; in times of light contention, especially, we expect the *CR* algorithm to do better than its *RA* counterpart. The *Schneider* algorithm requires the process seeking access to the critical section to send and receive at least twice the number of messages as its competitors. If message processing time is significant, we can expect the *Schneider* algorithm to be less efficient than the other algorithms.

3.3 Temporal Distribution of Messages

Another metric that deserves consideration is the temporal distribution of messages sent to effect mutual exclusion. The following series of timelines show where surges in the number of messages sent occur over time in a fail-free environment with maximum contention for the critical section.

These figures show the types of messages employed by each algorithm. The messages above the timelines are requests made at arbitrary times by processes in the distributed system; some algorithms require these messages to be acknowledged. The messages below the timelines are messages sent when a process exits the critical section; these messages are always sent when the critical section transitions from *busy* to *idle*. Again, an algorithm may require these messages to be acknowledged. Other algorithms, such as those of *BS* and *Maekawa*, require a controller to relay state information to the next process waiting to enter the critical section.

Figure 3.1 shows that, in the centralized implementation of *BS*, bursts of messages do not impact the system. The claim that a centralized mutual exclusion algorithm is inefficient because the controller becomes a bottleneck appears to be unfounded. When the system reaches steady-state as it has in Figure 3.1, requests for the critical section and releases occur at approximately the same rate. Therefore, contention at the controller is minimal and is not a major contributing factor to its performance. Rather, the lesser efficiency of the *BS* algorithm can be attributed to the number of messages sent between every critical section execution (which we have shown to be two), which is a function of the state maintained by each process in the system.

Figure 3.2 reveals that *relinquish* messages are unnecessary when maximum contention occurs in the *Maekawa* algorithm.

In contrast to the timeline of the *BS* algorithm in Figure 3.1, the timelines shown in Figure 3.2 through Figure 3.5 show that bursts of messages flood the system at regular intervals for virtually every distributed implementation. Figure 3.3 reveals that the *Schneider* algorithm continuously sends large numbers of messages into the network. Large numbers of messages can significantly impact the underlying computation of the system if control messages are routed through a small number of network nodes.

If large volume traffic significantly impedes the performance of a particular network, the *BS* algorithm may very well prove preferable to its fully distributed counterparts.

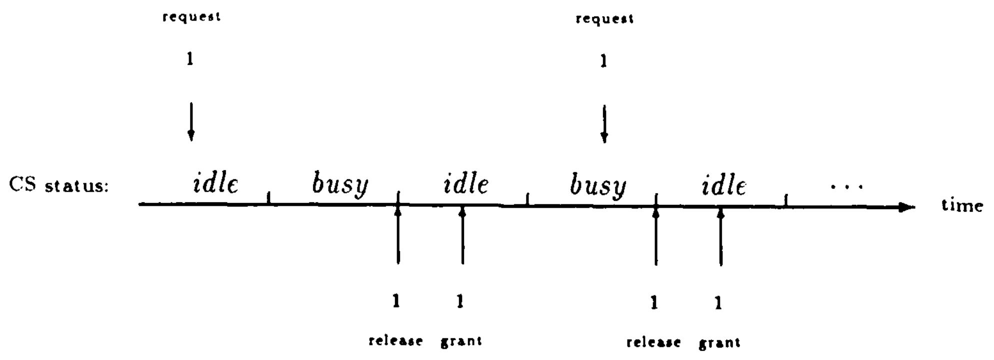


Figure 3.1: Temporal Distribution of Messages Sent for Buckley and Silberschatz's Algorithm

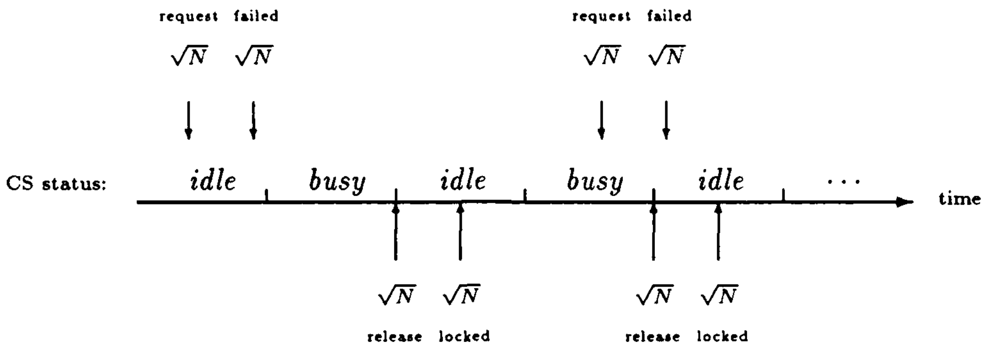


Figure 3.2: Temporal Distribution of Messages Sent for Maekawa's Algorithm

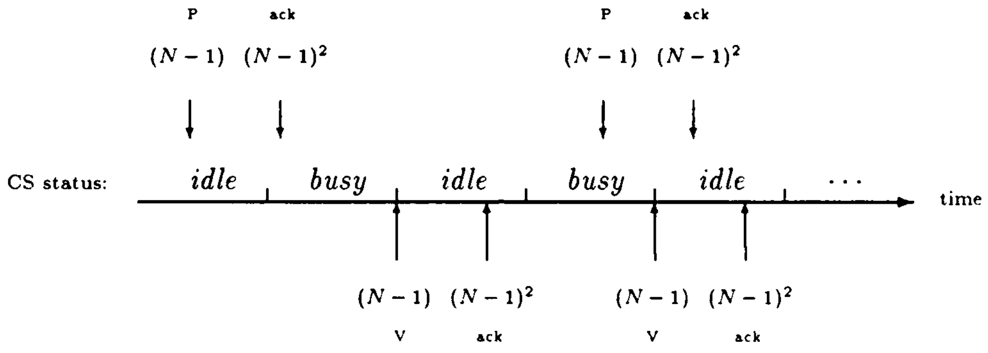


Figure 3.3: Temporal Distribution of Messages Sent for Schneider's Algorithm

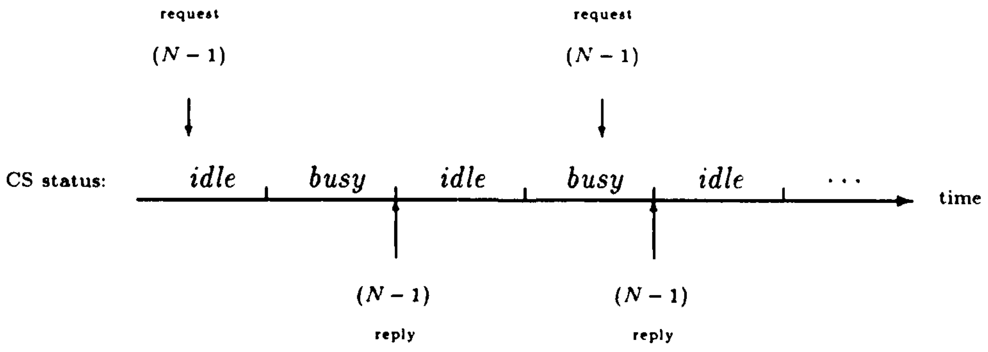


Figure 3.4: Temporal Distribution of Messages Sent for Ricart and Agrawala's Algorithm

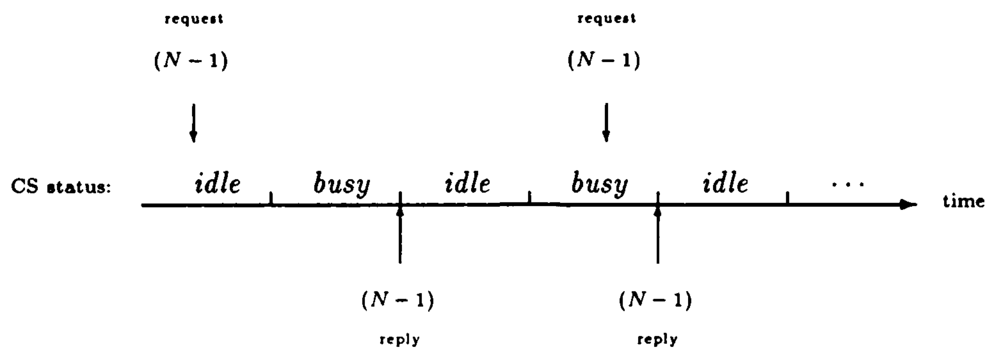


Figure 3.5: Temporal Distribution of Messages Sent for Carvalho and Roucairol's Algorithm

3.4 Utilization of the Critical Section

Our fourth metric is utilization of the critical section. Utilization is defined to be the percentage of observed time the critical section is occupied by any process.

The metric of utilization was chosen over all other metrics as the basis for ranking distributed mutual exclusion algorithms. The time between critical section executions, response time and queue length are other candidates for providing the basis for our ranking. We will now examine why utilization is the appropriate metric for our purposes.

We would like to assign a value to an algorithm, creating what might be thought of as a fingerprint of its performance. If two algorithms have the same fingerprint, then we should be able to claim that they are equivalently efficient. Likewise, if the fingerprint of one algorithm is in some way smaller than the fingerprint of another algorithm, we should be able to say that the efficiency of the one with the smaller fingerprint exceeds the efficiency of the other. Two questions arise: does such a fingerprint exist; and if so, how might one compute it.

The reason for computing such a fingerprint is that once the fingerprint is obtained, a ranking immediately follows. Then, instead of having to compare each algorithm to every other algorithm to see how it performs relative to all the others, the fingerprint lends immediate insight into how the algorithms stack up against one another in terms of performance.

The way to create such a fingerprint is to compute a ratio using a common metric for each algorithm. We have chosen to compute a ratio consisting of a common metric for each algorithm and the same metric of the ideal mutual exclusion algorithm.[1] The ideal mutual exclusion algorithm, as we define it, continuously utilizes the critical section; that is, there is never a time when the critical section is idle. This implies that there is no significant time from the moment when one process exits the critical section, until the moment the next process enters it. That is, the interval between successive executions of the critical section is 0. This observation immediately makes it clear why we cannot use the metric of interval between successive executions of the critical section as a basis for computing the fingerprint: the ratios $\frac{0}{interval}$ and $\frac{interval}{0}$ provide no basis from which to compute a value. Utilization, on the other hand, provides a simple basis from which we can compute our fingerprint.

For the ideal algorithm, $U_{ideal} = 1.0$, where U is utilization. By taking the ratio of U_{ideal} over the utilization of a given mutual exclusion implementation, we can immediately determine how that algorithm performs relative to the ideal implementation. By then computing this ratio for every algorithm, we

[1]The notion of an ideal algorithm is not an original one. The notions of *idealistic* and *realistic* are commonly used in reference to architectures.

can create a fingerprint for each algorithm and immediately establish a ranking between them.

For example, when message processing time is negligible, successive entries into the critical section for the *RA* algorithm occur at the rate of once every $(t_{cs} + t_{msg})$ time units when there is always a process waiting in the mutual exclusion queue; or more precisely,

$$X_{cs} = \frac{1}{t_{cs} + t_{msg}}$$

where X_{cs} is the throughput of the critical section. By the utilization law,

$$U_{cs} = X_{cs} S_{cs} = \frac{S_{cs}}{t_{cs} + t_{msg}}$$

But we know that $S_{cs} = t_{cs} = 1.0$, so we have:

$$U_{cs} = \frac{1}{1.0 + t_{msg}}$$

which reflects the utilization of the *RA* algorithm. We can derive a similar result for the *BS* algorithm:

$$U_{cs} = \frac{1}{1.0 + 2t_{msg}}$$

By assuming $t_{msg} = 0.1$, we can now create the following fingerprints:

$$\frac{U_{ideal}}{U_{RA}} = \frac{1.0 + t_{msg}}{1.0} = 1.1$$

$$\frac{U_{ideal}}{U_{BS}} = \frac{1.0 + 2t_{msg}}{1.0} = 1.2$$

The ranking of the algorithms follows immediately, and we can claim that the *RA* algorithm utilizes the critical section more efficiently under our given assumptions. Note that the fingerprint can be computed independently of the number of processes in the network. Our only assumption is that a process is always waiting in the mutual exclusion queue when a process exits the critical section.

We now address the issue of why one should choose utlization as a metric instead of response time or queue length. For mutual exclusion, there is only one ideal utilization: 1.0. However, the queue length and response time for the ideal algorithm vary according to the number of processes in the distributed system; when the arrival rate of processes and the service time of the critical section are held constant, increasing the number of processes in the system increases both queue length and response time. This observation implies that the ideal metric would have to be recomputed each time the number of processes in the system changes. In addition, since the ideal value for the metric is inconsistent for systems with differing numbers of processes, there is no common basis on which to compare the performance of an algorithm with itself when it runs on two networks where the number of processes is not the same. Hence, the ideal utilization is not only easy to compute (it's always 1.0), but it provides a common basis for the comparison of the performance of all mutual exclusion algorithms in all given distributed systems, independently of number of processes.

We have also noted in Section 2.3.2 that our ranking remains unchanged in systems with maximum contention for the critical section whether we rank our algorithms using response time, queue length, utilization or time between critical section executions. Hence, our motivation in selecting utilization as our metric of choice is derived primarily from our ability to compute a meaningful fingerprint from it.

3.5 Effects of Topology

We would like to estimate how an algorithm performs on varying topologies in order to determine if a particular algorithm is suitable for all networks, or if perhaps it is suited only for a particular topology. We will use the model of mutual exclusion algorithms developed in Chapter 2 to derive results to answer this question. We find that if we estimate the time between critical section executions for the centralized and fully distributed implementations, we can use these results to approximate the performance of the *Maekawa* hybrid implementation. We will specifically look at three very different topologies: fully-connected, ring and star. The fully-connected topology represents the ideal topology in the sense that the diameter of the network is one. The ring topology is the other end of the spectrum, giving us the greatest diameter possible short of a network where all processes are linked by a straight line. Finally, the star topology falls somewhere in between with a network diameter of two; an additional factor of the star topology is its use of a "hub" through which all traffic is routed. We will look at three of our algorithms: *BS*, *Maekawa*, and *RA*. We will not look at the *CR* algorithm, because it is so similar to the *RA* algorithm. We will not investigate the performance of the *Schneider* algorithm; it performs so poorly in a fail-free environment compared to the others that it is unlikely anyone will seriously consider it for implementation. Therefore, we limit our investigation of the *CR* and *Schneider* algorithms to simulation results presented in Chapters 4 and 5. Throughout our analyses, we assume that every process is equally likely to seek access to the critical section.

3.5.1 Fully-connected Topology

Figure 3.6 shows a fully connected topology. We have already rated our algorithms on a fully-connected topology when message processing time is negligible:

1. RA — t_{msg}
2. $Maekawa$ — $\frac{2N-K}{N} t_{msg}$
3. BS — $2t_{msg}$

We can see that this ranking remains unchanged, regardless of the number of processes in the distributed system. Without considering the additional cost of message processing time, we can never measure the impact that the number of messages will have on the network. Therefore, any model that we develop must account for the number of messages sent by an algorithm. Additionally, since messages "collide" in the network, any model that we develop must also account for contention. We now show that we can measure message passing overhead due to the sending of messages as well as contention in the network.

When message processing time was not significant, our results were computed by the simple formulae given in our ranking. When message processing time becomes significant, our formulae are different. Our first task is to determine the length of the mutual exclusion queue in our algorithms. We can derive the queue length using an M/M/1/∞/m queueing formula. An M/M/1/∞/m queueing formula can be derived under the following assumptions:

1. single server, infinite queue capacity

2. first come, first served queueing discipline
3. negative exponential inter-arrival time
4. negative exponential service time
5. closed system with m processes
6. system is in steady state

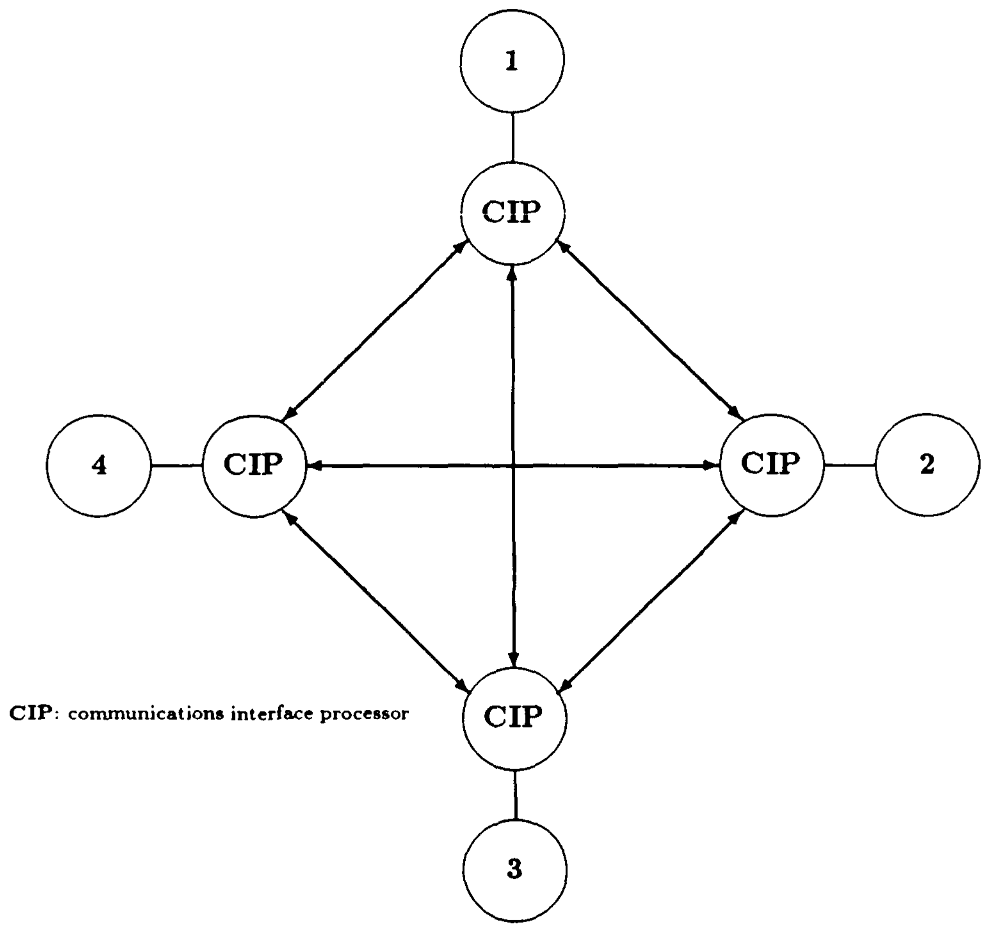


Figure 3.6: An Example of a Fully-connected Distributed System

We violate the exponential service time assumption; but as we shall see, we still derive excellent results for a fixed, constant service time. The queue length, though technically unbounded, will never exceed m since we have a closed system. The queueing network we want to model is shown in Figure 3.7.

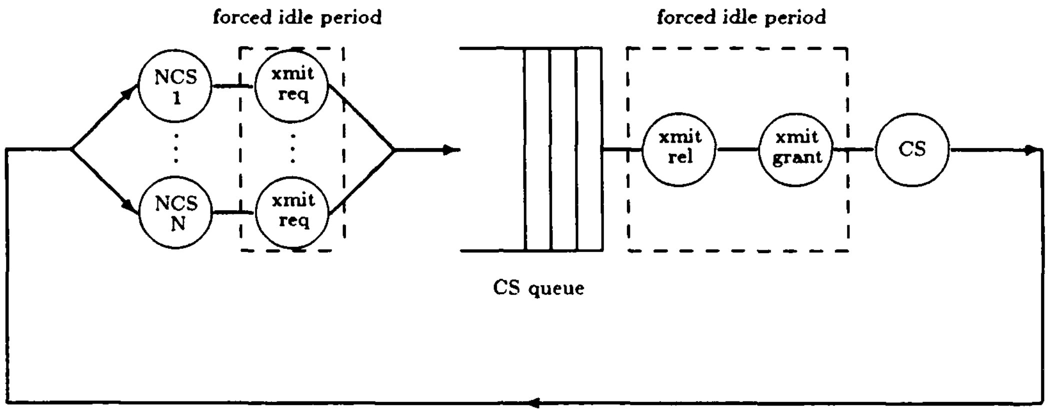


Figure 3.7: Queueing Network for the *BS* Algorithm

The forced idle periods make it impossible to derive exact analytic results. When a process exits the *NCS* server in the queueing network of Figure 3.7, it must wait time equal to *xmit req* until it can enter the mutual exclusion queue. Similarly, when it exits the queue, the process must wait time equal to *xmit rel* followed by *xmit grant* before it can enter the *CS* server. The queueing network that we can model and for which we can derive analytical results is shown in Figure 3.8.

In the queueing network of Figure 3.8, the time *xmit req* has been ignored, and the times *xmit rel* and *xmit grant* have been incorporated into *CS*. The time *xmit req* is ignored, because the time spent within the *NCS* is so much larger in comparison; we also have the problem that *NCS* has an exponential distribution, while *xmit req* is fixed and constant. On the other hand, the times *xmit rel* and *xmit grant* are fixed, constant and significant in comparison to the fixed and constant value of *CS*. We cannot ignore them, and we are able to account for their effect by making them part of *CS*. We can now present our M/M/1/∞/m formula and derive analytical results for the network shown in Figure 3.8. The following queueing formula was derived using a Markov chain.

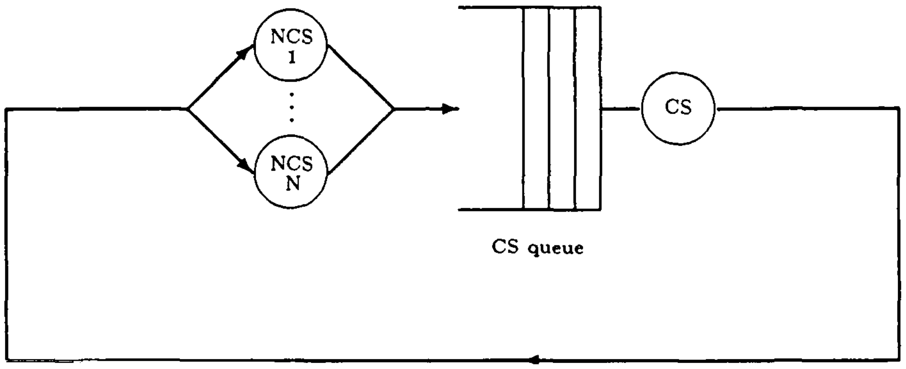


Figure 3.8: Queueing Network for Analytical Results

$$\lambda = \text{arrival rate of requests for the critical section}$$
$$S = \text{service time of the critical section}$$
$$p(n) = \text{probability that there are } n \text{ processes in the queue}$$

$$p(0) = \frac{1}{\sum_{n=0}^{m} \frac{m!}{(m-n)!}(\lambda S)^n}$$

$$p(n) = \frac{m!}{(m-n)!}(\lambda S)^n p(0)$$

$$Q = \sum_{n=1}^{m}(n * p(n))$$

Each *xmit* in our network represents a single message transmission; each transmission can be thought of as a link in a chain of messages between successive entries to the critical section.[2] In a centralized implementation, for example, the length of the chain of messages between critical section executions is two. This leads to the following minimum delay between critical section executions:

$$t_p + t_{msg} + t_p + t_p + t_{msg} + t_p = 2t_{msg} + 4t_p$$

Let's examine how we derived this formula. A message is placed by the sender's CIP onto its outgoing communications path in time t_p. It takes time t_{msg} to arrive at the receiver. The receiver's CIP removes the message from the incoming communications path and hands it off to the receiver in time t_p. When the receiver replies, an equivalent sequence of events takes place in the opposite direction. If $t_{msg} = 0.1$ and $t_p = 0.005$, then we compute S_{cs} in our queueing formula as $t_{cs} + 2t_{msg} + 4t_p = 1.0 + 0.2 + 0.02 = 1.22$ time units.

For a fully distributed implementation, the length of the chain of messages between critical section executions is one. Our minimum delay between critical section executions is:

$$t_p + t_{msg} + t_p = t_{msg} + 2t_p$$

If $t_{msg} = 0.1$ and $t_p = 0.005$, then we compute S_{cs} in our queueing formula as $t_{cs} + t_{msg} + 2t_p = 1.0 + 0.1 + 0.01 = 1.11$ time units.

Our next step is to determine a value for λ. Since each process delays in the *NCS* for a mean time of 10.0 time units, we can compute the arrival rate into the queue as $\frac{1}{10} = 0.1$ arrivals per time unit. This value is constant for all uses of the M/M/1/∞/m queueing formula.

[2]The notion of the length of the chain of messages between events in a distributed system was first advanced by G. Buckley in her PhD proposal at the University of Texas at Austin.

The result we obtain from the M/M/1/∞/m queueing formula is especially important for the *RA* algorithm, because the queue length tells us how many *reply* messages must be sent when a process exits the critical section. It is important for all of our algorithms for a second reason: it enables us to model contention at the CIP when a message releasing the critical section arrives at its destination. An example will show how this can be done.

We will use our M/M/1/∞/m formula to derive the mean queue length for $m = 31$ and $S_{cs} = 1.11$ for the *RA* algorithm:

$$Q_{RA} = 21.990992 \approx 22$$

This number reveals that the average length of the mutual exclusion queue in the *RA* algorithm is about 22 processes. When a process is in the mutual exclusion queue, it does not send any messages. It will defer a reply to any requests made subsequent to its own. *If the mean queue length is 22 processes* for a system with 31 processes, then we can argue that the only processes that will send messages into the system are the 9 processes not waiting for the critical section. A heuristic that we can employ is to assume that these 9 processes will contend for a common communications path. Using our queueing formula, we can compute the mean number of messages on the communications path at any one time. We can then assume that this mean value represents the worst-case contention at the CIP, since all messages sent to a particular receiver in a fully-connected network converge at a common destination.

We can now measure contention, using the M/M/1/∞/m queueing formula, for the communications path between the requesting process and every other process by letting $m = 9$ and $S_{path} = 0.11$. We have selected $S_{path} = 0.11$, because a message will spend at least 0.11 time units on the communications path between adjacent processes when $t_{msg} = 0.1$ and $t_p = 0.005$.

$$Q_{path} = 0.10722$$

We get worst-case contention at the CIP by assuming that Q_{path} is equal to the number of messages at the CIP when the message releasing the critical section arrives.

When the *RA* algorithm releases the critical section, the message releasing the critical section to the next entering process will be sent, on the average, half-way through the outgoing messages. For a system with 31 processes, we have computed $Q_{cs} = 22$. The *reply* message can expect to find a queue length at the CIP of 0.10722. Our time between critical section executions for the *RA* algorithm can now be computed as follows:

$$\frac{Q_{cs}}{2} t_p + t_{msg} + Q_{CIP} t_p + t_p = 0.1630351$$

Simulation results show 0.158384 time units, for a relative error of 2.9%.

For 21 processes, we can make the following computation:

$$Q_{cs} = 11.993402 \approx 12$$

$$Q_{CIP} = 0.107022$$

Our time between successive executions of the critical section becomes:

$$\frac{Q_{cs}}{2} t_p + t_{msg} + Q_{CIP} t_p + t_p = 0.1355351$$

Simulation results show 0.132984 time units, for a relative error of 1.9%. The M/M/1/∞/m queueing formula used in conjunction with our heuristic appears to give excellent results for the *RA* algorithm.

We can make similar computations for the *BS* algorithm. We can again assume that those processes waiting in the mutual exclusion queue send no messages; only the processes executing outside the critical section will send messages in the form of requests. Since all messages go to the controller, we can still use our heuristic to measure contention at the CIP of the controller. For $m = 31$ and $S_{cs} = 1.22$:

$$Q_{cs} = 22.803282 \approx 23$$

$$Q_{CIP} = 0.094040$$

The time between successive executions of the critical section is:

$$2t_{msg} + 4t_p + Q_{CIP}t_p = 0.2204702$$

Simulation results show 0.211901 time units, for a relative error of 4.0%.

For 21 processes, our computation becomes:

$$Q_{cs} = 12.803960 \approx 13$$

$$Q_{CIP} = 0.09040$$

The time between successive executions of the critical section is:

$$2t_{msg} + 4t_p + Q_{CIP}t_p = 0.2204702$$

Simulation results show 0.207932 time units, for a relative error of 6.0%. The results derived by our model are conservative. Our results will be conservative, because in our simulations the process co-located with the controller can exchange messages with the controller in negligible time. This observation leads us to identify another heuristic that we can apply to make the results of our model more accurate: about 1 of every 31 messages sent to the controller will be sent instantaneously in a system with 31 processes; similarly, about 1 of every 21 messages sent to the controller will be sent instantaneously in a system with 21 processes. Therefore, we can reduce the result obtained by our model for m processes by $\frac{1}{m}$. Using this heuristic, we compute for 31 processes a mean of 0.213358 time units between each execution of the critical section, for a new relative error of 0.7%; for 21 processes, we compute 0.2099716 time units, for a new relative error of 1.0%.

We can conclude that this model yields excellent results for the *RA* and *BS* algorithms on a fully-connected topology. We now use these results to estimate the performance of the *Maekawa* algorithm.

With $m = 31$, $K = 6$, $t_{msg} = 0.1$ and $t_p = 0.005$, we can initially compute a rough approximation of the performance of the *Maekawa* algorithm. By assuming performance equal to the *RA* algorithm $\frac{K}{N}$ of the time and performance equal to the *BS* algorithm $\frac{N-K}{N}$ of the time, we compute:

$$.11(\frac{K}{N}) + .22(\frac{N-K}{N}) = .11(\frac{6}{31}) + .22(\frac{25}{31}) = 0.1987097$$

We now set $S_{cs} = t_{cs} + 0.1987097 = 1.1987097$ and compute the following results with our M/M/1/∞/m queueing formula:

$$Q_{cs} = 22.6577 \approx 23$$

$$Q_{CIP} = 0.09404$$

In our analysis of the *Maekawa* algorithm in Section 2.3.2, we cited three scenarios for the *Maekawa* algorithm and assigned a probability for each one. Our times between critical section executions for the three scenarios become:

$$\frac{K-1}{2}t_p + t_{msg} + Q_{CIP}t_p + t_p = \frac{K+1}{2}t_p + t_{msg} + Q_{CIP}t_p = 0.1179702$$

$$t_{msg} + 2t_p + Q_{CIP}t_p = 0.1104702$$

$$\frac{K-1}{2}t_p + 2t_{msg} + 3t_p + Q_{CIP}t_p = \frac{K+5}{2}t_p + 2t_{msg} + Q_{CIP}t_p = 0.2279702$$

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	.207932	.209972	.010
	.010	.226261	.229554	.015
	.015	.249974	.249007	.004
	.020	.263759	.268669	.019
	.025	.284512	.288409	.014
RA	.005	.132984	.135535	.019
	.010	.170289	.176032	.034
	.015	.215405	.214187	.006
	.020	.249163	.252438	.013
	.025	.289648	.290785	.004
Mae	.005	.189798	.200947	.059
	.010	.218217	.221985	.017
	.015	.243800	.245021	.005
	.020	.270454	.267817	.010
	.025	.295950	.290924	.017

Table 3.3: Application of the Model to 21 Processes in a Fully-connected Topology

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	.211901	.213358	.007
	.010	.231104	.233257	.009
	.015	.250220	.253023	.011
	.020	.269114	.273002	.014
	.025	.289619	.293060	.012
RA	.005	.158384	.163035	.029
	.010	.221061	.226032	.022
	.015	.290419	.289187	.004
	.020	.350377	.352438	.006
	.025	.422540	.415785	.016
Mae	.005	.201890	.206438	.023
	.010	.230817	.232322	.007
	.015	.255324	.258300	.012
	.020	.286294	.284038	.008
	.025	.314168	.310087	.013

Table 3.4: Application of the Model to 31 Processes in a Fully-connected Topology

Using the probabilities associated with each scenario, we obtain:

$$\frac{K-1}{N}(0.1179702) + \frac{1}{N}(0.1104702) + \frac{N-K}{N}(0.2279702) = 0.2064379$$

Simulation results show 0.201890, for a relative error of 2.3%.

For $N = 21$ and $K = 5$, our computation becomes:

$$Q_{cs} = 12.658563 \approx 13$$

$$Q_{CIP} = 0.09404$$

Our times between successive executions of the critical section for the three scenarios become:

$$\frac{K-1}{2}t_p + t_{msg} + Q_{CIP}t_p + t_p = \frac{K+1}{2}t_p + t_{msg} + Q_{CIP}t_p = 0.1154702$$

$$t_{msg} + 2t_p + Q_{CIP}t_p = 0.1104702$$

$$\frac{K-1}{2}t_p + 2t_{msg} + 3t_p + Q_{CIP}t_p = \frac{K+5}{2}t_p + 2t_{msg} + Q_{CIP}t_p = 0.2254702$$

Using the probabilities associated with each scenario, we obtain:

$$\frac{K-1}{N}(0.1154702) + \frac{1}{N}(0.1104702) + \frac{N-K}{N}(0.2279702) = 0.2009464$$

Simulation results show 0.189798 time units, for a relative error of 5.9%.

We can conclude that our model also yields excellent results for the *Maekawa* algorithm in a fully-connected topology. Table 3.3 and Table 3.4 summarize our findings, for 31 and 21 processes, for all three algorithms in a fully-connected topology.

Our results lose their accuracy as contention for the critical section decreases, until it is unusable for systems with light contention. Our model is not accurate for periods of light contention, because we cannot predict the length of time between critical section executions; i.e., the system is not in steady state. In the *RA* algorithm, for example, if no process is waiting for the critical section, the length of the chain of messages between executions of the critical section becomes two: a process sends a *request* message to all other processes and receives a *reply* message from each of them. In addition, we can expect intervals when no process seeks access to the critical section. Without a reasonable guess for these values, we cannot model the delay between critical section executions using the methodology we have just illustrated.

3.5.2 Ring Topology

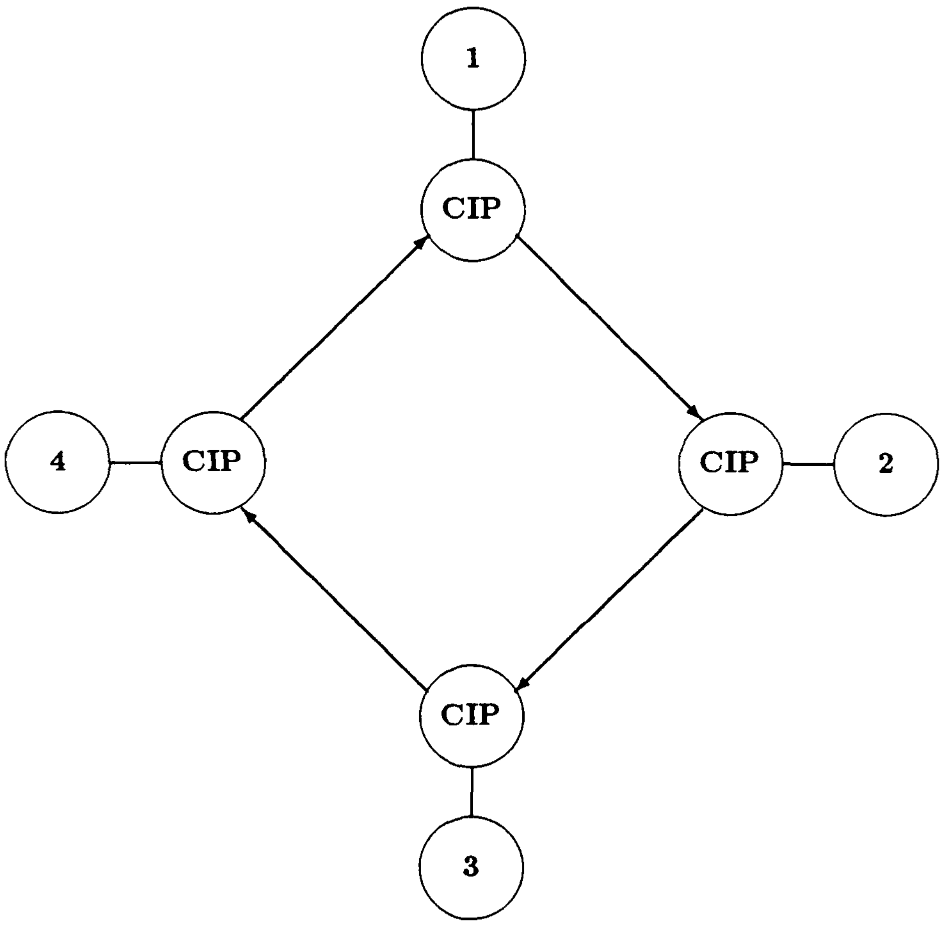


Figure 3.9: A Network with a Ring Topology

Figure 3.9 shows a ring topology. There is a single, uni-directional path through each vertex in the network. Each vertex has a single incoming edge and a single outgoing edge. Messages can be passed in only one direction and must be relayed when a message is addressed to a process more than one hop away from the sending process.

We assume that the system has multicasting capability, where a single message can be addressed to all or a subset of other processes. Hence, a process using the *RA* algorithm can send a single *request* message through the ring to notify all system processes of its intention to enter the critical section. Likewise, a process can send a single *reply* message when it exits the critical section. We are now ready to derive our analytical results.

When a process in the *RA* algorithm exits the critical section, it sends a *reply* message onto the ring. We can expect the *reply* message to travel, on the average, half-way around the ring before it reaches the process which is next to enter the critical section. If message processing time is negligible, then the time between each execution of the critical section is $\frac{N}{2}t_{msg}$ for systems with an even number of processes and $\frac{N-1}{2}t_{msg}$ for those with an odd number of processes.

When a process in the *BS* algorithm exits the critical section, it sends a *release* message to the controller. The controller then sends a *grant* message to the next process with priority to enter the critical section. When this process subsequently exits the critical section, it will send a message releasing the critical section back to the controller. We can observe that exactly one trip around the ring is required for each execution of the critical section. If message processing time is negligible, then the time between each execution of the critical section is Nt_{msg}.

We can again see that the length of the chain of the messages between each execution of the critical section is the determining factor in the efficiency of these two algorithms. We now use these results to estimate the performance of the *Maekawa* algorithm.

With our assumptions, the *Maekawa* algorithm approximates the behavior of the *RA* algorithm $\frac{K}{N}$ of the time; it approximates the behavior of the *BS* algorithm $\frac{N-K}{N}$ of the time. This rough analysis gives us an estimated time between successive executions of the critical section for the *Maekawa* algorithm of:

$$\frac{K}{N}(\frac{N}{2}t_{msg}) + \frac{N-K}{N}(Nt_{msg}) = \frac{K}{2}t_{msg} + (N-K)t_{msg} = \frac{2N-K}{2}t_{msg}$$

where N is even, and

$$\frac{K}{N}(\frac{N-1}{2}t_{msg}) + \frac{N-K}{N}(Nt_{msg}) = \frac{KN-K}{2N}t_{msg} + (N-K)t_{msg} = \frac{2N^2-NK-K}{2N}t_{msg}$$

where N is odd.

We again find the *Maekawa* algorithm performing somewhere inbetween the *RA* and *BS* algorithms, although its behavior is dominated by the less efficient centralized portion of the algorithm.

In a ring with 31 processes, we can expect the *RA*, *BS*, and *Maekawa* algorithms to be ranked as follows:

1. *RA* — $15t_{msg}$
2. *Maekawa* — $27.9t_{msg}$
3. *BS* — $31t_{msg}$

In a ring with 21 processes, we can expect a similar ranking:

1. *RA* — $10t_{msg}$
2. *Maekawa* — $18.4t_{msg}$

3. $BS - 21t_{msg}$

We are now ready to add message processing time and contention to our analysis.

For the RA algorithm, a message will travel about half-way around the ring between critical section executions. For a 31-process ring, we can expect 15 hops between each execution of the critical section. Since each message is removed from an incoming communications path and placed onto an outgoing communications path at each hop, a message traveling 15 hops will be handled 30 times: once at the outgoing process, 28 times by the intervening CIPs and once at the destination process. This gives us an interval between critical section executions of $15t_{msg} + 30t_p$. With $t_{cs} = 1.0$, $t_{msg} = 0.1$ and $t_p = 0.005$, we obtain $S_{cs} = t_{cs} + 15t_{msg} + 30t_p = 2.65$ time units. We now compute

$$Q_{cs} = 27.226418 \approx 28$$

This leaves only 3 processes to send requests onto the ring when the system reaches steady state. The time around the ring (a message travels $N - 1$ hops before being removed from the ring) is $(N - 1)t_{msg} + 2(N - 1)t_p = 3.3$ time units. With $m = 3$ and $S_{path} = 3.3$, we can compute

$$Q_{path} = 1.029606$$

But this is the contention for the entire ring. If we disperse this queue length equally across all 30 processes, then the contention in the ring is nominal; for our purposes, this value is negligible and can be ignored. We can use our initial estimate of $15t_{msg} + 30t_p = 1.65$ as our result for the performance of the RA algorithm. We will find this same observation holds for all of our estimates for the ring topology. Table 3.5 and Table 3.6 catalog our results; they reveal that our estimates for the performance of the *RA* algorithm on a ring topology are well within 10% for all cases.

For the *BS* algorithm, there are N hops between each execution of the critical section. Since each message is handled by the CIP twice, the result is $Nt_{msg} + 2Nt_p$. For 31 processes, $t_{msg} = 0.1$ and $t_p = 0.005$, our computation is 3.41. With $t_{cs} = 1.0$, we let $S_{cs} = t_{cs} + 3.41 = 4.41$. We now compute

$$Q_{cs} = 28.732424 \approx 29$$

Again, contention on the ring is negligible. Therefore, we use our initial estimate of 3.41 time units as our result. But we again have the case where the process co-located with the controller will exchange messages with the controller in negligible time. We will employ the heuristic where our result is reduced by $\frac{1}{m}$ for a m-process system. For example, our result for $t_p = 0.005$ and $N = 31$ is 3.41. Reducing this value by $\frac{1}{31}$ gives us 3.3 time units. Actual simulation results show 3.299258 time units, for a relative error of 0.0%. Table 3.4 and Table 3.5 show that our estimates for the *BS* algorithm are consistently within 1.0% of the actual values.

For the *Maekawa* algorithm, the processes are not arranged in any special order around the ring in an attempt to optimize performance; the processes are still numbered from 1 to N in clockwise fashion. Because of this simple numbering scheme, we might expect that using the results that we derived for the *RA* and *BS* algorithms as the basis of our estimate will cause a distortion in our results — after all, some of the sets of size K within the *Maekawa* algorithm are contiguous on the ring, while others are spread throughout the ring. However, our results are pleasantly accurate.

For 31 processes, we compute mean time around the ring as 27.9 hops. Since each message is handled twice at each CIP (once to remove it from the

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	2.195716	2.2	.002
	.010	2.386511	2.4	.006
	.015	2.604817	2.6	.002
	.020	2.785099	2.8	.005
	.025	2.995982	3.0	.001
RA	.005	1.048771	1.1	.049
	.010	1.145060	1.2	.048
	.015	1.227219	1.3	.059
	.020	1.319470	1.4	.061
	.025	1.409020	1.5	.065
Mae	.005	1.912121	2.024	.059
	.010	2.111645	2.208	.046
	.015	2.292978	2.392	.043
	.020	2.634786	2.576	.022
	.025	2.775042	2.760	.005

Table 3.5: Application of the Model to 21 Processes in a Ring Topology

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	3.299258	3.3	.000
	.010	3.597543	3.6	.001
	.015	3.886130	3.9	.004
	.020	4.200229	4.2	.000
	.025	4.500290	4.5	.000
RA	.005	1.524705	1.65	.082
	.010	1.706341	1.80	.055
	.015	1.887260	1.95	.033
	.020	1.927918	2.10	.089
	.025	2.125856	2.25	.058
Mae	.005	2.925223	3.069	.049
	.010	3.326011	3.348	.007
	.015	3.593556	3.627	.009
	.020	3.897614	3.906	.002
	.025	4.158292	4.185	.006

Table 3.6: Application of the Model to 31 Processes in a Ring Topology

communications path and once to place it onto the outgoing communications path), we obtain $27.9t_{msg} + 55.8t_p$ time units as the mean time around the ring. With $t_{msg} = 0.1$ and $t_p = 0.005$, our result is 3.069. Letting $t_{cs} = 1.0$, we set $S_{cs} = 4.069$. We now compute:

$$Q_{cs} = 28.542397 \approx 29$$

Again, contention on the ring is negligible, and our result of 3.069 will serve as our estimate. Table 3.5 and Table 3.6 reveal that our estimates for the performance of the *Maekawa* algorithm on a ring topology are within 6.0% of the simulation values. The accuracy of our results is somewhat surprising, given that we derived them using the *RA* algorithm as a basis for the distributed portion of the algorithm.

3.5.3 Star Topology

Figure 3.10 shows a network with a star topology. There is a single full-duplex channel between the hub and each process in the network. Each process sends its messages to the hub; the hub then relays the message to the appropriate process. We assume that the system has multicasting capability, where a single message can be addressed to all or a subset of other processes. Hence, a process using the *RA* algorithm can send a single *request* message to the hub, and the hub will relay the message to the appropriate set of processes. Likewise, a process can send a single *release* message when it exits the critical section. We are now ready to derive our analytical results.

When a process in the *BS* algorithm exits the critical section, it sends a *release* message to the hub. The hub process is the controller, and it sends a *grant* message to the next process with priority to enter the critical section. For each execution of the critical section, the length of the chain of messages is two, and each message in the chain requires t_{msg} time to reach its destination. If message processing time is negligible, then the time between each execution of the critical section is $2t_{msg}$.

When a process in the *RA* algorithm exits the critical section, it sends a *reply* message to the hub. The hub then relays the message to all processes waiting for a reply from this process. For each execution of the critical section, the length of the chain of messages is one; however, each message in the chain requires $2t_{msg}$ to reach its destination. If message processing time is negligible, then the time between each execution of the critical section is identical to that of the *BS* algorithm.

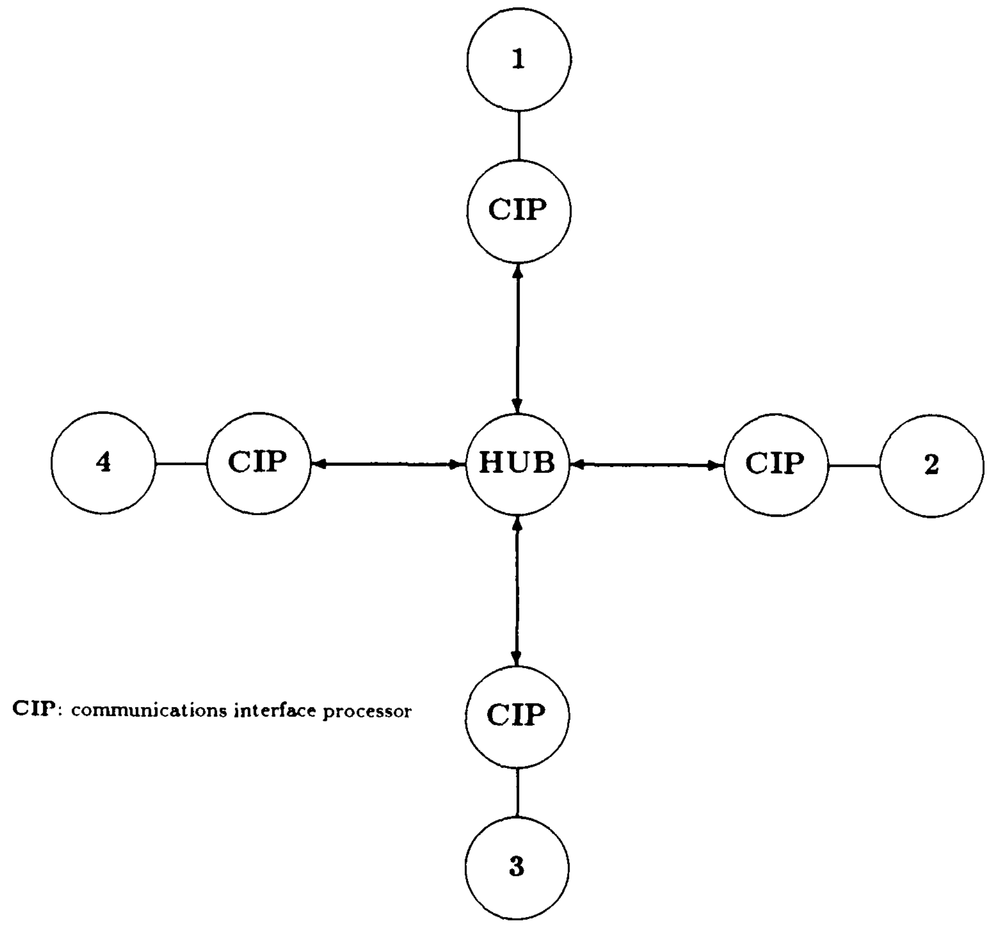


Figure 3.10: A Network with a Star Topology

For the *Maekawa* algorithm, each process is a controller. This characteristic of the algorithm prevents us from using the hub as a controller as in the *BS* algorithm. Rather, each controller process must send its messages across the hub. Therefore, when the behavior of the *Maekawa* algorithm mimics the centralized algorithm, each message requires $2t_{msg}$ to reach its destination. Since the length of the chain of messages is two in the centralized portion of the algorithm, the time between each execution of the critical section is $4t_{msg}$. When the *Maekawa* algorithm behaves in a distributed manner, it takes time equal to $2t_{msg}$ between each execution of the critical section. Using our probability analysis for the *Maekawa* algorithm, we can predict the time between each execution of the critical section to be:

$$\frac{K}{N}(2t_{msg}) + \frac{N-K}{N}(4t_{msg}) = \frac{4N-2K}{N}t_{msg}$$

On a star topology with 31 processes and negligible message processing time, we can expect the *BS*, *RA* and *Maekawa* algorithms to be ranked as follows:

1. *RA* and *BS* — $2t_{msg}$
2. *Maekawa* — $3.61t_{msg}$

On a star topology with 21 processes, we can expect a similar ranking:

1. *RA* and *BS* — $2t_{msg}$
2. *Maekawa* — $3.52t_{msg}$

We are now ready to explore performance with the addition of message processing time and contention at the hub.

For the *BS* algorithm, our results will be identical for the fully connected topology with one exception. Since the controller resides at the hub, there is no process co-located with the controller. For $m = 31$, $t_p = 0.005$ and $t_{msg} = 0.1$, we set $S_{cs} = 1.22$ and $S_{path} = 0.11$. We now derive the following results.

$$Q_{cs} = 22.803282 \approx 23$$

$$Q_{CIP} = 0.094040$$

The time between successive executions of the critical section is:

$$2t_{msg} + 4t_p + Q_{CIP}t_p = 0.220470$$

Simulation results show 0.218660 time units, for a relative error of 0.8%.

For $m = 21$, our computation becomes:

$$Q_{cs} = 12.803960 \approx 23$$

$$Q_{CIP} = 0.094040$$

The time between successive executions of the critical section is:

$$2t_{msg} + 4t_p + Q_{CIP}t_p = 0.220470$$

Simulation results show 0.218492 time units, for a relative error of 0.9%. The model continues to give good results on the star topology for the *BS* algorithm. However, when considering the *RA* and *Maekawa* algorithms, the complexity of the analysis greatly increases because of contention at the hub.

We can begin with the identical analysis that we used to determine the performance of the *RA* algorithm on a fully-connected topology. Our initial estimate of the time between critical section executions is $2t_{msg} + 3t_p + \frac{Q_{cs}}{2}t_p$. Now the challenge becomes to estimate contention at the hub. Since the hub can only process messages sequentially, a message will not be sent until the previous message has been fully transmitted. If the previous message is a request message, there may be up to $m - 1$ messages that must be transmitted until the subsequent message can be sent. A heuristic that we will use is the following. If a message arrives at the hub and "sees" half of a message in front of it in the queue according to our M/M/1/∞/m formula, then we will assume that half of the m outgoing messages must still be transmitted before the subsequent message can be sent. For example, if we assume that contention at the hub can be estimated using $S_{path} = 0.11$, then $Q_{CIP} = 0.094040$ for 8 processes. We will then assume that about one tenth of the $m - 1$ outgoing messages must still be transmitted before the subsequent message can be sent. In a fully-connected network and ring topology, we ignored the fact that for every *request* message sent, every process that is not waiting in the mutual exclusion queue will respond with a *reply* message. If n processes are available to send *request* messages into the system, then for each message sent there are $n - 1$ *reply* messages sent also. If there are m processes in the system and Q_{cs} is the length of the mutual exclusion queue, then $n = m - Q_{cs}$. Our measure for contention now becomes:

$$Q_{CIP} = Q_{CIP} m t_p + Q_{CIP}(m - Q_{cs} - 1)t_p$$

Since Q_{CIP} is defined recursively, it suggests the use of an iterative derivation. We can make an initial guess for Q_{CIP} and then iterate until the time between successive executions of the critical section converges to within some ε.

We encountered one problem using the iterative method. Since our M/M/1/∞/m formula uses only integers, some of our estimates did not converge. To get convergence, we interpolated the mean queue lengths of the truncated value and next higher integer. The following equations illustrate the heuristic. Let Q be the function that returns the queue length from our M/M/1/∞/m formula and let m be a real number.

```
a := trunc (m)
b := a + 1
q1 := Q (a)
q2 := Q (b)
interpolate := m - float (a)
q := interpolate * ( abs (q2 - q1) ) + q1
```

Using these heuristics, we now derive our results letting $\varepsilon = 0.001$. To see the power of the iterative heuristic, we will let $m = 31$ and $t_p = 0.005$. The following table shows the sequence of computations.

Algorithm	time between CS executions	$m - Q_{cs}$	Q_{CIP}
RA	0.220000	8.196718	.077500
	0.297500	7.707129	.156270
	0.307935	7.645636	.115672
	0.299002	7.698215	.108401
	0.297318	7.708208	.107168
	0.297030		

Simulation results show 0.285454 time units, for a relative error of 4.1%.

The *Maekawa* algorithm also requires the use of an iterative method to obtain reasonable estimates of its performance. Our equation to compute the interval between critical section executions is the same one that we have seen earlier for a fully-connected environment:

$$\frac{K}{N}(2t_{msg} + 3t_p) + \frac{N-K}{N}(4t_{msg} + 7t_p) + \frac{N-1}{N}(\frac{K-1}{2}t_p) + \frac{1}{N}t_p$$

Our formula to compute Q_{CIP} must also include the notion that the Maekawa algorithm sends $K - 1$ *failed* and *locked* messages across the hub for every *request* message. For Q_{CIP} we now have:

$$Q_{CIP} = \frac{N-1}{N}(Q_{CIP}(Kt_p)) + \frac{1}{N}(Q_{CIP}(2t_p)) + (Q_{CIP}(2Kt_p))$$

We again use the heuristic of interpolation to ensure that our results converge. With $m = 31$, $t_p = 0.005$ and $\varepsilon = 0.001$, we have the following sequence of computations.

Algorithm	time between CS executions	$m - Q_{cs}$	Q_{CIP}
MAE	0.397475	7.155766	.015025
	0.412500	7.079769	.090310
	0.412804		

Simulation results show 0.403856 time units between each execution of the critical section, for a relative error of 2.2%.

Table 3.7 and Table 3.8 show the results for the star topology without broadcast capability. For the *RA* algorithm, all of our results derived analytically are very close when t_p is small, gr[illegible]ually increasing to just under 25% when t_p reaches a maximum. For the *Maekawa* algorithm, all of our results are within 12% of the actual values.

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	.218492	.220470	.009
	.010	.239714	.241032	.005
	.015	.261559	.261457	.000
	.020	.279686	.282102	.009
	.025	.298896	.302829	.044
RA	.005	.251856	.266084	.056
	.010	.329886	.339806	.030
	.015	.439550	.421403	.041
	.020	.575410	.512504	.109
	.025	.720480	.611457	.151
Mae	.005	.393280	.398744	.014
	.010	.443247	.447277	.009
	.015	.505781	.494301	.023
	.020	.567644	.543798	.042
	.025	.638844	.594291	.070

Table 3.7: Application of the Model to 21 Processes in a Star Topology

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
BS	.005	.218660	.220470	.008
	.010	.239882	.241032	.005
	.015	.261687	.261457	.001
	.020	.278428	.282102	.013
	.025	.299407	.302829	.038
RA	.005	.285454	.297030	.040
	.010	.416801	.405176	.028
	.015	.597982	.525157	.122
	.020	.816122	.657547	.194
	.025	1.059925	.802100	.243
Mae	.005	.403856	.412804	.022
	.010	.465786	.464555	.003
	.015	.539534	.518356	.039
	.020	.634396	.573600	.096
	.025	.714092	.630713	.117

Table 3.8: Application of the Model to 31 Processes in a Star Topology

A reasonable question to ask is whether or not broadcast capability will enhance the efficiency of the *RA* and *Maekawa* algorithms to the extent that they may be competitive with the *BS* algorithm on a star topology. We can expect the *BS* algorithm to perform at the same level of efficiency whether it runs in an environment with broadcast capability or without it. However, we can expect improvement in the performance of the *RA* and *Maekawa* algorithms. The contention for the *RA* algorithm in an environment with broadcast capability was computed using the following equation.

$$Q_{CIP} = Q_{CIP}(2t_p + 2(m - Q_{cs} - 1)t_p)$$

The iterative method was then employed as shown previously. For the *Maekawa* algorithm, contention was computed using the following equation.

$$Q_{CIP} = Q_{CIP}(6t_p)$$

The equation is again compensated for the extra *failed* and *locked* messages sent across the hub.

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
RA	.005	.222334	.228840	.029
	.010	.252592	.260742	.032
	.015	.286326	.295784	.033
	.020	.333558	.334160	.002
	.025	.374652	.375494	.002
Mae	.005	.370148	.399854	.080
	.010	.419412	.438639	.046
	.015	.463123	.477647	.031
	.020	.510497	.516877	.012
	.025	.565150	.556322	.016

Table 3.9: Application of the Model to 21 Processes in a Star Topology with Broadcast

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
RA	.005	.221674	.228842	.032
	.010	.252690	.260742	.032
	.015	.285195	.295784	.037
	.020	.319373	.334160	.046
	.025	.362986	.375494	.034
Mae	.005	.392837	.399854	.018
	.010	.441730	.438639	.007
	.015	.494069	.477647	.033
	.020	.553554	.516877	.066
	.025	.592609	.556322	.060

Table 3.10: Application of the Model to 31 Processes in a Star Topology with Broadcast

Table 3.9 and Table 3.10 show the results for the star topology with broadcast capability. Results obtained for the broadcast environment are closer to the actual simulation values than the results shown for an environment without broadcast, indicating our approximations underestimate message contention in the network.

One interesting result of the simulations of the *RA* algorithm on a star topology is that the *RA* algorithm performed more efficiently with 31 processes than with 21 processes in a broadcast environment. This result seems to imply that the star topology does not become saturated until beyond 21 processes; that is, the star topology with 21 processes is not utilizing the critical section to its fullest potential. Our iterative method of estimating performance predicts that the *RA* algorithm performs with equal efficiency on both systems. This same observation did not hold for the *Maekawa* algorithm; our iterative method predicted, and simulation confirmed, that the *Maekawa* algorithm would be more efficient on a star topology with 21 processes than one with 31 processes. In any case, broadcast capability did not enhance the performance of either the *RA* or *Maekawa* algorithm sufficiently to make them more efficient than the *BS* algorithm. Although the performance of the *RA* algorithm is almost comparable to the performance of the *BS* algorithm in a broadcast environment, its performance degrades as message processing time increases.

3.6 Application of the Model to a Recently Published Algorithm

The usefulness of this model depends on whether or not it can provide insight into the performance of newly-conceived mutual exclusion algorithms. In February of this year, a tree-based algorithm for distributed mutual exclusion was published [Raym]. The algorithm employs a logical topology that corresponds to a tree. A privilege is passed from process to process. The process that holds the privilege can either enter the critical section or pass it to a neighbor process. Each root process of a subtree maintains a FIFO priority queue. When the root process receives the privilege, it enters the critical section if its own request is at the head of the queue; or alternately, it sends the privilege to the process whose request resides at the head of the queue. The author claims that the algorithm works best on a radiating star topology. To allow a comparison between this algorithm and those looked at in this dissertation, we have added a fifth subtree to the published network as shown in Figure 3.11; this gives the distributed system 21 processes. We will now examine how this new algorithm performs when overlayed on a fully-connected topology.

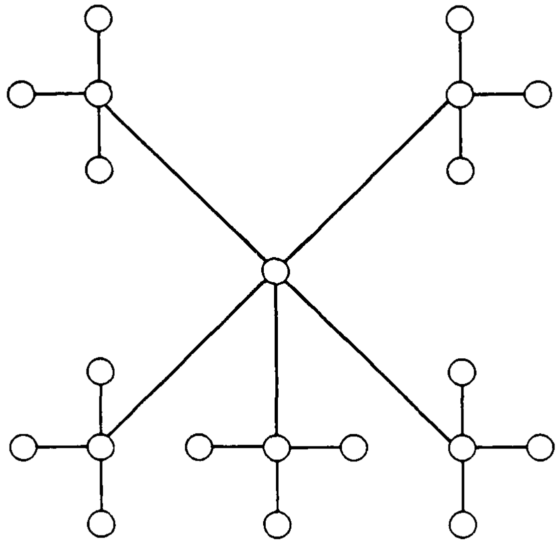


Figure 3.11: A Network with a Radiating Star Topology

The paper claims that there is no simple formula for the average distance between nodes. However, once we understand the algorithm and apply our understanding of the relationship between controller nodes and leaf nodes, a simple formula is available to estimate the performance of the algorithm.

Our first assumption is always that every process equally seeks access to the critical section. In order for some process at the root of a subtree (this process is a controller) to obtain the privilege for its subtree, it must make a request to the central node. The central node then passes the request on to the subtree that holds the privilege. When the subtree currently holding the privilege releases it, it is passed back to the central node for relay. Since every process seeks access to the critical section equally, the privilege could have been released by any node in the system with the exception of the subtree that has requested it. Since every subtree has 4 nodes, the token must be received from one of the other 17 nodes. Of these, 12 are leaf nodes 3 hops away; 4 are root processes 2 hops away; and 1 is the central node 1 hop away. This observation gives rise to an average distance that the privilege must travel before arriving at the root process of a subtree.

$$\frac{12}{17}(3t_{msg}) + \frac{4}{17}(2t_{msg}) + \frac{1}{17}(t_{msg}) = 2.647t_{msg}$$

Once the privilege is received by the controller of the subtree, there are two scenarios: the controller can accept the privilege and enter the critical section, or it can pass the privilege to one of its 3 leaves. This leads to the following observed average distance that the privilege will travel once it enters the subtree:

$$(.25)0t_{msg} + (.75)t_{msg} = .75t_{msg}$$

When the privilege arrives in the subtree, it will have traveled, on the average, $2.647t_{msg} + .75t_{msg} = 3.397t_{msg}$ time units until it is finally accepted by some process.

This algorithm then exhibits the characteristic that each process in the subtree eventually receives the privilege to enter the critical section before the privilege is passed back to the central node. The reason behind this property is that the privilege requires a substantial amount of time to migrate to the subtree. In fact, when the privilege is passed to the subtree, the central node will most likely pass a request for its return when it sends the privilege to the root process. This protocol prevents processes in the subtree from reusing the privilege more than once before the privilege is returned to the central node.

If a leaf node first uses the privilege, then there are two possible sequences that can take place: the privilege can circulate to the other 3 processes in time $t_{msg} + t_{msg} + 2t_{msg}$ or in time $2t_{msg} + 2t_{msg} + t_{msg}$. This gives an average circulation time of $1.5t_{msg}$ time units when the privilege begins at a leaf node.

If the root process first uses the privilege, then the privilege circulates to the other 3 processes in time $t_{msg} + 2t_{msg} + 2t_{msg}$. Since there are 3 leaf nodes and a single root node, we can expect the mean time required for the privilege to circulate through the subtree to be equal to $.75(.5(\frac{4}{3}t_{msg}) + .5(\frac{5}{3}t_{msg})) + .25(\frac{5}{3}t_{msg})$ which equals $1.125t_{msg} + .4167t_{msg} = 1.5417t_{msg}$.

We now observe that 4 nodes make up the subtree. One node will receive the privilege in a mean time of $3.397t_{msg}$ time. The other 3 will receive the privilege in a mean time of $1.5417t_{msg}$ time. We now have:

$$.25(3.397) + .75(1.5417) = 2.005525t_{msg}$$

There will be $2.005525t_{msg}$ between each execution of the critical section. Simulation results show $2.00868t_{msg}$. When $t_{msg} = 0.1$, the relative error is 0.2%.

When t_p is significant, we find that the mean message processing time is always $4.01105t_p$. This constant mean message processing time is perhaps the strength of the algorithm — regardless of the number of processes in the system, the mean message processing time remains constant when the system is in steady-state. This observation leads to the results shown in Table 3.11.

Algorithm	t_p	Simulation Results	Analytical Results	Relative Error
Raymond	.000	.200868	.200553	.002
	.005	.223877	.220055	.017
	.010	.249109	.240111	.036
	.015	.266022	.260166	.022
	.020	.287356	.280221	.025
	.025	.305844	.300276	.018

Table 3.11: Application of the Model to 21 Processes in a Radiating Star Topology

Chapter 4

Simulation Results for Fail-free Systems

In general, we find that fully distributed algorithms perform best in all systems with broadcast capability. Message processing time has little impact on their performance. The *Schneider* algorithm performs with comparable efficiency when message processing time is below 0.010. Its performance degrades as message processing time and the number of processes in the system increase. Broadcast capability makes negligible improvement in the performance of the *BS* and *Maekawa* algorithms, regardless of number of processes. It is interesting to note, however, the slight advantage in efficiency the *Maekawa* algorithm holds in a broadcast system with 7 processes when message processing time is small.

Without broadcast, the performance of the *Schneider* algorithm degrades rapidly as message processing time increases past 0.005. In general, the fully distributed algorithms perform best during periods of heavy contention when message processing time is at or below 0.015. During periods of moderate contention, the fully distributed algorithms perform best across the spectrum of message processing time. Finally, during periods of light contention, there is little to distinguish the performance of the five algorithms. We note that it is only in large systems with large message processing times where the *BS* and *Maekawa* alg[illegible]hms perform appreciably better than their fully distributed counterparts.

4.1 The Simulation System

POSSUM is the simulation system chosen for the implementation of the algorithms studied in this dissertation [Bru]. POSSUM is a process-oriented simulation system built on Modula-2 and lends itself to the simulation of distributed systems. The events of our simulations are the messages sent and received by system processes. Upon receipt of a message, a process executes a sequence of instructions peculiar to the type of message received. The process then suspends execution until the next message arrives. This message-passing paradigm is discussed extensively in [BCM, Mis86].

Our simulation results are for a fully-connected network. It is imperative that the reader be familiar with the model of the underlying system as outlined in Chapter 2 before attempting to understand the significance of the simulation parameters introduced in the following discussion.

4.2 Simulation Parameters Common to all Fail-free Environments

Every fail-free simulation has six parameters associated with it:

1. t_{cs} — the interval spent in the critical section: the time from when a process enters the critical section until the time it exits the critical section

2. t_{ncs} — the interval spent in the noncritica' section: the time from when a process begins execution until it submits a request to enter the critical section; or the time from when a process exits the critical section until it submits a request to reente. the critical section

3. t_{msg} — message transit time: the interval it takes a message to transit a communications path between two adjacent processes in the network

4. t_p — message processing time: the time from when a message is removed from its communications path and its address examined by a CIP, until it is either handed off to its parent applications process or it is ready to be sent onto another communications path; or conversely, the time from when a message address is examined and the message is subsequently placed onto its outgoing communications path, until the message departs the CIP

5. N — number of processes: the number of active processes in the system

6. *broadcast* — broadcast capability: in a system with broadcast capability. a CIP can send the same message simultaneously on any number of its outgoing channels

This number of parameters is too large for an effective simulation strategy. If we add the fact that we intend to simulate 5 algorithms, and then allow each of the first 5 parameters to assume 5 values, we can expect $2*5^6$ runs to complete the simulation. Clearly, it is essential that we hold some variables constant while studying the effects of changing other variables.

We must keep in mind the purpose of our simulations. Specifically, our goal is to rank the efficiency of these algorithms in fail-free and fail-prone environments. Our goal does not involve trying to find out how each algorithm performs in all ranges of all parameters. Consequently, we can make some simplifications without compromising our goal.

Our first assumption is to make time spent in the critical section constant and then express all other parameters in terms of that constant. We arbitrarily chose mean time spent in the critical section. t_{cs}. to be 1.0. This assumption is reasonable if every process executes an identical critical section.

Our second assumption is to make the message transit time between sender and receiver a constant. This assumption requires that all processes be equally spaced from one another. We do not compromise our ability to rank the overhead inherent to each algorithm by selecting a constant value. Message transit time between adjacent sender and receiver, t_{msg}, is equal to 0.1 for all simulations.

We can explore system performance under changing loads by varying the number of system processes, or we can explore system performance under changing loads by varying the value for t_{ncs}. If t_{ncs} is fixed, then we can increase the number of processes in the system to generate a corresponding increase in the frequency of requests for the critical section. In a similar fashion, we can decrease the number of processes in the system to cause a corresponding decrease in the frequency of requests for the critical section. If the number of processes is fixed, then decreasing t_{ncs} causes each process to request the critical section at shorter intervals; this increases the number of requests for the critical section. Likewise, increasing t_{ncs} causes each process to request the critical section less frequently. We have chosen to leave t_{ncs} at a fixed value and vary the number of processes. This allows a third assumption: time spent in the noncritical section, t_{ncs}, has an exponential distribution with a fixed mean value of 10.0. The exponential distribution captures the variations in computation times among the differing processes.

These three assumptions leave two paramters whose values we will vary over a selected range of values: N and t_p. The parameter *broadcast* will be either *true* or *false*.

The number of processes in the system, N, is varied to fit the number of processes required by the *Maekawa* algorithm to fulfill the non-null intersetion property: 7, 13, 21, and 31. These values provide two examples of systems where the critical section is not saturated (7 and 13), and two examples of systems where the critical section is saturated (21 and 31).

Finally, message processing time, t_p, is varied as a percentage of message transit time [LM]. It varies from 5% to 25% at 5% intervals.

There is no warm-up of the system before collecting statistical results. End effects are minimal for a fully-connected topology. In addition, statistics are only valid when all of them are initialized from the beginning of the simulation. For example, if the average number of processes waiting is reinitialized after a certain period of execution, processes that exit the *waiting* state will do so without ever having entered it during the simulation. No execution time is charged by the applications process to execute the statements called for by an incoming message; therefore, t_{cs} is never extended, regardless of the number of messages received by the process executing the critical section. Our reasoning to make this assumption valid requires that t_{cs} be of sufficient duration to make execution time for these statements negligible. Five hundred executions of the critical section is the basis for the results of each simulation.

All algorithms are implemented in the fail-free environment as published.

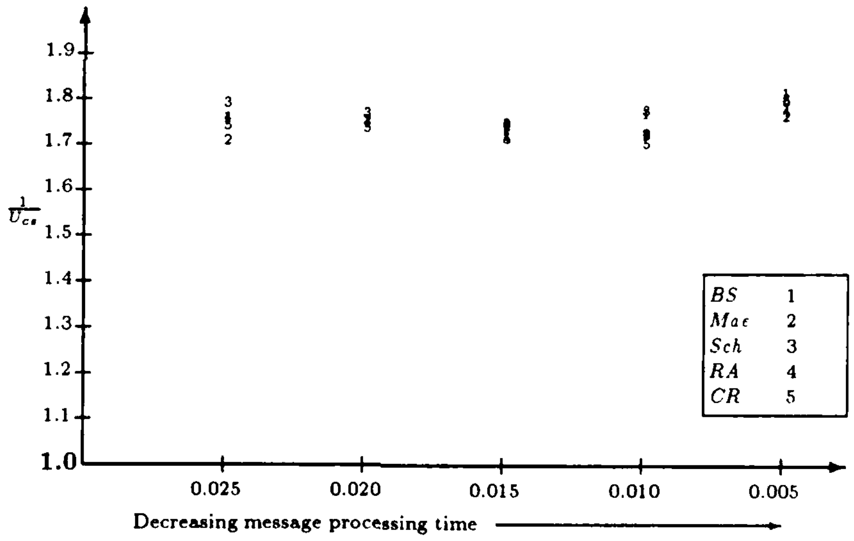
Interval	Distribution	Mean Value
t_{cs}	constant	1.0
t_{ncs}	exponential	10.0
t_{msg}	constant	0.1
t_p	constant	0.005–0.025
N	—	7,13,21,31
broadcast	—	True,False

Table 4.1: General Parameter Values for the Fail-free Environment

4.3 Simulation Results

4.3.1 Without Broadcast

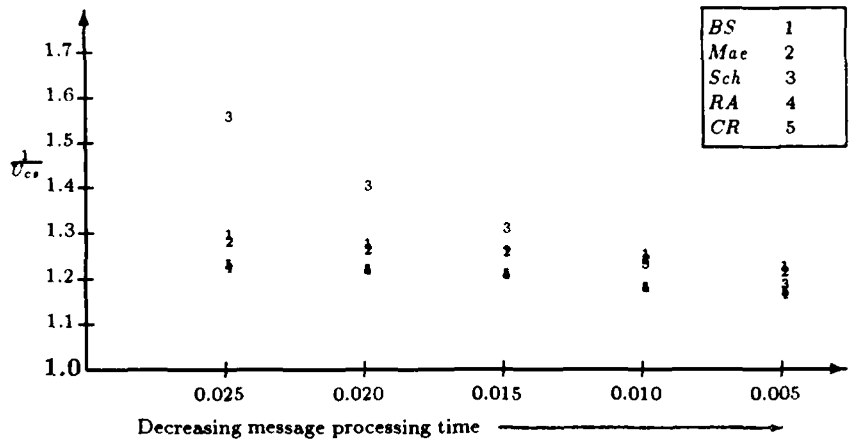
One of the most interesting results can be oberved in Figure 4.1. For almost every algorithm, performance is better as message processing time increases from 0.005 to 0.015; then performance degrades as message processing time increases from 0.015 to 0.025. When there is very light contention, it appears that larger message processing time makes it more likely that a process will be waiting in the queue when the critical section is released. This results in slightly more efficient use of the shared resource. As message processing time continues to increase, this advantage diminishes until the time a process must wait in the queue outweighs the advantage gained by the greater likelihood of being in the queue when the critical section is released.



Number of Processes: 7 Without Broadcast

Figure 4.1: Algorithms in Fail-free Environment

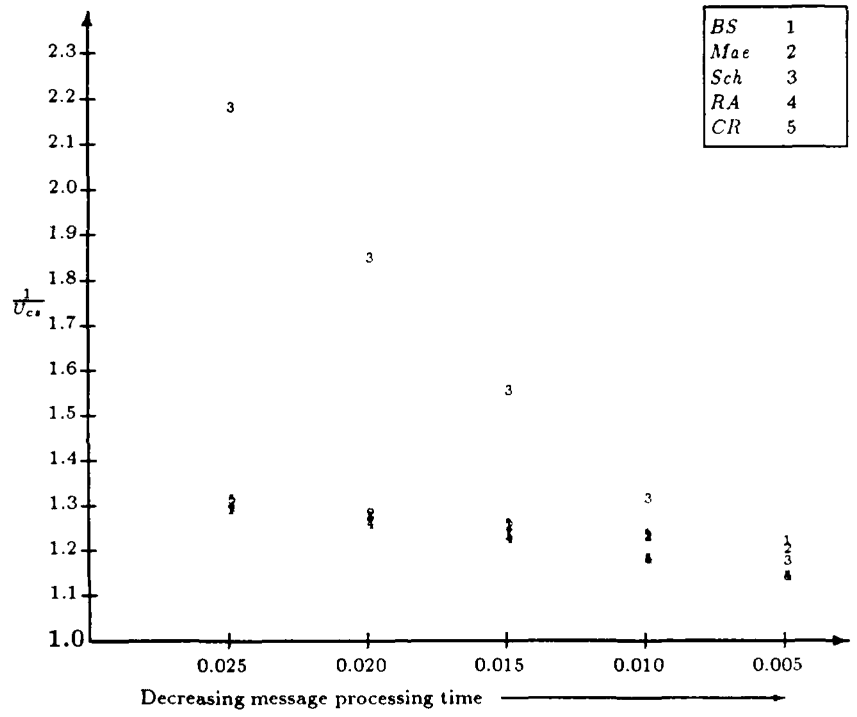
Let's look at the *Schneider* algorithm to help better understand this phenomenon. When message processing time is small, the message releasing the critical section will arrive at each process very quickly. The next process seeking entry to the critical section is more likely not to have made its request; therefore, when it seeks entry to the critical section, it must send a *request* message and receive an acknowledgement from every other process before it can enter the critical section. If, however, messages travel slower in the system, then it is more likely that a request will already have been submitted and a reply either received or in the process of being received by the time the message releasing the critical section arrives. Hence, it will be able to enter the critical section with little or no delay. The window where this advantage holds is very small. Once message processing time reaches a particular limit, the longer message transit time, which results from the increased message processing time, causes a longer mutual exclusion queue, greater response time and lower utilization of the resource.



Number of Processes: 13 Without Broadcast

Figure 4.2: Algorithms in Fail-free Environment

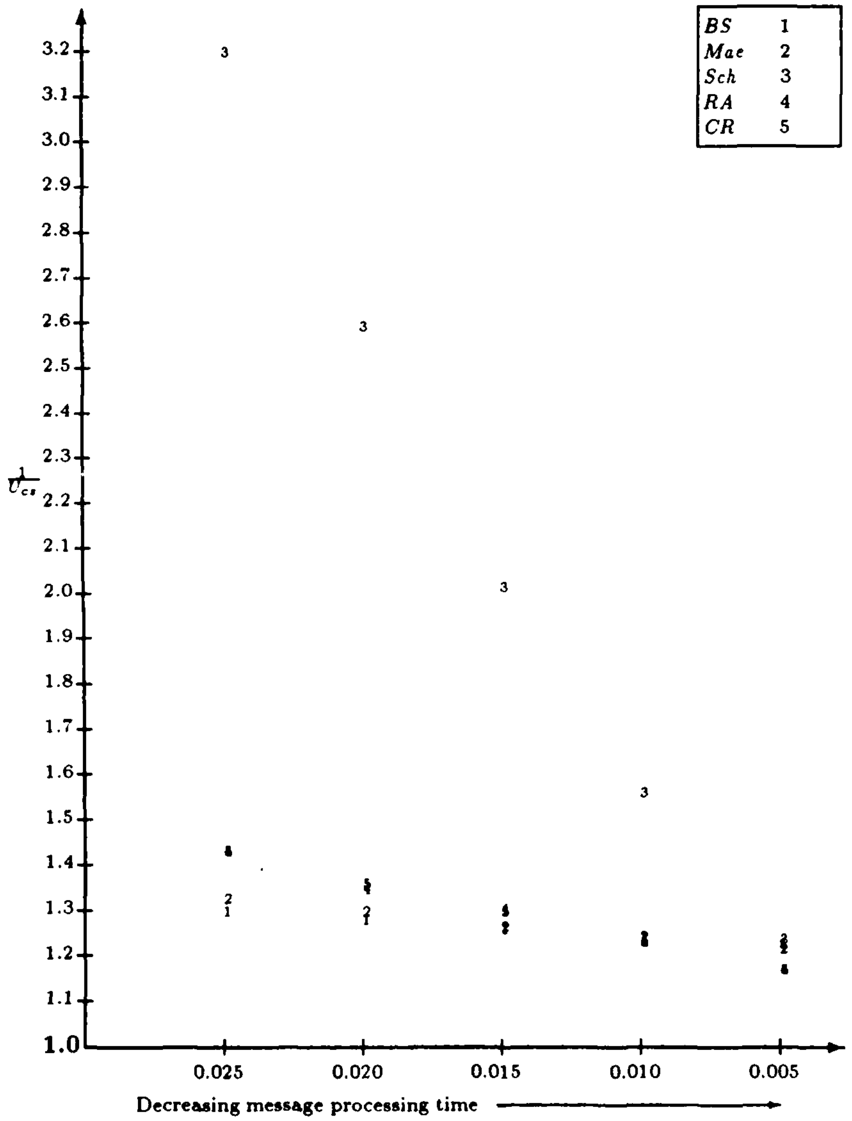
Figure 4.2 shows that with moderate contention, the *RA* and *CR* algorithms perform best. Message processing time still does not diminish their performance by creating significant contention at the CIP. The *Schneider* algorithm, on the other hand, suffers from the effects of contention when message processing time reaches 0.010, and its performance rapidly degrades thereafter. The performance of the *BS* algorithm is fairly constant, regardless of message processing time. The *Maekawa* algorithm performs slightly better than the *BS* algorithm across the spectrum of message processing time.



Number of Processes: 21 Without Broadcast

Figure 4.3: Algorithms in Fail-free Environment

For 21 processes, we find that all of the algorithms, with the exception of the *Schneider* algorithm, perform similarly until message processing time decreases to 0.015. The *RA* and *CR* algorithms perform best at lower message processing times; even the *Schneider* algorithm is almost comparable when message processing time is at 0.005. However, the *Schneider* algorithm rapidly bogs down as message processing time becomes significant. The number of messages to be processed and the resulting contention make the algorithm inefficient in comparison to the others.

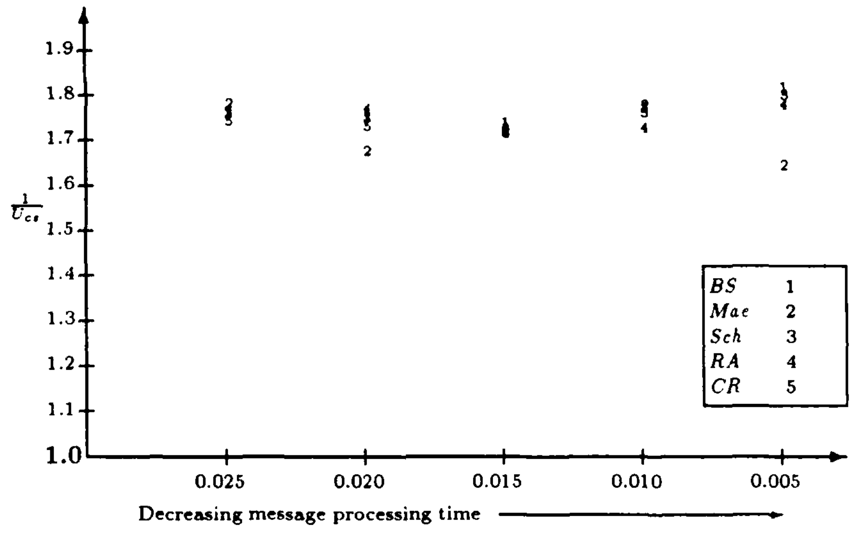


Number of Processes: 31 Without Broadcast

Figure 4.4: Algorithms in Fail-free Environment

We can see that with 31 processes, the *BS* and *Maekawa* algorithms perform best as message processing time increases through 0.010. The point where the *CR* and *RA* algorithms match and then exceed the performance of the *BS* and *Maekawa* algorithms occurs at and below 0.010 message processing time. The *Schneider* algorithm is uncompetitive at all but the lowest message processing time; even then, it is still the least efficient algorithm.

4.3.2 With Broadcast

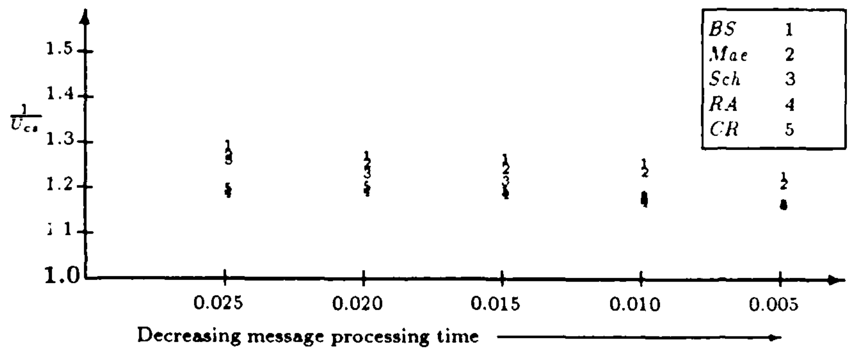


Number of Processes: 7 With Broadcast

Figure 4.5: Algorithms in Fail-free Environment

With broadcast, we note in Figure 4.5 the identical phenomenon as we did without broadcast. In a system with light contention, we can see that increasing message processing time increases efficiency until message processing time reaches about 0.015. As before, efficiency then decreases as message processing time continues to grow. For systems with light contention, this phenomenon seems to suggest that an artificial delay may actually improve utilization of the shared resource when message processing time is small.

With broadcast in a 13-process system, the *Schneider* algorithm performs better than both the *BS* and *Maekawa* algorithms. This is in contrast to systems without broadcast, where contention quickly degraded the performance of the *Schneider* algorithm. One interesting result to note is that the *RA* algorithm performs consistently better, albeit slightly, than the *CR* algorithm. This phenomenon did not hold for a 13-process system without broadcast. This inconsistency probably exists because the *CR* algorithm requires that a *request* message follow relinquishment of an implicit authorization by a process waiting entry to the critical section. If the *request* message were to be "piggy-backed" onto the *reply* message, we could expect the *CR* algorithm to perform at least as efficiently as the *RA* algorithm. However, the current algorithm calls for the sending of a *reply* message followed immediately by a *request* message. The consequence of this protocol is reduced efficiency.



Number of Processes: 13 With Broadcast

Figure 4.6: Algorithms in Fail-free Environment

Even with 21 processes, the *RA* and *CR* algorithms establish their superiority in a fail-free environment. The *Maekawa* algorithm performs slightly better than the *BS* algorithm across the spectrum. All of the algorithms provide performance similar to their performance in a system with 13 processes, with the exception of the *Schneider* algorithm. This stability in performance is a desirable characteristic for system designers interested in a consistent level of efficiency by the particular algorithm selected for implementation.

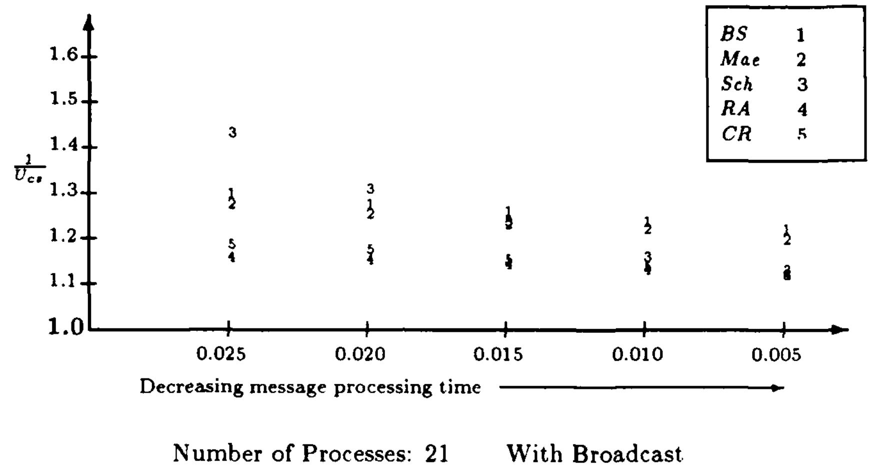
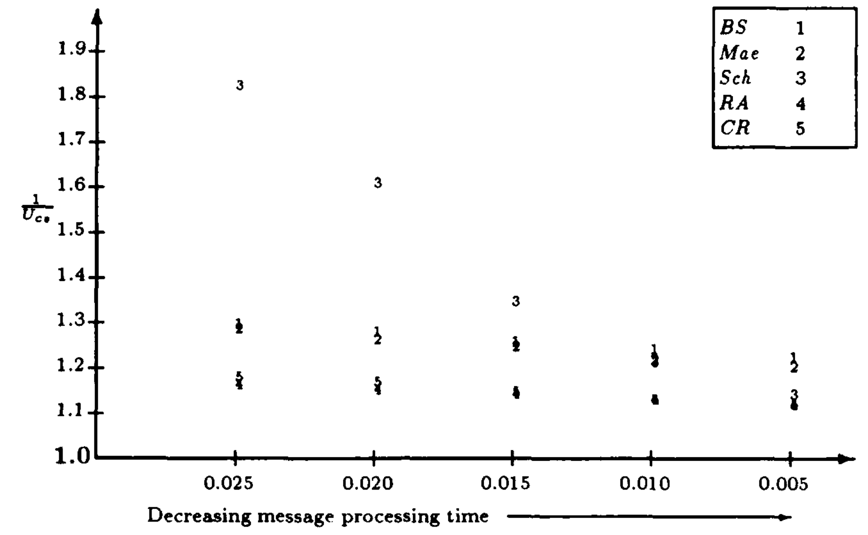


Figure 4.7: Algorithms in Fail-free Environment

Again, with the exception of the *Schneider* algorithm, we see very stable performance in all of the algorithms in systems with 31 processes; contention does not become a factor to any great degree for any of the algorithms. The *Schneider* algorithm performs well when message processing time is small; however, its performance quickly degrades as message processing time is increased.

It is clear that the fully distributed implementations enjoy an advantage throughout the range of message processing time in broadcast environments; this finding is in contrast to the loss of efficiency displayed in large systems with significant message processing times in systems without broadcast. As long as contention of messages does not become a significant factor, their shorter chains of messages give the distributed algorithms better performance.



Number of Processes: 31 With Broadcast

Figure 4.8: Algorithms in Fail-free Environment

Chapter 5

Simulation Results for Fail-prone Systems

In this chapter, we investigate the effects of process failure on the *BS*, *RA*, *CR* and *Schneider* algorithms. Specifically, we investigate process failure on a fully-connected topology. Our first task was to define timing intervals necessary for conducting leader election and probes of processes suspected of having failed. One of our more interesting results concerns the interval required to conduct a probe of a process suspected of having failed within a fully distributed algorithm. We discover that the interval necessary to determine whether or not a process has failed is identical for fail-prone environments with and without broadcast. As a result, our simulations reveal that systems having 7 and 13 processes and no broadcast capability perform with efficiency comparable to those systems with broadcast capability. Only as the number of processes increases to 21 do we see notable differences in efficiency between systems with and without broadcast.

We can also observe that because the *BS* algorithm must only compensate for failed controllers and processes which fail in the critical section, it performs well throughout all ranges of message processing time and number of processes in systems without broadcast. Even with broadcast, the *BS* algorithm performs more efficiently than its fully distributed counterparts until the number of processes in the system approaches 13. In contrast, because all failures must be compensated for in the *RA* and *CR* algorithms, performance is impeded due to any failure. Consequently, the fully distributed implementations spend a significant portion of their time managing failure.

We find that the *Schneider* algorithm cannot cope with frequent failure in small systems. Eventually, the situation develops where, due to a sequence of rapid failures, no process retains a complete message queue. The result is that the algorithm fails and must be restarted. For larger systems, we find the *Schneider* algorithm performing almost as efficiently in fail-prone environments as in fail-free ones. The reason for this is two-fold. First, with its detection of failure of processes that do not acknowledge messages, failed processes are generally removed from the mutual exclusion queue before they reach the head of the queue. The result is a throughput of processes into the critical section that approaches a fail-free implementation. Second, since each process sends $(N - 1)$ messages into the system for every message, the algorithm may at times transmit significantly fewer messages when processes have failed.

5.1 Compensating for Failure in Fail-prone Systems

In the presence of failures, an algorithm written for distributed systems must detect failure, compensate for it, and continue computation. When a failed process attempts to recover, the algorithm must provide a means for the process to reacquire local state consistent with the state of the system. Specifically, we study process failure. An algorithm must compensate for four types of process failure:

1. A process fails while executing outside the critical section.

2. A process fails while waiting for the critical section.

3. A process fails while executing within the critical section.

4. A process that plays a key role in the execution of the algorithm fails.

Failure is detected by an inquiring process sending an *are you alive* message to a process that appears to have failed. If a process suspected of failure is active, it acknowledges each *are you alive* message sent to it individually. This particular protocol of responding *I am alive* to each individual query has implications as to how quickly the inquiring process can assume that the non-responding process has failed. After a pre-determined interval has expired with no response, the inquiring process can assume that the suspect process is down. Each type of failure requires a unique response from the algorithm. This response varies according to whether an algorithm employs a centralized, hybrid or fully distributed implementation.

When a recovering process attempts to rejoin the algorithm, it must reacquire local state consistent with the execution of the algorithm. The recovery protocol used depends almost completely on the state maintained by each process. For example, within a centralized implementation, a recovering process is concerned only with its own state; consequently, its recovery protocol need only consist of determining which process is currently serving as controller. In contrast, within an implementation where each process maintains global state, a process must reacquire consistent global state. One method to reacquire global state involves using a "sponsor" to run a restart protocol. This is a difficult protocol to implement, because as soon as the sponsor process sends its global state to the recovering process, the global state is outdated.

The published papers for our algorithms discuss process failure in general terms. Many important details are missing. In order to run concrete simulations, we are forced to make assumptions about what length intervals may take, and we must then assign values to these assumptions. What values we assign have an impact on the results of our simulations, so they must be justified. Whether or not certain types of messages should have priorities over other types of messages is also an important issue. The following sections provide insight into the thinking that went into determining the lengths of the intervals used in our simulations; the justification for the numbers assigned to them; and the priorities assigned to certain types of messages.

Our results are presented for a fully-connected network where only applications processes can fail. Processes fail in a non-Byzantine manner, and the communications network never fails. Processes fail one at a time, and all processes are equally likely to fail. We assume that a failed process cannot send or receive messages, nor can it execute instructions. When a process fails, any local information pertaining to the use of the critical section is invalidated. Furthermore, a process that fails in the critical section has not effectively utilized it; consequently, the time spent in the critical section by that process is not counted toward utilization. When a failed process recovers, it recovers into the state *NCS*.

5.2 Simulation Parameters Common to all Fail-prone Environments

Fail-prone simulations add the following five parameters:

1. t_{fail} — the mean interval between process failures

2. t_{dead} — the time from when a process fails, until it starts to reboot

3. t_{reboot} — the time from when a process begins to reboot, until it is ready to restart

4. t_{delay} — the interval that a process will wait, until it inquires as to whether or not another process has failed

5. t_{max} — the interval that a process will wait, until it inquires as to whether or not the process in the critical section has failed

Our simulations are being conducted for the purpose of ranking the efficiency of our algorithms in a fail-prone environment. We are not attempting to find "hard numbers" for a wide range of values. Rather, we wish to test the ability of these algorithms to detect, cope with and recover from failure. This testing is done under conditions of frequent failure and rapid recovery.

We will explore the performance of our algorithms under loads of frequent failure. Our simulations have a mean time between exponential failures of 5.0 time units.

t_{dead} will have an exponential distribution and a mean value of 3.0 time units. The exponential distribution allows for a variation in the down time of system processes. t_{reboot} will be a constant time of 5.0 time units for every rebooting process; our assumption is that all processes execute an identical reboot routine. The sum of these two values equals $t_{recover}$, where $t_{recover}$ is the time from when a process fails, until it is ready to restart. During the interval $t_{recover}$, a process cannot send or receive messages. Since we are interested in testing the recovery mechanisms of the various algorithms, we have chosen values for t_{dead} and t_{reboot} that allow consistent exercise of the recovery protocol in the midst of consistent exercise of failure detection and compensation.

t_{delay} is the time from when a process sends a message, until it inquires as to whether or not a process which has not responded has failed. This time will be derived from our fail-free simulations; we will use twice the mean acknowledgement time of our fail-free simulations as the basis for causing a process waiting for a response to inquire as to whether another process is down. When the interval t_{delay} expires, a process will send an *are you alive* message to the process it suspects has failed.

t_{max} is similar to t_{delay}, except that it will be used in conjunction with receiving a response from the process currently executing the critical section. For example, if the controller in a centralized implementation has granted the critical section to a particular process, and a release of the critical section is not received within t_{max} time, then the controller inquires as to whether or not the process in the critical section has failed. When the interval t_{max} expires, the controller will send an *are you alive* message to the process it suspects has failed.

Interval	Distribution	Mean Value
t_{cs}	constant	1.0
t_{ncs}	exponential	10.0
t_{msg}	constant	0.1
t_p	constant	0.005–0.025
N	—	7,13,21,31
broadcast	—	True,False
t_{fail}	exponential	5.0
t_{dead}	exponential	3.0
t_{reboot}	constant	5.0
t_{delay}	constant	2∗(fail-free mean time)
t_{max}	—	2∗(fail-free mean time)+t_{cs}

Table 5.1: General Parameter Values for the Fail-prone Environment

All algorithms are implemented as published. Exceptions are explictly noted. Familiarity with the various algorithms is assumed in the following discussion.

5.3 BS Algorithm

The *BS* algorithm is the first published algorithm to give a viable election procedure and recovery mechanism for a centralized implementation of mutual exclusion. Specifically, the *BS* algorithm shows that election of a new controller in a fault-tolerant centralized implementation of mutual exclusion is a solvable problem. We implement the *BS* algorithm as published with one exception — the election algorithm used is the one developed by *Chang* [CG]. We have modified the *Chang* election algorithm as follows. If a process in the critical section is notified of an election, it immediately declares itself the winner of the election. This modification is consistent with the suggested election algorithm in [BS84].

5.3.1 Simulation Parameters Unique to BS

The *BS* algorithm adds the following three parameters to a fail-prone environment:

- $t_{c-probe}$ — the time from when the controller sends a message to the process executing in the critical section, until the controller can determine whether or not that process has failed
- t_{probe} — the time from when a process, p, sends a message to the controller, until p can determine whether or not the controller has failed
- t_{elect} — the time it takes to elect a new controller after it is first discovered that the controller has failed

$t_{c-probe}$ is a minimum interval based on the assumption that priority messages can be sent into the network to determine if a process has failed. Since our network is fully-connected, we can use the time to exchange a message between adjacent neighbors with no contention at the CIP as our value for $t_{c-probe}$:

$$\begin{aligned} t_{c-probe} &= t_p + t_{msg} + t_p + t_p + t_{msg} + t_p \\ &= 2t_{msg} + 4t_p \end{aligned}$$

Upon expiration of this interval, the controller is able to determine whether or not the process in the critical section has failed. We can also note that an *are you alive* message involves only two processes in the system: the inquiring process and its non-responding counterpart. The message exchange is transparent to the rest of the system.

The derivation of t_{probe} and t_{elect} are discussed in Section 5.3.3.

Interval	Broadcast	Distribution	Mean Value
$t_{c-probe}$	No	constant	$4t_p + 2t_{msg}$
	Yes	constant	$4t_p + 2t_{msg}$
t_{probe}	No	constant	$2Nt_p + 2t_{msg}$
	Yes	constant	$2Nt_p + 2t_{msg}$
t_{elect}	No	constant	$(3N-2)t_p + 2t_{msg}$
	Yes	constant	$(N+2)t_p + 2t_{msg}$

Table 5.2: Parameter Values for the Fail-prone Environment of BS

5.3.2 Message Priorities

When a process inquires whether or not the controller has failed, it sends a *probe* message to the controller. If a response is not received by t_{probe}, then the process knows that the controller has failed. The converse also holds; when the controller inquires whether or not the process executing the critical section has failed, the controller sends a *c-probe* message to the suspect process. If a response is not received by $t_{c-probe}$, then the controller knows that the process has failed. Since we have set $t_{c-probe}$ to be a minimum interval, *c-probe* messages must be of the highest priority. We note that *probe* messages can be assigned a lower priority than the *c-probe* message with no impact on the interval t_{probe}. The process in the critical section will not poll the controller while in the critical section and neither will it send a *probe* message to the controller until at least t_{max} time units after it exits the critical section; the implication is that our interval t_{probe} is large enough to compensate for the presence of a *c-probe* message.

When it is discovered that the controller has failed, electing a new controller must take priority over all other activity in the system. Since we want to elect the controller in minimum time, votes for the new controller, denoted as *election* messages, must be of the highest priority.

We must now resolve the conflict in terms of highest priority between *probe* messages and *election* messages. Only one type of message can be of the highest priority. When the controller initiates a *c-probe* message, there will obviously not be any election messages in the system, because the controller has not failed. In contrast, when a process inquires whether or not the controller has failed, it may be the case that an election has begun. The *probe* message is no longer of any consequence, however, because the inquiring process will know the controller has failed upon receipt of an *election* message. We also want the election to span the shortest possible interval, so that the mutual exclusion algorithm can restart. This leads us to choose the *election* message as having highest priority, the *c-probe* message second highest, the *probe* message third highest, all other mutual exclusion messages fourth highest, and the messages of the underlying computation the lowest priority. We now have the following priority ranking:

1. *election* messages
2. *c-probe* messages
3. *probe* messages
4. all other mutual exclusion messages
5. messages of the underlying computation

5.3.3 Specifying Intervals

We have defined the length of all intervals in our fail-prone simulation of the *BS* algorithm with the exception of $t_{c-probe}$ and t_{elect}. As previously mentioned, *election* messages have precedence over all other messages. This implies that the election takes place in minimum time. We must assume that every process might participate in the election of a new controller, including the failed controller that has just rejoined the algorithm. Given this assumption, we can establish some upper bounds on the time it takes for every process to send its vote to every other process. All of the following results hold for a fully-connected distributed system with N processes.

Theorem 5.1: An interval of sufficient length for leader election in a distributed system without broadcast, where t_{msg} is the longest message transmission time between any two adjacent processes, is at most

$$4(N-1)t_p + 2t_{msg}$$

where $t_p > 0$.

Proof: Each process votes exactly once per election by sending a single *election* message to every other process in the network. It will take a process $(N-1)t_p$ time to send $N-1$ messages. Each message requires at most t_{msg} time to transit the network from sending to receiving process. Since each process votes only once in an election, there can never be more than $N-1$ *election* messages received at any process. If a process receives an *election* message from every other process prior to sending its own vote, it will take $(N-1)t_p$ time to process those messages. Upon processing the last vote, it will send its own $N-1$ *election* messages in time $(N-1)t_p$. Again, each message transits the network in at most t_{msg} time. Finally, there can never be more than $N-2$ messages already in any message queue when the last message arrives. Therefore, no more than time $(N-1)t_p$ can pass to complete the election. We now have the following figures:

1. $(N-1)t_p$ is required for a process to vote.
2. t_{msg} time is required to send the *election* messages to the farthest process.
3. At most $(N-1)t_p$ time units can pass before all incoming votes are processed and the last process to receive its votes participates in the election.
4. The last process votes in $(N-1)t_p$ time.
5. t_{msg} time is required to send the messages to the farthest receiving process.
6. At most $(N-1)t_p$ time can pass to conclude the election after the last message arrives at its destination.

The resulting time is the sum of the individual times for these 6 events:

$$(N-1)t_p + t_{msg} + (N-1)t_p + (N-1)t_p + t_{msg} + (N-1)t_p$$
$$= 4(N-1)t_p + 2t_{msg}. \ \square$$

Because t_{msg} and t_p are constants in our simulations, we can derive a tighter bound for the time it takes to hold an election. We prove two lemmas to help us establish the bound. Our proofs assume a fully-connected network with N processes.

Lemma 5.1: If $t_{msg} > 0$ and $t_p = 0$, then the time it takes to hold an election is at most $2t_{msg}$.

Proof: If m processes discover simultaneously that the controller has failed, then m messages are simultaneously sent to each of the remaining $N-m$ processes in the network, and $m-1$ messages are sent to each of the m processes. Since $t_p = 0$, the messages will be transmitted and processed in time equal to t_{msg}. At this time, all remaining $N-m$ processes are aware of the election and $N-m$ messages are sent to each of the original m processes in the network, and $(N-m-1)$ messages are sent to each of the remaining $N-m$ processes. But again, $t_p = 0$, so we can expect the messages to be transmitted and processed in time equal to t_{msg}. The resulting time is $t_{msg} + t_{msg} = 2t_{msg}$. $\square$

Lemma 5.1 has a corollary that we will use in our later proofs.

Corollary 5.1: If $t_{msg} > 0$ and $t_p = 0$, then the time it takes for a process to send a message to one or more other processes and receive an acknowledgement from at least one other process is at most $2t_{msg}$.

Lemma 5.2: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes to hold an election in a distributed system without broadcast is at most $(3N-2)t_p$.

Proof: If m processes discover simultaneously that the controller has failed, then m messages are simultaneously sent to each of the remaining $N-m$ processes in the network, and $m-1$ messages are sent to each of the original m processes. Since $t_{msg} = 0$, each message arrives at its destination instantaneously. However, since $t_p > 0$, each group of m messages arrives at its destination precisely t_p apart. By definition of our underlying distributed system, each message is sent into the network from lowest numbered destination address to highest. Without loss of generality, assume that the highest numbered process is not one of the original m processes. After $(N-1)t_p$ time has passed, m messages will arrive at the highest numbered process. After at least mt_p has passed (additional messages may have arrived), the highest numbered process will participate in the election by sending its own $N-1$ messages in time equal to $(N-1)t_p$. After sending its vote, either (1) $N-1$ messages arrived before this process sent its vote, in which case the last message sent will arrive at its destination to an empty queue, and it will be processed in time equal to t_p; or (2) this process will have to process up to an additional $(N-m-1)$ messages after sending its own *election* messages. In case (2), the resulting time is $3(N-1)t_p$. In case (1), the resulting time is $2(N-1)t_p + (m+1)t_p$, where $m = (N-1)$. Case (1) now gives us time equal to at most $(3N-2)t_p$. □

Theorem 5.2: An interval of sufficient length for leader election in a distributed system without broadcast, with constant t_{msg} and t_p, is at most $(3N-2)t_p + 2t_{msg}$.

Proof: The election interval is maximized by assuming that one or more processes must be notified to participate in an election. By Lemma 5.1, the maximum time necessary to transmit a vote to another process and receive a reply in an election is $2t_{msg}$. In a similar fashion, the number of messages at a destination is at a maximum if all messages arrive at their destinations instantaneously. In this case, messages which might not otherwise contend with one another at the same site arrive simultaneously. By Lemma 5.2, the message processing time when messages arrive at their destinations instantaneously is at most $(3N-2)t_p$. Hence, the time to hold an election where t_{msg} and t_p are constant is at most $(3N-2)t_p + 2t_{msg}$. □

We now prove the following lemma to establish an upper bound for election with broadcast.

Lemma 5.3: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes to hold an election in a distributed system with broadcast is at most $(N+2)t_p$.

Proof: If m processes discover simultaneously that the controller has failed, then m messages are simultaneously sent to each of the remaining $N-m$ processes in the network. Since $t_{msg} = 0$, each message arrives at its destination instantaneously. With broadcast, all messages are sent in t_p time and every remaining process receives m messages at the same moment. m messages are processed in mt_p time, and $N-m$ messages are then broadcast to each of the original m processes in the network in t_p time. All messages again arrive instantaneously. After time $(N-m)t_p$, the last *election* message is processed. The resulting time is $t_p + mt_p + t_p + (N-m)t_p = (N+2)t_p$. □

We are now ready to prove an upper bound for controller election with broadcast.

Theorem 5.3: An interval of sufficient length for leader election in a distributed system with broadcast, with constant t_{msg} and t_p, is at most $(N+2)t_p + 2t_{msg}$.

Proof: Follows from Lemma 5.1 and Lemma 5.3. □

Lemma 5.4: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process in a distributed system, with or without broadcast, to send a *probe* message to the controller and then receive a reply, is at most $2Nt_p$.

Proof: Assume that all processes in the system elect to send a *probe* message to the controller. After time t_p, $N - 1$ *probe* messages will arrive at the controller. It will then take time $(N - 1)t_p$ to process those messages and $(N - 1)t_p$ to individually acknowledge them. The last message will arrive at its destination and be processed in time t_p. The resulting time is $t_p + (N - 1)t_p + (N - 1)t_p + t_p = 2Nt_p$. □

Theorem 5.4: An interval of sufficient length for probing the controller in a distributed sytem, with or without broadcast and with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.4. □

5.3.4 Conducting an Election

When a process determines that the controller has failed, it sends an *election* message to all other processes in the system. We assume that the *Chang* election algorithm is used by the *BS* algorithm [CG]. The *Chang* election algorithm requires that a process will not fail in the middle of casting its vote; i.e., every process must receive an identical set of votes to arrive at an identical conclusion as to which process has been elected controller. It's important to note that only processes waiting for the critical section have an interest in whether or not the controller has failed; a process not waiting for the critical section is not impacted by controller failure. Since it is desirable to limit message traffic in the network. our election need only involve those processes waiting for the critical section. A further advantage of this heuristic is efficiency. If a process waiting for the critical section becomes controller, it can grant the critical section to itself while it informs the other processes that it has become the new controller. If a process is currently executing the critical section, it immediately declares itself winner of the election upon receipt of an *election* message. It then notifies the rest of the system that it is the new controller. This technique also minimizes election time by completely eliminating the need to conduct an election. If the critical section is free when the controller fails, an election of some form must take place. Regardless of the election algorithm chosen, the physics of the problem constrains the optimum time in which an election can be performed. First, when a process determines that the controller has failed, it will send an *election* message to every other process in the system. Upon receipt of an *election* message, if a process waiting for the critical section has not already voted, it will send its vote in an *election* message to every other process in the system; otherwise, it saves the vote of the process which sent the message.

By virtue of the *Chang* election algorithm every process, which was active within t_{msg} of when the election began, knows what process has been elected as the new controller no later than t_{elect} time units after the first *election* message is sent. Upon expiration of t_{elect}, those processes seeking access to the critical section send their requests to the controller; the controller reconstructs the FIFO queue using the incoming requests. The new controller also sends a message throughout the system declaring its new status. This message is primarily for processes that attempt to rejoin the algorithm during an election; these processes may not receive the entire vote (part of it may have already been sent before they become active again), so this notification is essential.

It may be the case that a process attempts to rejoin the algorithm during an interval when the controller is dead and there are no processes waiting for the critical section; the message seeking the identity of the controller, sent by the restarting process to the other processes in the system, will be ignored. After t_{elect} time has passed, this process calls an election and notifies the rest of the system of its vote. This protocol implies that the process rejoining the algorithm participates in the election, even though it is not waiting for the critical section. This participation ensures the restarting process becomes controller if no other process participates in the election. If the critical section is occupied, the process in the critical section declares itself the winner of the election as previously discussed.

5.3.5 Miscellaneous Procedures

If a process fails while waiting for the critical section and then recovers before its failure is discovered by the controller, it will send a message requesting the identity of the controller. Upon receiving the message, the controller will remove the previous request of the process from the mutual exclusion queue.

When the controller determines that the process executing the critical section has failed, it generates a *release* message on behalf of the failed process and either sends a *grant* message to the process at the head of the mutual exclusion queue or marks the critical section as available if the queue is empty.

5.4 RA Algorithm

The published *RA* algorithm recommends using a sponsor process as the means of admitting restarting processes back into the execution of the algorithm. Once the process is readmitted, it must update its local sequence number using one of several methods described in the paper. Only after its local sequence number is updated can the process request mutual exclusion. This is a rather lazy approach to recovery. Instead, we want an aggressive recovery protocol. We have chosen to allow a restarting process to send a message to all other processes in the system. Those processes, which are active, will acknowledge the restarting process. After the expiration of a predetermined interval, the restarting process will have been added to the list of all active processes in the system; it will have identified all active processes; and it will have updated its local sequence number consistent with the execution of the algorithm. Advantages of this method include a process rejoining the mutual exclusion algorithm in minimum time and the ability of a process to be admitted to a system where all other processes have failed.

All other aspects of the *RA* algorithm are implemented as published.

5.4.1 Simulation Parameters Unique to RA

Interval	Broadcast	Distribution	Mean Value
t_{probe}	No	Constant	$2Nt_p + 2t_{msg}$
	Yes	Constant	$2Nt_p + 2t_{msg}$
$t_{restart}$	No	Constant	$(3N-2)t_p + 2t_{msg}$
	Yes	Constant	$2Nt_p + 2t_{msg}$

Table 5.3: Parameter Values for the Fail-prone Environment of *RA*

The *RA* algorithm adds the following two parameters to a fail-prone environment:

1. t_{probe} — the time from when a process, p, sends a message to all processes in its *Awaiting Reply Queue*, until p can determine which of these processes have failed

2. $t_{restart}$ — the interval it takes a process to rejoin the mutual exclusion algorithm after its first *restart* message is sent

The derivation of these intervals is presented in Section 5.4.3.

5.4.2 Message Priorities

When a process inquires whether or not one or more other processes have failed, it sends a *probe* message to each of those processes. Those processes which have not responded by t_{probe} have failed. Since we want to discover process failure in minimum time, *probe* messages must have the highest priority. If the process at the head of the mutual exclusion queue conducts a probe, it has authority to enter the critical section at the expiration of t_{probe}. This observation indicates the importance of *probe* messages to the continuation of the algorithm.

When a process reenters the system after having failed, it sends a *restart* message to all other processes in the system. Those processes responding by a certain time are active; those which have not responded when the interval expires are down. We would like to make this interval minimal for two reasons. First, it is possible for every process to restart in the same interval, which by necessity makes the delay caused by observing this interval unreasonably large in most cases. Second, the sooner a process rejoins the algorithm, the more frequently the critical section is requested.

We will show in Section 5.4.3 that the maximum interval of a probe is always less than or equal to the maximum interval required for a restart. We must consider that all processes may restart in the same interval. However, any process attempting to restart will not be sending *probe* messages. Therefore, we can simply assign higher priority to *probe* messages without compromising our restart interval. The intuition for this observation is that when the m^{th} process in the mutual exclusion queue conducts a probe, it only sends *probe* messages to those processes ahead of it in the queue; that is, the m^{th} process in the mutual exclusion queue will send m *probe* messages. Therefore, no more than $\frac{N(N-1)}{2}$ messages can ever be sent into the network during any probe. By contrast, up to $(N-1)^2$ *restart* messages can be sent if every process attempts to restart in the same interval. Since a process conducting a probe will not be restarting, and no process sends more *probe* messages than *restart* messages, granting *probe* messages priority over *restart* messages will not affect the duration of the restart. To ensure restart within some minimum time, the *restart* messages must by given priority over other mutual exclusion messages. We now have the following priority ranking:

1. *probe* messages
2. *restart* messages
3. all other mutual exclusion messages
4. messages of the underlying computation

This order of priority ensures minimum intervals for both failure detection and process restart.

5.4.3 Specifying Intervals

Under our discussion of simulation parameters unique to the *RA* algorithm, one can observe that $t_{restart}$ for the *RA* algorithm is equivalent to t_{elect} for the *BS* algorithm in an environment without broadcast. This is no accident: every process can restart or vote in the same interval of time. We will not prove the duration of the restart interval. Theorem 5.2 holds for both the *Chang* election protocol and the restart protocol described here in a distributed system without broadcast. With broadcast capability, the same message can be broadcast on $N - 1$ outgoing channels in time t_p. Our result for a broadcast capability, shown in Theorem 5.5, is therefore less than the result shown in Theorem 5.2. However, since each *restart* message must be acknowledged individually, our result is greater than that of Theorem 5.3.

Lemma 5.5: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process in a distributed system with broadcast to conduct a restart is at most $2Nt_p$.

Proof: Assume m processes restart simultaneously. m messages are sent to the remaining $N-m$ processes in the network in t_p time. m messages are processed in mt_p time, and replies are then sent in mt_p time. The last message crosses the network and is processed in t_p time. The result is $t_p+mt_p+mt_p+t_p = 2(m+1)t_p$ time units. This interval is at a maximum when $m = N - 1$. The result follows. □

Theorem 5.5: The time it takes for a process in a distributed system with broadcast to conduct a restart, with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.5. □

When conducting probes, we obtain a similar interval. To understand our subsequent proofs of the intervals necessary to conduct probes of other processes, we first provide an informal intuition of how we derived our results. In the *RA* algorithm, there is an implicit mutual exclusion queue. When there are no failures and the critical section is occupied, the process at the head of the mutual exclusion queue is waiting for a reply from exactly one process; that is, it is waiting for a reply from the process in the critical section. Similarly, the second process in the mutual exclusion queue is waiting for a reply from exactly two processes; that is, it is waiting for a reply from the process in the critical section and the process at the head of the queue. If every process is waiting in the mutual exclusion queue, then the last process is waiting for exactly $N - 1$ replies. If every process were to send a *probe* message to every process ahead of it in the queue, exactly $\frac{N(N-1)}{2}$ messages would be sent into the network. Our proofs will rely on the observation that the m^{th} process in the queue sends exactly m *probe* messages to those processes ahead of it in the queue and receives at most $(N - m - 1)$ *probe* messages from those processes behind it in the queue. By the same token, the m^{th} process receives at most m acknowledgements from the processes ahead of it in the queue and sends at most $(N - m - 1)$ acknowledgements to those processes behind it in the queue. The following results are presented for a fully-connected network with N processes.

Lemma 5.6: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process in a distributed system without broadcast to send a *probe* message to every process ahead of it in the mutual exclusion queue and then receive a reply from each of them, is at most $2Nt_p$.

Proof: Assume that $N - 1$ processes are waiting in the mutual exclusion queue and elect to simultaneously send *probe* messages. Without loss of generality, assume that processes are ordered in the queue such that process 1 is in the critical section, process 2 is at the head of the mutual exclusion queue, process 3 is second in the queue, and so on. Since messages are sent in order from lowest to highest, all processes waiting in the queue will send their first *probe* message to the process in the critical section. Furthermore, since a process will only send messages to its predecessors in the queue, the m^{th} process will send m *probe* messages in time mt_p. After additional t_p time has passed, a process, in the worst case, can expect to receive $(N-m-1)$ *probe* messages, where m is its position in the queue ($m = 0$ for the critical section process). These *probe* messages can be processed in $(N-m-1)t_p$ time. A process will then respond with $(N-m-1)$ acknowledgements in $(N-m-1)t_p$ time, the last of which is processed in time t_p. Finally, the m^{th} process will then process at most m acknowledgements to its *probe* messages. The resulting time is $mt_p + t_p + (N-m-1)t_p + (N-m-1)t_p + mt_p + t_p = 2Nt_p$. □

Theorem 5.6: An interval of sufficient length for the successful probing, by a process in a distributed system without broadcast, of all processes ahead of it in the mutual exclusion queue, with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.6. □

Lemma 5.7: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process in a distributed system with broadcast to send a *probe* message to every process ahead of it in the mutual exclusion queue and then receive a reply from each of them, is at most $2Nt_p$.

Proof: Assume that all processes in the mutual exclusion queue elect to send *probe* messages, and we assume the same ordering in the mutual exclusion queue as in Lemma 5.6. If $N-1$ *probe* messages are sent to the process in the critical section, the process in the critical section can still only process them in $(N-1)t_p$ time units, exactly t_p time units after the messages were broadcast. Then, since each message must be individually acknowledged, another $(N-1)t_p$ time units is required to acknowledge the *probe* messages. Finally, the last message is sent and then acknowledged in t_p time. The resulting time is $t_p + (N-1)t_p + (N-1)t_p + t_p = 2Nt_p$. □

It does not seem appropriate that an interval for a probe using broadcast should be no less than the interval for a probe where broadcast is not used. The intuition for this anomaly is that the concurrency available when messages are broadcast cannot be exploited. Incoming messages are still processed one at a time, and our protocol requires that each *probe* message be acknowledged with a separate response. Consequently, a process is unable to detect failure of the controller in less time in a distributed system with broadcast capability than one without it. This observation is crucial when we attempt to understand our results for the fail-prone environment later in this chapter.

Theorem 5.7: An interval of sufficient length for the successful probing, by a process in a distributed system with broadcast, of all processes ahead of it in the mutual exclusion queue, with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.7. □

5.4.4 Conducting a Restart

A restarting process sends a *restart* message to all processes in the system. After the interval $t_{restart}$ expires, the restarting process will have received a reply, which contains a sequence number, from all active processes in the system. Processes that did not reply by $t_{restart}$ have failed. The highest sequence number returned by the active processes will be adopted as the local sequence number of the restarting process. The restarting process enters the state *NCS* upon completion of its restart.

When a process restarts, it may send a *restart* message to an active process that was never aware of its failure. If this active process is awaiting a reply from the restarting process, it accepts the message as permission to enter the critical section. In any case, the active process acknowledges the restart.

5.4.5 Conducting a Probe

A process inquires as to whether or not there has been a failure in the system after t_{delay} expires. At this time, a *probe* message is sent to all processes with outstanding replies. After t_{probe} expires, the inquiring process knows which processes have failed.

Once the probe is completed, the process notifies all other processes in the system of the failures by means of a *FailDis* — failure discovered — message. This message contains the identifiers of all failed processes discovered by the process that originated the message. Upon receipt of a *FailDis* message, a process updates its list of active processes. It is possible to receive a *FailDis* message notifying a process of a particular failure, immediately following receipt of a *restart* message from the allegedly failed process. We assume that invalid *FailDis* messages are ignored through the observation of a grace period that begins upon receipt of a *restart* message. We do not attempt to define this interval. To simplify our simulation, we only respond to valid *FailDis* messages.

5.5 CR Algorithm

The published *CR* algorithm makes no mention of failure. Actually, it is an alleged optimization of the *RA* algorithm. Its difference is that a process, which never requests the critical section, is assumed to have given implicit permission to enter the critical section to all other processes in the system. Only when that process requests the critical section is the permission revoked.

All message priorities, intervals, restarts, and probes are identical to those of the *RA* algorithm. All other facets of the *CR* algorithm are implemented as published.

5.6 Schneider Algorithm

The *Schneider* algorithm is a distributed implementation, where every process maintains global state. Because each process maintains global state, every process knows precisely which process is in the critical section, which processes have or have not acknowledged messages and of which processes the mutual exclusion queue is comprised.

When a process has failed and attempts to rejoin the algorithm, the published algorithm recommends using a sponsor process to run a restart protocol for the restarting process. This protocol involves a significant amount of message forwarding. After completion of the remote start protocol, the process then runs a local start protocol. The completion of restart is delayed until certain further conditions are met. This protocol was designed before the *Chandy* and *Lamport* algorithm for distributed snapshots was discovered [CL]. We will use the idea of a sponsor process, but we will use what we know of distributed snapshots to help us derive a more efficient protocol. We will use the notion of a sponsor sending a copy of its queue to the restarting process, but only when the sponsor knows that the restarting process will have access to all messages from the time at which the queue is sent.

Failure detection has two components. First, when a message is received, each process eventually becomes aware of any process that fails to acknowledge the message. Processes that do not acknowledge messages within a predetermined interval are suspected of failure. The second component consists of the notion that we are familiar with from all of the previous algorithms: if the critical section is not released after a predetermined interval, the process in the critical section is suspected of failure.

5.6.1 Simulation Parameters Unique to Schneider

The *Schneider* algorithm adds the following three parameters to a fail-prone environment:

1. t_{probe} — the time from when a process, p, sends a message to the process in the critical section, until p can determine if the process executing the critical section has failed

2. $t_{ack-probe}$ — the time from when a process, p, sends a message to those processes that have not responded to a message, until p can determine which of those processes have failed

3. $t_{restart}$ — the interval after which a restarting process that has yet to receive a message queue knows that its sponsor process has failed

The derivation of these intervals is presented in Section 5.6.3.

Interval	Broadcast	Distribution	Mean Value
t_{probe}	No	Constant	$2Nt_p + 2t_{msg}$
	Yes	Constant	$2Nt_p + 2t_{msg}$
$t_{ack-probe}$	No	Constant	$4(N-1)t_p + 2t_{msg}$
	Yes	Constant	$(3N-2)t_p + 2t_{msg}$
$t_{restart}$	No	Constant	$O(N^2)t_p + 2t_{msg}$
	Yes	Constant	$O(N^2)t_p + 2t_{msg}$

Table 5.4: Parameter Values for the Fail-prone Environment of *Schneider*

5.6.2 Message Priorities

There are two types of *probe* messages in the *Schneider* algorithm: one for each component of failure detection. The first is the probe of the process in the critical section; the second is the probe of any process which does not acknowledge a message. Both components of failure detection are essential to the successful execution of the algorithm in a fail-prone environment. If all processes are waiting for the critical section, all messages are fully-acknowledged and the process in the critical section fails, then the algorithm will deadlock, unless the process in the critical section is probed. Similarly, if a process releases the critical section, but the request at the front of the mutual exclusion queue has never been fully acknowledged because of a failed process, then the algorithm again deadlocks unless the process that never acknowledged the request is probed. Hence, we must use probes of both types.

With its global state, each process is actually a controller and is capable of identifying the current process in the critical section. Using the same reasoning as we did in the *BS* algorithm, we can minimize t_{probe} by giving it the highest priority. If the process at the head of the mutual exclusion queue sends a *probe* message to the process in the critical section, and the process in the critical section has failed, then the process at the head of the queue can enter the critical section. This observation indicates the importance of the *probe* message to the overall performance of the algorithm.

When a process fails to receive an acknowledgement from another process, it sends an *ack-probe* message to the process in question. We want to minimize this interval for several reasons. Failure to receive an acknowledgement from the process in the critical section may indicate that it has failed, and the process at the head of the mutual exclusion queue can enter the critical section without waiting for the expiration of t_{max}. Identifying a failed process may also be all that is preventing the process at the head of the mutual exclusion queue from entering an unoccupied critical section. Once a process has failed, its messages can be removed from the queue maintained at each process; management of a smaller message queue is more efficient for both execution and memory purposes. Finally, and most importantly, failed processes will have their requests removed from the mutual exclusion queue before they reach the head of the queue and can slow down the algorithm.

Table 5.4 showed that the interval to conduct a probe of processes, which fail to acknowledge a message, always exceeds that of conducting a probe of the process in the critical section. Common sense tells us to give highest priority to the probe of the process in the critical section, while allowing probes of other processes to serve as an "early warning" fail detection system in the background. A process will not probe any process with more than one type of probe message. This protocol obviates the necessity of increasing the interval $t_{ack-probe}$ to account for *probe* messages that might be sent concurrently with *ack-probe* messages.

restart messages will be assigned lower priority than either type of *probe* message. Since a process sending *probe* messages will not send *restart* messages, the restart interval will not be compromised. We want a process to rejoin the algorithm as quickly as possible; therefore, we will assign a *restart* message a higher priority than the remaining mutual exclusion messages. We now have the following priority ranking:

1. *probe* messages
2. *ack-probe* messages
3. *restart* messages
4. all other mutual exclusion messages
5. messages of the underlying computation

5.6.3 Specifying Intervals

Because every process in the *Schneider* algorithm maintains global state, each process knows precisely which other process is in the critical section. A *probe* message can be sent directly to the process executing the critical section when failure is suspected. However, unlike the *BS* algorithm, which has only a single controller, it is possible that every active process in the system will probe the process in the critical section at the same time. Because every process maintains global state, every process will eventually probe the process in the critical section — not just those processes waiting in the mutual exclusion queue. This leads to the following observations about probing the process in the critical section within the *Schneider* algorithm. All results are for a fully-connected network with N processes.

Lemma 5.8: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process with global state, in a distributed system without broadcast, to send a *probe* message to the process in the critical section and then receive a reply, is at most $2Nt_p$.

Proof: Assume that all processes in the system elect to send a *probe* message to the process in the critical section. After time t_p, $N - 1$ *probe* messages will arrive at the process in the critical section. It will then take time $(N-1)t_p$ to process those messages and $(N-1)t_p$ to acknowledge them. The last message will arrive at its destination and be processed in time t_p. The resulting time is $t_p + (N-1)t_p + (N-1)t_p + t_p = 2Nt_p$. □

Theorem 5.8: An interval of sufficient length for probing the process in the critical section by a process with global state in a distributed system without broadcast, with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.8. □

Lemma 5.9: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process with global state, in a distributed system with broadcast, to send a *probe* message to the process in the critical section and then receive a reply, is at most $2Nt_p$.

Proof: Assume that all processes in the system elect to send a *probe* message to the process in the critical section. After time t_p, $N-1$ *probe* messages will arrive at the process in the critical section. It will then take time $(N-1)t_p$ to process those messages and $(N-1)t_p$ to acknowledge them. The last message will arrive at its destination and be processed in time t_p. The resulting time is $t_p + (N-1)t_p + (N-1)t_p + t_p = 2Nt_p$. □

Theorem 5.9: An interval of sufficient length for probing the process in the critical section by a process with global state in a distributed system with broadcast, with constant t_{msg} and t_p, is at most $2Nt_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.9. □

For conducting a probe of processes that fail to acknowledge a message, we must assume that the worst case probe will occur. In practice, this is an unlikely event, and we can expect probes to be conducted rather efficiently. However, it is theoretically possible that every process will attempt to probe every other process in the same interval. We will stipulate that a process that sends a *probe* message to the process in the critical section will not send an *ack-probe* message to the same process. Hence, *probe* messages will not extend the interval required to conduct a probe of processes that fail to acknowledge messages. These observations lead to the following results.

Lemma 5.10: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process with global state, in a distributed system without broadcast, to send a probe message to another process and then receive a reply, is at most $4(N-1)t_p$.

Proof: Assume that all processes in the system elect to send an *ack-probe* message to every other process in the same interval. It takes a process $(N-1)t_p$ time to send $N-1$ *ack-probe* messages. After sending its last *ack-probe* message, each process can expect to receive $N-1$ *ack-probe* messages from the other processes in the system, which require $(N-1)t_p$ time units to process; each process must then acknowledge each message in $(N-1)t_p$ time units. Finally, $N-1$ acknowledgements can be expected to its original messages, which require an additional $(N-1)t_p$ time units to process. The result is time equal to at most $4(N-1)t_p$. □

Theorem 5.10: An interval of sufficient length for probing an arbitrary process by a process with global state in a distributed system without broadcast, with constant t_{msg} and t_p, is at most $4(N-1)t_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.10. □

Lemma 5.11: If $t_{msg} = 0$ and $t_p > 0$, then the time it takes for a process with global state, in a distributed system with broadcast, to send a probe message to another process and then receive a reply, is at most $(3N-2)t_p$.

Proof: Assume that all processes in the system elect to send an *ack-probe* message to every other process in the same interval. It takes a process with broadcast capability t_p time to send $N-1$ *ack-probe* messages. Each process can expect to receive $N-1$ *ack-probe* messages from the other processes in the system, which require $(N-1)t_p$ time units to process; each process must then acknowledge each message in $(N-1)t_p$ time units. Finally, $N-1$ acknowledgements can be expected to its original messages, which require an additional $(N-1)t_p$ time units to process. The result is time equal to at most $t_p + 3(N-1)t_p = (3N-2)t_p$. □

Theorem 5.11: An interval of sufficient length for probing an arbitrary process by a process with global state in a distributed system with broadcast, with constant t_{msg} and t_p, is at most $(3N-2)t_p + 2t_{msg}$.

Proof: Follows from Corollary 5.1 and Lemma 5.11. □

The time to restart is theoretically at most $O(N^2)$, because every process (except one – there must be at least one process acting as sponsor) may attempt a restart in the same interval. In this case, every restarting process will send a *restart* message to every other process, which must be acknowledged by sending a message to every other process. Even if broadcast is available, a process may have at most $(N-1)^2$ incoming acknowledgements. Because of the potentially extended length of this interval, we have chosen to use a different protocol than that which we might otherwise have chosen if the interval was of a more reasonable duration. A restarting process will send an *ack-probe* message to its sponsor process t_{max} time units after it receives the first acknowledgement to its restart message. It cannot set the timer sooner, because it is only after receiving an acknowledgement that it knows which process is serving as its sponsor; this particular idea of sponsors is discussed in more detail in Section 5.6.5. At the expiration of $t_{ack-probe}$, the restarting process knows whether or not its sponsor process has failed. If the sponsor is still active, the restarting process will wait t_{max} before probing the sponsor again.

5.6.4 Conducting a Probe

A process conducts a probe of the process executing the critical section in an identical fashion to that of the controller in the *BS* algorithm. Conducting a probe of processes that fail to acknowledge messages, however, is different for a process that maintains global state.

Because a process maintains global state, it knows precisely which processes have failed to respond to a message. It is not required to send a number of messages "into the blind" and await a reply. Rather, it can send *are you alive* messages to specific processes that have not answered. This ability to identify the processes suspected of failure is in contrast to the *RA* algorithm, which must conduct probes without knowledge as to which processes may have failed.

5.6.5 Conducting a Restart

The restarting process sends a *restart* message onto every outgoing communications path. We assume that there is an existent mechanism for determining an active sponsor for the restarting process. Since each active process maintains global state, we do not believe this assumption is unreasonable. For example, each process may be identified with one other process that it will sponsor for restart. When that process fails, the sponsor process assumes responsibility for sponsoring any restarts that the failed process was previously responsible for. This responsibility is maintained by the sponsor process until it sends its message queue to the previous sponsor process after it restarts. At that time, the restarting process once again assumes its responsibilities as a sponsor. The sponsor process for any restarting process is therefore known throughout the system.

Each process, upon receiving the *restart* message, acknowledges it by sending an acknowledgement to every other process. Once the *restart* message is fully acknowledged at the sponsor process, the sponsor sends a copy of its message queue to the restarting process. This implies that all messages received from each process, up to and including the acknowledgement, are part of the message queue sent by the sponsor process.

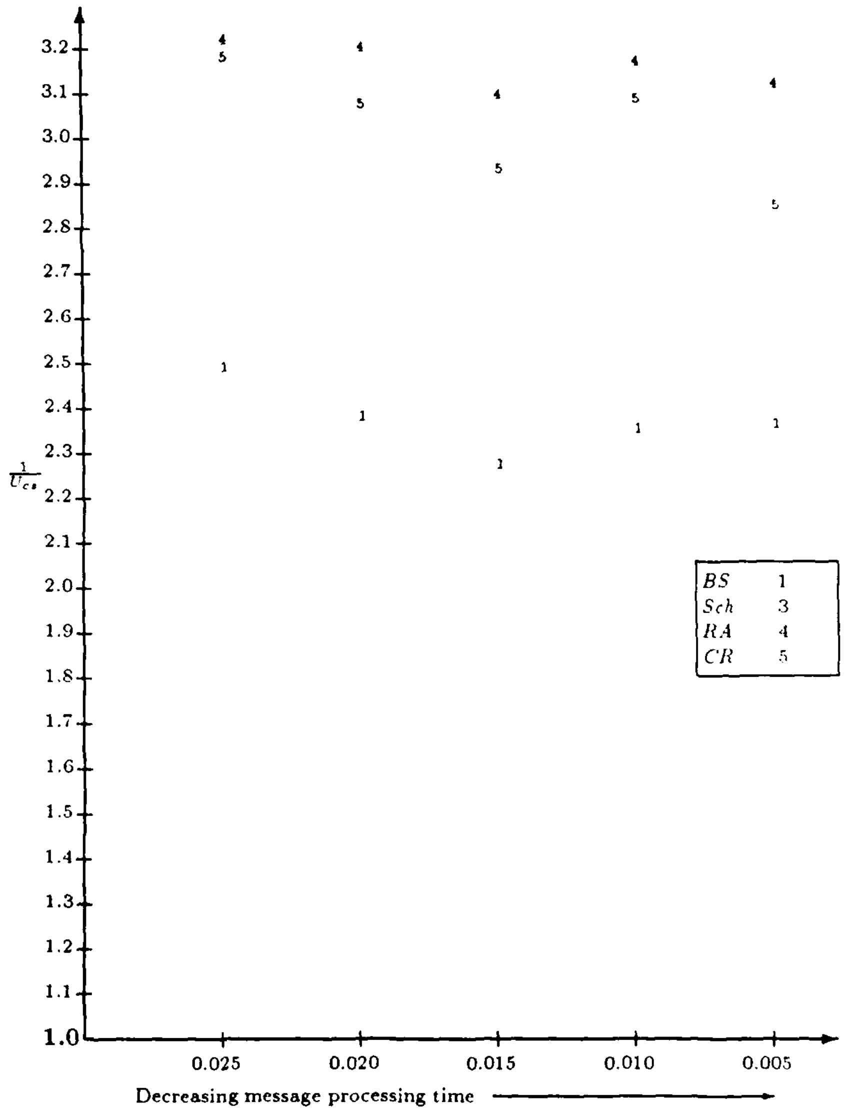
For its own part, the restarting process ignores all messages that it receives from any process prior to the acknowledgement of its own *restart* message. This action is possible, because the message queue it receives from its sponsor will contain all of these messages. Upon receipt of an acknowledgement of its *restart* message from a particular process, the restarting process begins to save and acknowledge all subsequent messages from that process. When its *restart* message is fully acknowledged, it will be saving and acknowledging all incoming messages. When the message queue finally arrives from its sponsor, the restarting process can update it with the messages it has been saving, disposing of any duplicates. Duplicate messages may be in the queue, because messages may have been received by the sponsor process from other processes that acknowledged the *restart* message and then sent subsequent messages to the sponsor before the *restart* message became fully acknowledged at the sponsor process. Once the message queue is updated, the restarting process can rejoin the algorithm.

The restarting process will delay for an interval equal to t_{max}. If a queue has not been received at the expiration of t_{max}, then the sponsor process is probed. If the sponsor process acknowledges the probe, the restarting process delays for an interval equal to t_{max} before probing the sponsor again. If the sponsor process has failed, the restarting process will send another *restart* message into the system. It is possible that a restarting process will never finish its restart using this protocol. In fact, in small systems with frequent failure, this protocol can break down completely, because it may be the case that no process has global state; much like deadlock, every process relies on some other process to send it the queue, but no other process has the queue. In this case, the entire algorithm has to be shut down and restarted.

5.7 Simulation Results

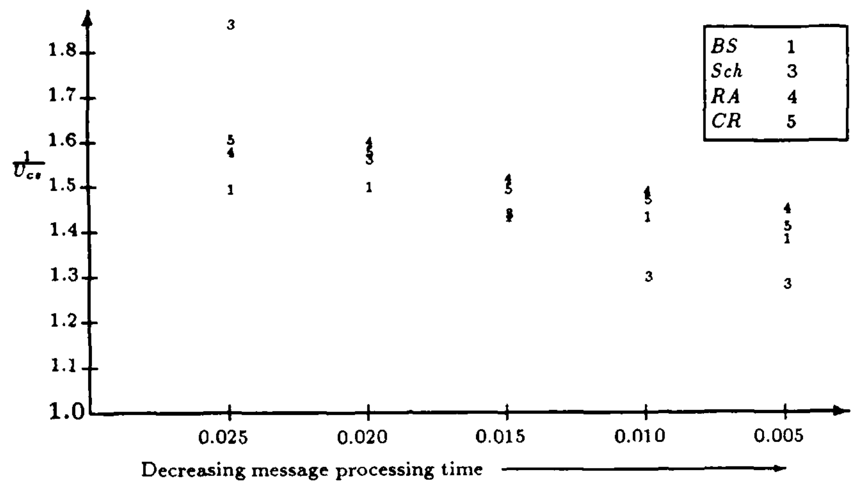
5.7.1 Without Broadcast

The *BS* algorithm has the property that failures of processes executing the non-critical section are transparent to the execution of the algorithm. The only failures that the algorithm must cope with in every case are controller failures — and elections can be conducted efficiently in small systems. If a process fails while waiting entry to the critical section, it may restart before the system is aware of its failure — in which case its *restart* message is used by the controller to remove its request seeking access to the critical section from the mutual exclusion queue. These helpful properties are in contrast to the fully distributed implementations, which must cope with all process failures in the system. We saw in Chapter 4 that all of the algorithms perform comparably inefficiently in small systems with 7 processes. When frequent process failure is a factor, we can see that the properties just noted play an important role in determining which algorithm is most efficient. Since the *BS* algorithm must only compensate for controller failures and failures of processes seeking access to the critical section, it executes relatively unencumbered by frequent failure. In contrast, it appears that compensation for failure occurs before almost every execution of the critical section in fully distributed implementations. Consequently, the *BS* algorithm runs most efficiently in systems with frequent failure and light contention for the critical section. The *RA* and *CR* algorithms exhibit substantially less efficiency. The *Schneider* algorithm's performance is not indicated on the graph with 7 processes, because it eventually shuts down due to the lack of any process holding a current message queue in the presence of frequent failure. Without a valid message queue held by any process in the system, there is no sponsor process available for restarting failed processes. In any case, it is evident that frequent process failure significantly affects the performance of all of the algorithms in small systems. The *Schneider* algorithm has points missing from other graphs when message processing time is large. These points are well off the graph and are not included.



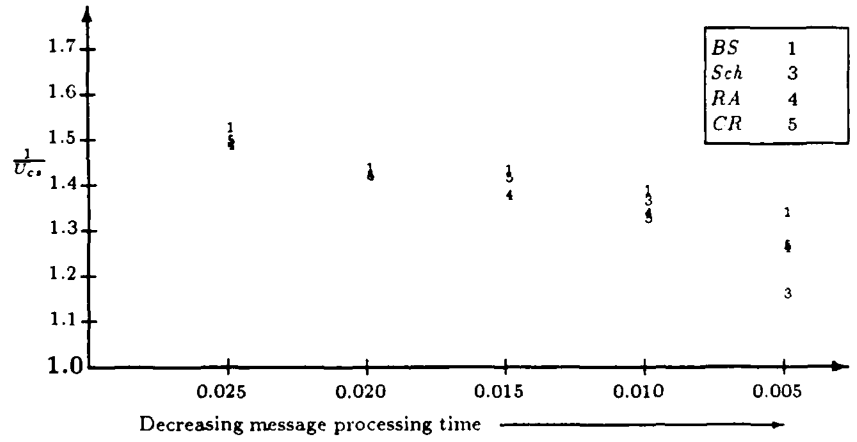
Number of Processes: 7 Without Broadcast

Figure 5.1: Algorithms in Fail-prone Environment



Number of Processes: 13 Without Broadcast

Figure 5.2: Algorithms in Fail-prone Environment

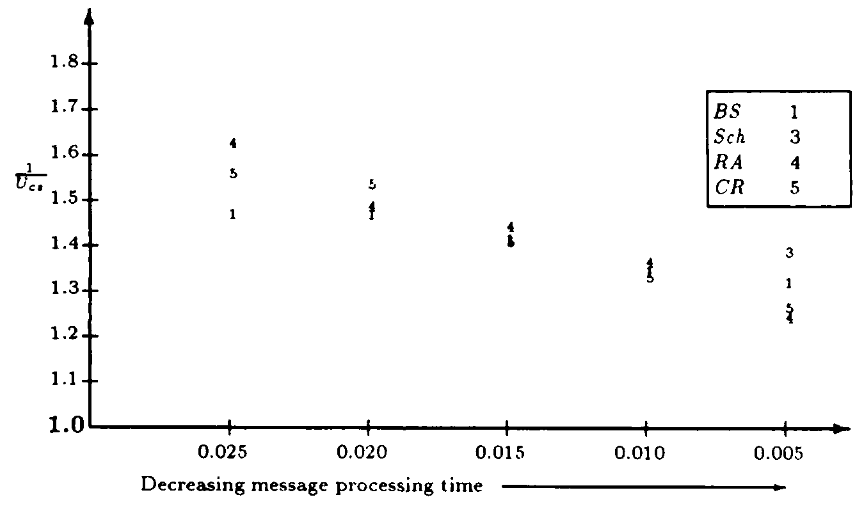


Number of Processes: 21 Without Broadcast

Figure 5.3: Algorithms in Fail-prone Environment

The *BS* algorithm still dominates its fully distributed counterparts when the number of processes is increased to 13. In fact, even when the number of processes is increased to 21, the *BS* algorithm performs most efficiently until message processing time decreases to 0.015. At that critical point, the cost associated with transmitting the extra message in the *BS* algorithm from the controller to the next process awaiting entry to the critical section exceeds the cost associated with propogating more messages and compensating for failure. The *Schneider* algorithm with its two levels of failure detection is most efficient with 13 and 21 processes and small message processing times.

For 31 processes, the *BS* algorithm performs as efficiently or more efficiently than its fully distributed counterparts across the spectrum of message processing time. Only when message processing time decreases to 0.005 do the *RA* and *CR* algorithms slightly exceed the efficiency of of the *BS* algorithm.



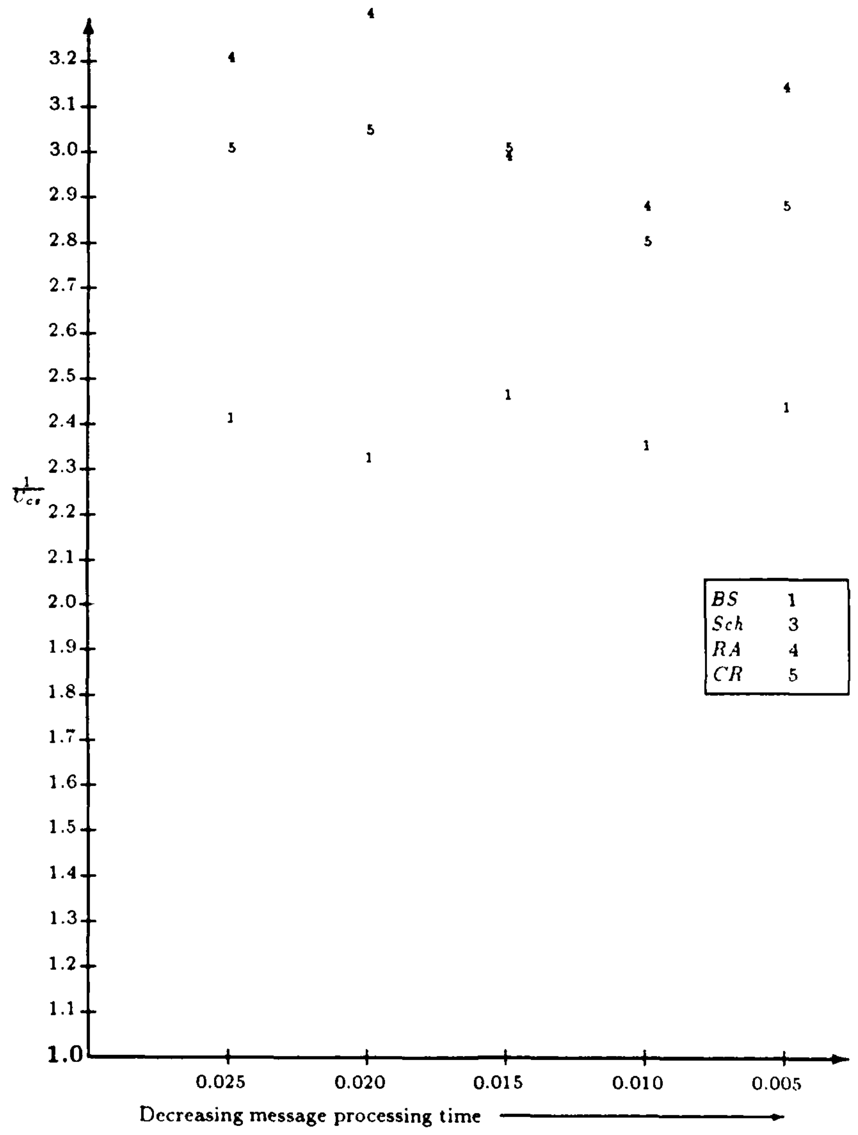
Number of Processes: 31 Without Broadcast

Figure 5.4: Algorithms in Fail-prone Environment

5.7.2 With Broadcast

The same analysis for the small systems with light contention and frequent failure in a no-broadcast environment holds for the environment with broadcast — the *BS* algorithm continues to execute most efficiently.

We can note that broadcast capability provides a forum for conducting efficient elections in the centralized implementation of the *BS* algorithm. Since elections are common is small systems with frequent failure and rapid recovery, the efficiency of the *BS* algorithm is not surprising.



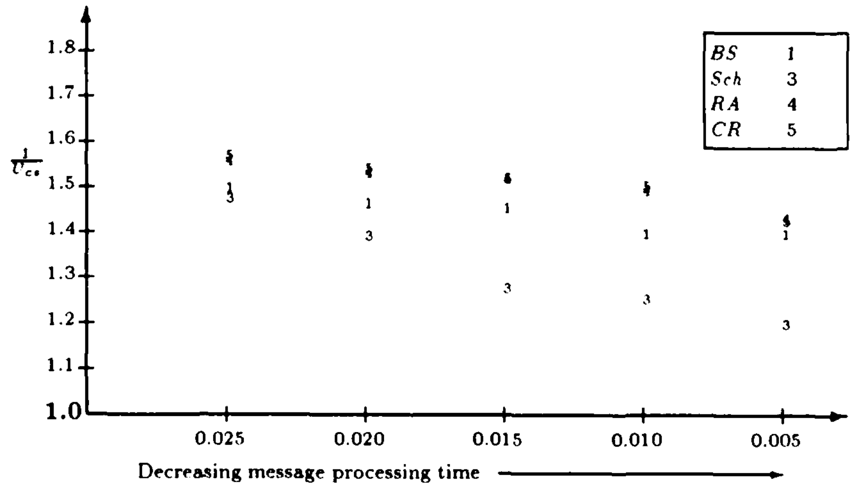
Number of Processes: 7 With Broadcast

Figure 5.5: Algorithms in Fail-prone Environment

As noted previously, broadcast does not enhance the detection of process failure in fully distributed implementations. Chapter 4 revealed that broadcast capability does little to improve the efficiency of small systems. Consequently, we can expect the *CR* and *RA* algorithms to perform poorly in small systems with frequent failure, as failure detection seems to dominate the activity of the algorithm.

We again note that the *Schneider* algorithm self-destructs in a system with only 7 processes, as the queue is eventually lost through a series of rapid failures.

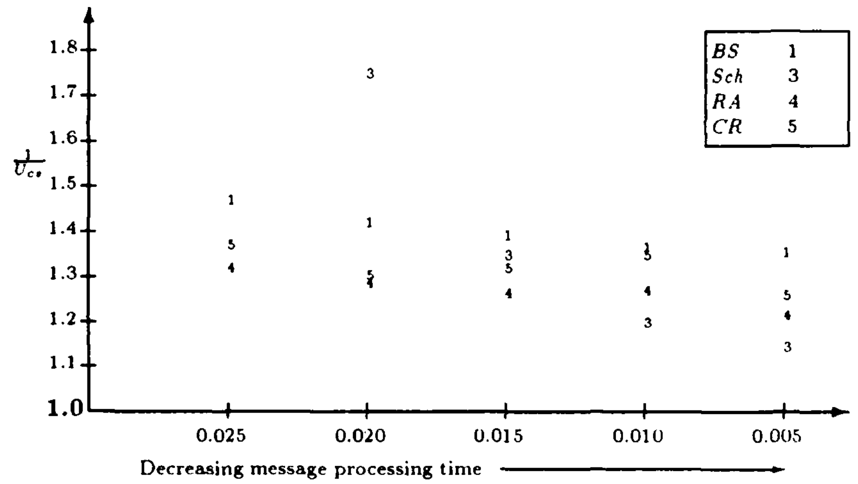
Unexpectedly, the *BS* algorithm continues to dominate its distributed counterparts when the number of processes is increased to 13. The moderate contention for the critical section by the *RA* algorithm and the advantages inherent in broadcast systems are still insufficient to overcome the cost of managing failure. The big surprise, however, is the performance of the *Schneider* algorithm, which is the most efficient algorithm across the spectrum of message processing time for 13 processes.



Number of Processes: 13 With Broadcast

Figure 5.6: Algorithms in Fail-prone Environment

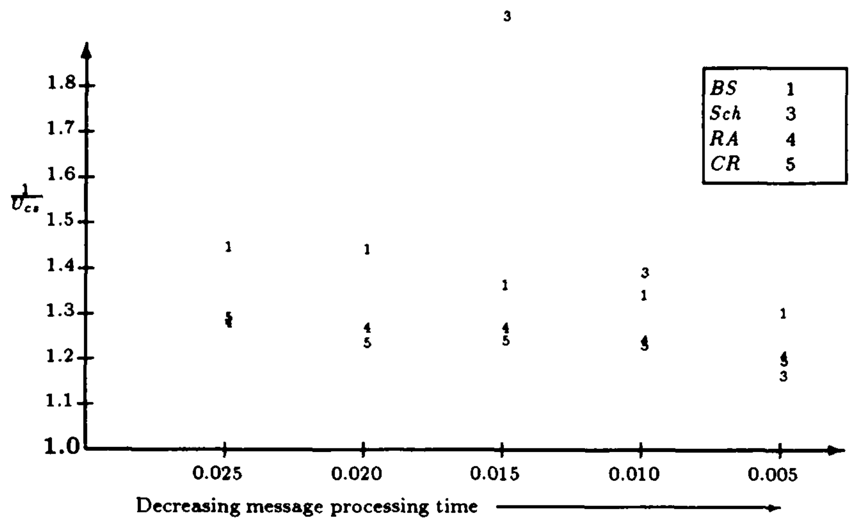
It is only when the number of processes is increased to 21 that the fully distributed algorithms exceed the efficiency of the centralized implementation. This higher efficiency holds to an even greater extent as the number of processes increases to 31. It is interesting to note that the *BS* algorithm performs in an almost identical manner for both 21 and 31 processes. Part of the cost associated with the *BS* algorithm may be the thousands of unnecessary probes of the controller by processes waiting for the critical section. For a 31-process system with $t_p = 0.005$, there is an average of 7 probes sent to the controller every time unit. Even though the communications medium is fully-connected,



Number of Processes: 21 With Broadcast

Figure 5.7: Algorithms in Fail-prone Environment

these *probe* messages will have an impact on the efficiency of the algorithm — especially since *probe* messages are assigned higher priority than the other mutual exclusion messages that are part of the algorithm. An alternate protocol might lower this number of unnecessary probes and improve the performance of the *BS* algorithm. Perhaps the biggest surprise continues to be the efficiency of the *Schneider* algorithm in fail-prone systems with small message processing times.



Number of Processes: 31 With Broadcast

Figure 5.8: Algorithms in Fail-prone Environment

Chapter 6

Conclusions

The contributions of this research can be classified into five main areas:

1. Definition of an underlying distributed system.
2. Identification of new performance metrics to measure the efficiency of distributed mutual exclusion algorithms.
3. A model of distributed mutual exclusion algorithms and analytical approximations of their performance.
4. Indepth simulation of distributed mutual exclusion algorithms on fully-connected, ring and star topologies.
5. Indepth simulation of distributed mutual exclusion algorithms on a fully-connected topology where processes may fail.

First, we have defined a model of an underlying distributed system, which represents a loosely coupled network where processes communicate only by exchanging messages. With this model, we identified and defined the parameters associated with both fail-free and fail-prone environments.

Second, we have identified a robust performance metric that enables one to compare the efficiency of all distributed mutual exclusion algorithms on any topology with any number of processes. This metric results in an immediate ranking of any algorithm for which we might seek performance measures.

Third, with an underlying system defined, and a performance metric identified, we introduced a state model of distributed mutual exclusion algorithms. This model enabled us to accurately predict the performance of centralized, fully distributed and hybrid implementations of mutual exclusion algorithms on a variety of topologies. The usefulness of the model was validated by applying it to a newly-published tree-based algorithm. We accurately predicted how this new algorithm performs on a fully-connected topology.

Fourth, we charted the performance of our algorithms in a fail-free, fully-connected distributed system. We followed that by describing performance results for a fail-free environment on ring and star topologies. These results indicate when a particular algorithm is appropriate for use in broadcast and no-broadcast systems.

Finally, we charted the performance of the *RA*, *CR*, *Schneider* and *BS* algorithms in a fail-prone environment where frequent failure and rapid recovery of processes are the norm. These results identify which algorithms perform most efficiently in broadcast and no-broadcast distributed systems with fully-connected topologies. In conjunction with obtaining these results, we identified three aggressive restart protocols and identified the intervals necessary to successfully detect failure.

6.1 Modeling the Underlying Distributed System

We have presented a detailed model of an underlying distributed system. Our model is a representation of loosely-coupled networks where message transmission time is significant and messages are queued at the destination.

Parameters to our model in the absence of failure are the following:

- N — number of processes in the system
- t_{ncs} — time a process spends executing outside the critical section
- t_{cs} — time a process spends executing within the critical section
- t_p — message processing time
- t_{msg} — message transmission time
- *broadcast* — whether or not a system has broadcast capability
- *topology* — the topology of the communications medium

When process failure is introduced, the following parameters are added to the model:

- t_{fail} — interval between process failures
- t_{delay} — time from when a process sends a message, until it inquires whether or not the receiver failed
- t_{probe} — time it takes a process to determine whether another process has failed
- t_{dead} — time a process remains failed
- t_{reboot} — time from when a process becomes active until it is ready to run a restart protocol

We can program centralized, fully distributed or hybrid implementations of distributed algorithms within the context of our model. Using the parameters defined for the model, we can explore the performance of both fail-free and fail-prone systems.

6.2 Choosing a Performance Metric

We have shown that the current metric in use to compare the performance of distributed mutual exclusion algorithms is not, by itself, sufficient to quantify their efficiency. This metric is the number of messages sent for each use of the critical section. The widespread use of this metric is not surprising — it is well-defined and easily calculated. Since the communications medium is a shared resource, it seems logical that the fewer the messages required, the better the algorithm. Using this metric, our algorithms are rated under conditions of maximum contention as follows:

1. *BS* — 3 messages
2. *Maekawa* — $4\sqrt{N}$ messages
3. *CR* and *RA* — $2 * (N - 1)$ messages
4. *Schneider* — $2 * N * (N - 1)$ messages

But when message processing time is negligible, we find that the algorithms are rated as follows when we measure the interval from when one process exits the critical section until the next process enters:

1. *CR*, *RA* and *Schneider* — t_{msg}
2. *Maekawa* — $\frac{2N-K}{N}t_{msg}$
3. *BS* — $2t_{msg}$

These rankings are inconsistent. The second ranking clearly shows that the *BS* algorithm is least efficient; yet the first ranking indicates that it sends the fewest messages for each use of the critical section.

We showed that we can build on the idea of the number of messages sent per use of the critical section by the introduction of two new metrics. Since messages burden the sites in the distributed system that must transmit and receive them, our first metric considers the number of messages that must be transmitted or received by each process in the system for a particular algorithm. And, because mutual exclusion messages contend with one another in the network, our second metric considers how these messages are distributed over time.

These two metrics poignantly revealed the shortcomings of the *Schneider* algorithm. The *Schneider* algorithm heavily burdens every process by requiring all processes to send and receive $2 * (N - 1)$ messages for every use of the critical section — including those processes which never seek access to the critical section. By examining how the *Schneider* algorithm behaves over time, it becomes clear that the *Schneider* algorithm regularly sends large bursts of messages into the network. In all but a fully-connected network, it appears the *Schneider* algorithm will perform poorly.

The fully distributed implementation of both *CR* and *RA* require the process seeking access to the critical section to send and receive most of the messages. The process never seeking access to the critical section must send and receive a single message. Over time, these algorithms send fairly large bursts of messages into the system at regular intervals. If messages should be routed through a small number of nodes in the communications medium, it appears that these algorithms may suffer from the effects of contention.

The hybrid algorithm of *Maekawa* requires the process seeking access to the critical section to send and receive $2\sqrt{N}$ messages for each use of the critical section. Processes never seeking access to the critical section average sending and receiving less than a single message for each use of the critical section. We can also note that small numbers of messages are sent into the communications medium at regular intervals.

These new metrics show that the *BS* algorithm operates efficiently both in number of messages sent and received and in the area of message conjestion over time. A process that does not seek access to the critical section does not take part in the mutual exclusion algorithm; only the central controller and the process seeking access to the critical section need exchange 3 messages. Over time, the *BS* algorithm sends a negligible number of messages into the network.

Our analysis using these metrics has been a subjective one. It will change, depending on the topology. For example, those algorithms sending bursts of messages into the network will perform less efficiently on a star topology than on a fully-connected topology; how much less efficiently is not clear. In addition, these metrics do not address the length of the chain of messages between events in a distributed system. These shortcomings were motives for finding additional metrics of performance.

A completely different approach to selecting a performance metric requires viewing the critical section as a shared resource for which processes throughout the distributed system contend. With this view, the critical section is modeled by the classical "queue/server pair" from queueing theory. The server represents the critical section which serves a single process at a time; the queue represents the set of processes seeking access to the critical section. This model presents four common performance metrics: the number of critical section usages per time unit (throughput); the elapsed time between requesting the critical section and acquiring it (response time); the number of processes waiting to acquire the critical section (queue length); and the proportion of time the critical section is in use (utilization).

We have argued that all four of these metrics result in an identical ranking of performance when there is maximum contention for the critical section. Our task was to determine which of these four metrics is most useful for the purpose of ranking distributed mutual exclusion algorithms. Our goal was to create a "fingerprint" of an algorithm's performance; once this fingerprint is obtained for a group of algorithms, it is desirable that a ranking immediately follow. When two fingerprints are "equal", the efficiency of the algorithms should be equivalent. When one fingerprint is in some sense "smaller" than another, the algorithm with the smaller fingerprint should be more efficient than the algorithm with the larger one. One method of obtaining a fingerprint is by use of a ratio. We decided to use the performance measurements of the idealistic algorithm for creating this ratio.

Since $S_{cs} = 1.0$ for our simulations and by the the Utilization Law, $U = XS$, we know that $U_{cs} = X_{cs}$ — therefore, either utilization or throughput will give us an identical result. We have arbitrarily chosen utilization as being preferable to throughput. We have already noted that $U_{ideal} = 1.0$ for the critical section of the idealistic algorithm. By taking the ratio $\frac{U_{ideal}}{U_{cs}}$, we compute a value greater than 1. For instance, when message processing time is negligible, $U_{RA} = \frac{1}{t_{cs}+t_{msg}}$. When $t_{msg} = 0.1$ and $t_{cs} = 1.0$, $\frac{U_{ideal}}{U_{RA}} = \frac{1.1}{1} = 1.1$. For the *BS* algorithm, $\frac{U_{ideal}}{U_{BS}} = \frac{1.2}{1} = 1.2$. These fingerprints show that the *RA* algorithm is more efficient than the *BS* algorithm under our assumption of negligible message processing time.

We also might have chosen response time, or queue length as the performance metric from which we obtain a ratio. What we found, however, was that the queue length and response time for the ideal algorithm vary according to the number of processes in the distributed system. Suppose, for example, we wish to compare the efficiency of the *RA* algorithm on a 21-process system with itself on a 31-process system. By looking at mean response time and queue length data of the idealistic algorithm, we find it is different for 31 and 21-process systems. With no common denominator in the ratio, we cannot directly compare performance of the algorithm on two systems of different sizes. Only with a common basis of comparison can we determine the overhead added to the algorithm with the addition of more processes to the system.

We find that not only is U_{ideal} easy to calculate (it's always 1.0), but it provides a common basis for the comparison of the performance of all mutual exclusion algorithms, independent of the number of processes in the system. We will also find that we can compare the performance of these algorithms on varying topologies using the metric of utilization in both fail-free and fail-prone environments; hence, we can also measure the impact of changing topology or adding process failure to a particular algorithm.

Our next requirement was to determine if we could analytically predict the performance of our mutual exclusion algorithms on varying topologies. We developed a model based on the state of a distributed system and the application of a function to that state [AB].

6.3 Analyzing the Effects of Topology

Every mutual exclusion algorithm consists of two distinct parts:

1. How the next process is chosen to enter the critical section when the critical section becomes available.

2. How the chosen process recognizes it has been selected.

The first part of a mutual exclusion algorithm, how the next process is chosen to enter the critical section, requires that an algorithm provide a mechanism whereby a process is chosen to enter the critical section from among those processes waiting to enter it. This mechanism can be thought of as applying a function f to the state S of the distributed system and then rendering a decision d as to which process is next to enter the critical section. The application of the function is dependent on the state maintained by each process in the system.

$$f(S) \rightarrow d$$

We have already noted that the state of some process i is denoted by s_i, and the state of the distributed system, S, is the set of states consisting of the state of every process in the system. We defined S as $\cup s_i$.

A process that maintains global state S has a "snapshot" of the state of the system [CL]. At the other end of the spectrum, a process that maintains only its own local state, s_i, disregards the state of the other processes in the system. A process may also maintain partial state — something more than its own state, but less than global state. Partial state may not be a subset of global state; for example, a process may not know what state another process is in, but it may know what state it isn't in.

The second part of a mutual exclusion algorithm, how the chosen process recognizes it has been selected, follows directly frcm the application of the function f. If f is a distributed function and is applied by every process in the system to its local state, it may be the case that (1) every process maintains global state and can immediately determine which process is next to enter the critical section; or (2) every process maintains partial state and can immediately determine whether it has been chosen to enter the critical section or that it must wait for some number of subsequent *release* messages before accessing the critical section. In either of these cases, when a distributed function is applied to the local state of a process, the process can determine whether or not it has been chosen to enter the critical section, thus assuring mutual exclusion. If f is a centralized function and is applied by only the central controller, the decision d must be relayed by the controller to the chosen process.

The tradeoff between distributed and centralized implementations is now clear. The more local state a process maintains, the more messages an algorithm must send for each execution of the critical section. However, every process can apply f to its own local state and determine d. The less local state a process maintains, the fewer messages an algorithm must send for each execution of the critical section. However, when f is applied by only the central controller, d must be relayed to the chosen process. If it is more expensive to update local state than it is to relay d, then the centralized algorithm is the preferred implementation. If it is more costly to relay d than it is to update local state, then a distributed implementation is the preferred algorithm. The purpose of most hybrid algorithms is to reduce the number of messages required to update local state as well reduce the number of relays of d to the processes chosen to enter the critical section. These observations were necessary before we could accurately assess the performance of distributed mutual exclusion algorithms using analytical techniques.

We examined the performance of three mutual exclusion algorithms on three widely different topologies. Our three algorithms were the centralized implementation of *BS*, the hybrid implementation of *Maekawa*, and the fully distributed implementation of *RA*; these three algorithms are representative of the spectrum of distributed mutual exclusion algorithms. The three topologies chosen to garner our results were a fully-connected, ring and star configuration. The fully-connected topology represented the optimum configuration where contention is minimized and the diameter of the network is one. The ring topology represented a configuration with the greatest diameter short of a network where all processes are linked in a straight line. The star configuration represented a topology with maximum contention, where all messages are routed through a single central hub.

Before acquiring analytical results, we had to determine what quantity we would attempt to measure. Since our metric of choice is utilization, our measure should easily translate into utilization. Without message processing time, we easily computed the interval between successive executions of the critical section. We then transformed this interval into utilization. The RA algorithm, for example, requires t_{msg} time between each execution of the critical section when message processing time is negligible. Since $S_{cs} = 1.0$ for our results, we computed $U_{RA} = \frac{1}{1.0+t_{msg}}$. If we can factor in message processing time and contention, then this interval is a useful measure to obtain with our analytical method.

6.3.1 Fully-connected Topology

Without significant message processing time, analytical results for the fully-connected topology are trivial.

1. RA — t_{msg}
2. $Maekawa$ — $\frac{2N-K}{N}t_{msg}$
3. BS — $2t_{msg}$

We must now factor in message processing time and contention.

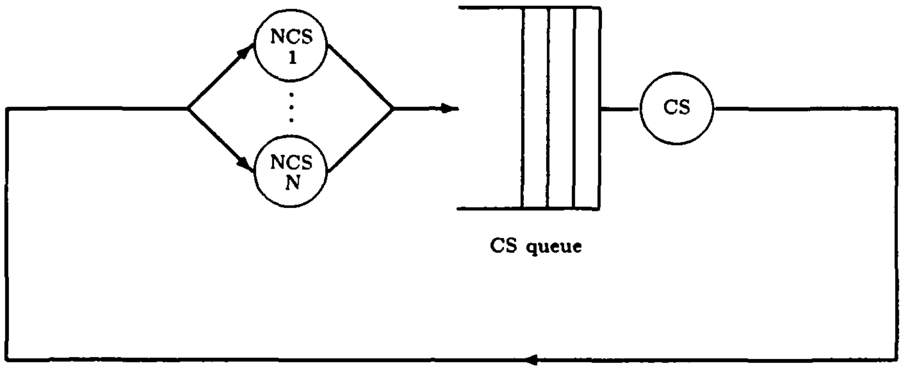


Figure 6.1: Queueing Network for Analytical Results

Using a Markov chain, we derived the following M/M/1/∞/m queueing formula for the queueing network shown in Figure 6.1.

$$p(0) = \frac{1}{\sum_{n=0}^{m} \frac{m!}{(m-n)!}(\lambda S)^n}$$

$$p(n) = \frac{m!}{(m-n)!}(\lambda S)^n p(0)$$

$$Q = \sum_{n=1}^{m}(n * p(n))$$

This formula is useful for several reasons. First, we can observe that processes waiting for the critical section do not send *request* messages for the critical section; consequently, if we can determine the length of the mutual exclusion queue, then we can determine how many processes will send *request* messages into the network. Second, when a process in a fully distributed implementation releases the critical section, it sends a message to all processes waiting in the mutual exclusion queue.

For the *RA* algorithm, with $t_{cs} = 1.0$, $t_{msg} = 0.1$ and $t_p = 0.005$, we obtain $S_{cs} = 1.11$ for our M/M/1/∞/m queueing formula. Letting $m = 31$, we compute $Q_{cs} \approx 22$. If we assume that a process exiting the critical section will send the message releasing the critical section to the next entering process, on the average, half-way through the messages it sends to those processes waiting for the critical section, we derive the following formula for our interval between each execution of the critical section:

$$\frac{Q_{cs}}{2}t_p + t_{msg} + t_p$$

However, we still have not factored in contention.

When a process requests the critical section, it sends a message to every process in the system. We can expect contention, then, at every process. We can use a heuristic where we assume that processes requesting the critical section contend for a common communications path. With our queueing formula, we then compute the mean number of messages on the common path at any one time. *By then assuming this mean value represents worst-case* contention at the CIP, we derive our interval. We have 22 processes in the mutual exclusion queue and 31 processes in the system; that leaves 9 processes to request the critical section. By letting $m = 9$, $t_{msg} = 0.1$ and $t_p = 0.005$, we obtain $S_{path} = 0.11$. Our queueing formula can then be used to compute $Q_{CIP} = 0.10722$. Our formula for the interval between successive executions of the critical section for the *RA* algorithm now becomes:

$$\frac{Q_{cs}}{2}t_p + t_{msg} + Q_{CIP}t_p + t_p$$

We did a similar derivation for the *BS* and *Maekawa* algorithms. Figure 6.2 and Figure 6.3 chart the relative error between actual results obtained with simulation and our analytical results derived using the heuristic described here. The figures indicate that our analysis was excellent for a fully-connected topology.

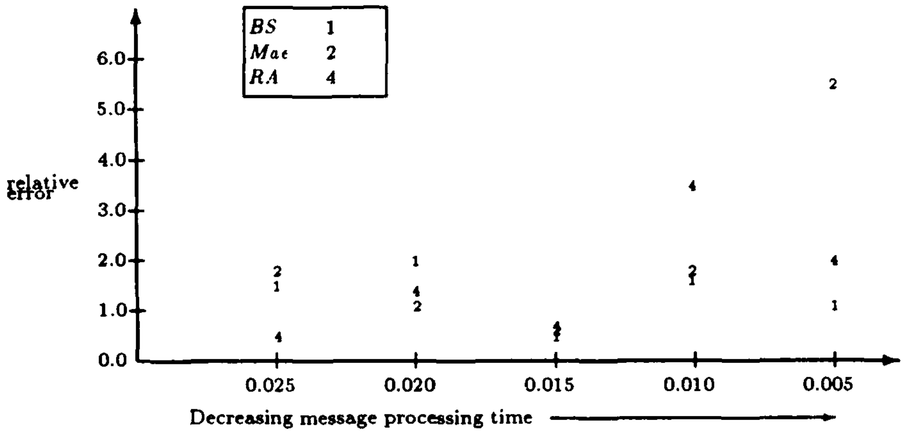


Figure 6.2: Relative Error in Analytical Approximation — 21 Processes in a Fully-connected Topology

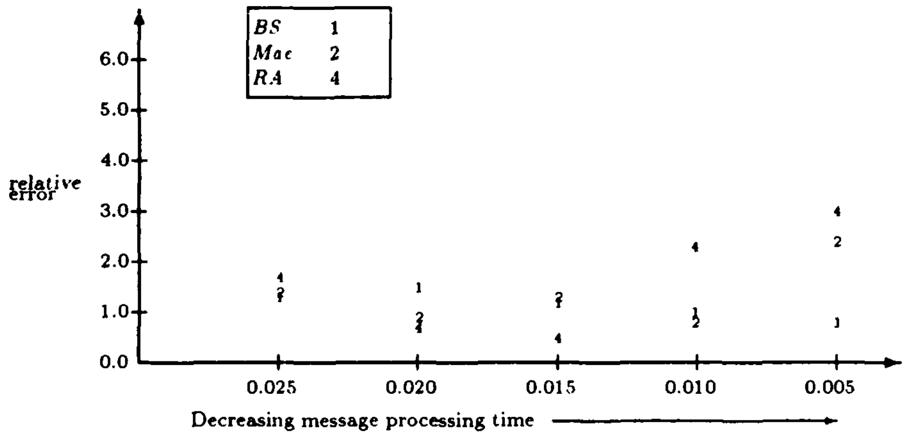


Figure 6.3: Relative Error in Analytical Approximation —31 Processes in a Fully-connected Topology

6.3.2 Ring Topology

For a ring topology, the notion of the length of the chain of messages between each event within a distributed system helped us establish performance in an environment without failure and with negligible message processing time. For example, the length of the chain of messages for the *RA* algorithm is one.

In addition, we can expect a *reply* message to travel half-way around the ring before reaching the process which is next to enter the critical section. Therefore, we can expect the interval between successive entries to the critical section to be $\frac{N}{2}t_{msg}$ time units for systems with an even number of processes and $\frac{N-1}{2}t_{msg}$ time units for systems with an odd number of processes.

For the *BS* algorithm, exactly one trip around the ring is required for each execution of the critical section; the controller issues a *grant* message to the next process to enter the critical section, and the user of the critical section sends a *release* message back to the controller when it exits. The expected interval between successive entries to the critical section is therefore Nt_{msg} time units.

Using the probabilities associated with each scenario in the *Maekawa* algorithm, we can expect performance equal to $\frac{K}{N}(\frac{N}{2}t_{msg}) + \frac{N-K}{N}(Nt_{msg}) = \frac{2N-K}{2}t_{msg}$ for ring topologies with an even number of processes. Our ranking for the ring topology is the following:

1. *RA* — $\frac{N}{2}t_{msg}$
2. *Maekawa* — $\frac{2N-K}{2}t_{msg}$
3. *BS* — Nt_{msg}

We now add message processing time and contention to our model.

Since every message is handled twice at each intermediate CIP, once by the sender and once by the receiver, we have $2Nt_p$ message processing time for each message sent on a ring. With $m = 31$, $t_{cs} = 1.0$, $t_{msg} = 0.1$ and $t_p = 0.005$, we derive $S_{cs} = t_{cs} + 15t_{msg} + 30t_p = 2.65$ time units for the performance of the *RA* algorithm. Using our M/M/1/∞/m queueing formula, we compute

$Q_{cs} \approx 28$. Utilizing the heuristic where only those processes executing outside the critical section and not waiting for the critical section send messages, we solve for Q_{CIP}. Since a message travels $N-1$ hops before being removed from the ring, we derive $S_{path} = (N-1)t_{msg} + 2(N-1)t_p = 3.3$ time units. With $m = 3$, we compute $Q_{CIP} = 1.029606$ using our queueing formula. But we must realize that this value represents contention for the entire ring. If we disperse this value equally across the ring, then contention becomes negligible. Hence, our estimate of 1.65 time units as the interval between successive entries to the critical section serves as a final result.

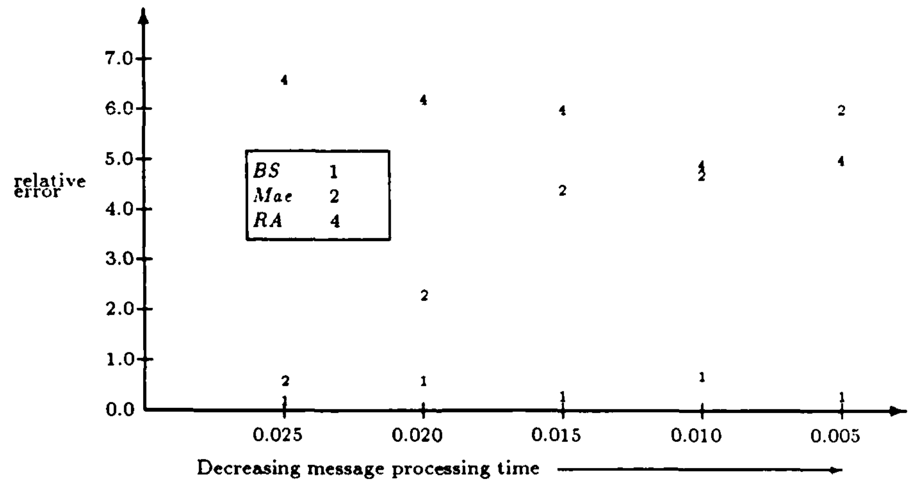


Figure 6.4: Relative Error in Analytical Approximation — 21 Processes in a Ring Topology

The assumption of negligible contention holds for the *BS* and *Maekawa* algorithms as well. Using our initial estimate and identical heuristics, we computed the relative errors shown in Figure 6.4 and Figure 6.5 for the ring topology. These figures show that we can effectively analyze the performance of an algorithm on a ring topology.

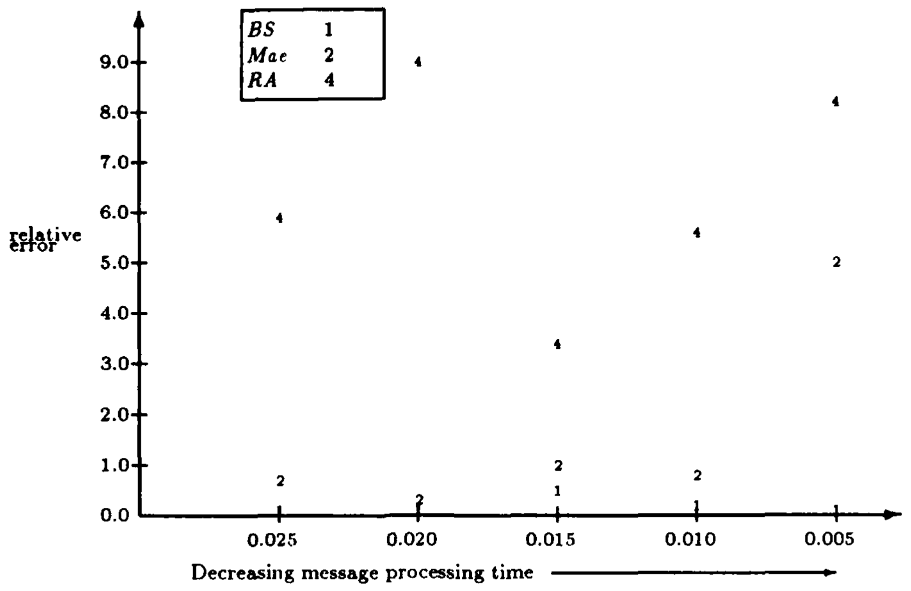


Figure 6.5: Relative Error in Analytical Approximation — 31 Processes in a Ring Topology

6.3.3 Star Topology

Analytical results for the star topology proved the most difficult to obtain. Contention was not significant in the two previous topologies, but it becomes an important part of our results for the star configuration.

With negligible message processing time, there is no contention in the star topology. With the hub of the star serving as controller in the *BS* algorithm, the time between critical section executions is $2t_{msg}$. For the *RA* algorithm, a message must travel to the hub and then be forwarded to its destination. With t_{msg} time required for each hop, the time between successive executions of the critical section is also $2t_{msg}$. For the *Maekawa* algorithm, every process is a controller. This means that the centralized portion requires a process to send messages over the hub — the hub cannot act as a controller.

For the centralized portion of the algorithm, there will be $4t_{msg}$ time units between successive entries to the critical section. The fully distributed portion's performance equals that of the *RA* algorithm. We derive the following estimate of performance for the *Maekawa* algorithm on a star topology with negligible message processing time:

$$\frac{K}{N}(2t_{msg}) + \frac{N-K}{N}(4t_{msg}) = \frac{4N-2K}{N}t_{msg}$$

We have the following ranking of our algorithms on a star topology:

1. *BS* and *RA* — $2t_{msg}$
2. *Maekawa* — $\frac{4N-2K}{N}t_{msg}$

Because the *BS* algorithm generates little contention except at the controller, our previous heuristic yields an excellent estimate of performance. With $m = 31$, $t_p = 0.005$ and $t_{msg} = 0.1$, we derive $S_{cs} = 1.22$ and $S_{path} = 0.11$. We now obtain the following results:

$$Q_{cs} = 23$$

$$Q_{CIP} = 0.09404$$

The time between successive executions of the critical section is:

$$2t_{msg} + 4t_p + Q_{CIP}t_p = 0.220470$$

The actual simulation result of 0.218660 gives us a relative error of 0.8%. Because of contention at the hub, this simple analysis is not sufficient for either the *RA* or *Maekawa* algorithms.

For the *RA* algorithm, $S_{cs} = t_{cs} + 2t_{msg} + 4t_p$ serves as an initial estimate for the interval from when one process exits the critical section until the next process enters. We can use this value to derive Q_{cs} as we have in the past; we now have the following formula as our interval for successive entries to the critical section:

$$2t_{msg} + 3t_p + \frac{Q_{cs}}{2}t_p$$

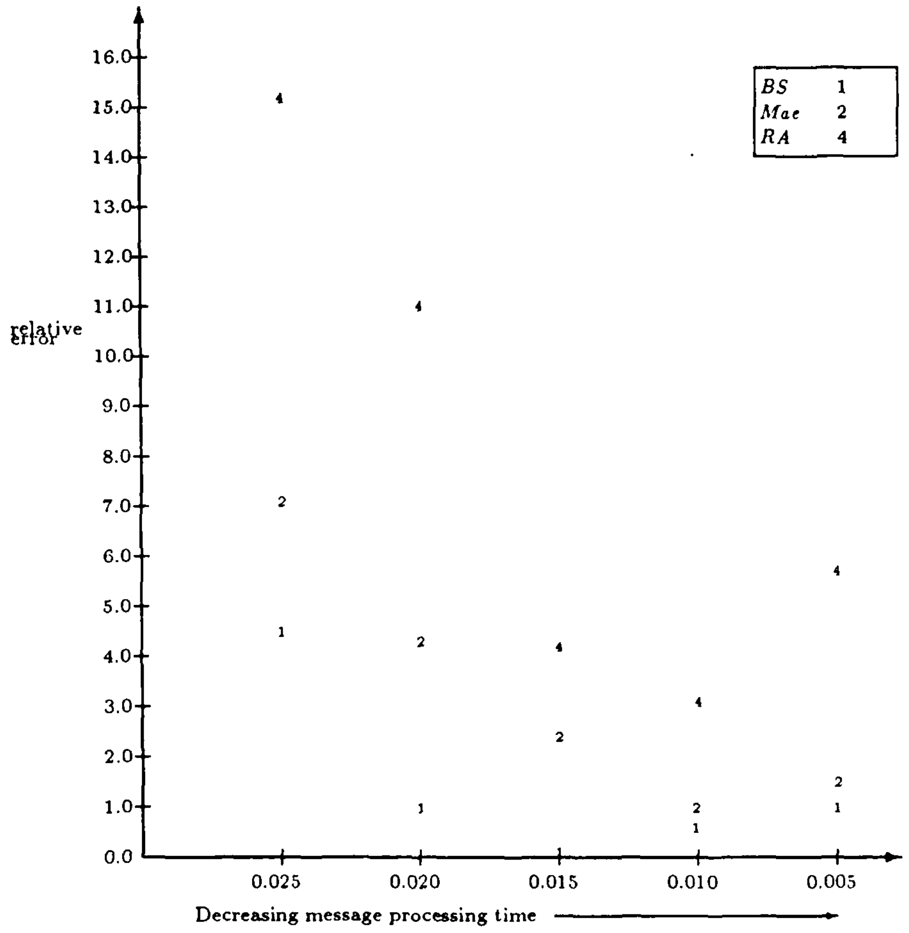


Figure 6.6: Relative Error in Analytical Approximation — 21 Processes in a Star Topology

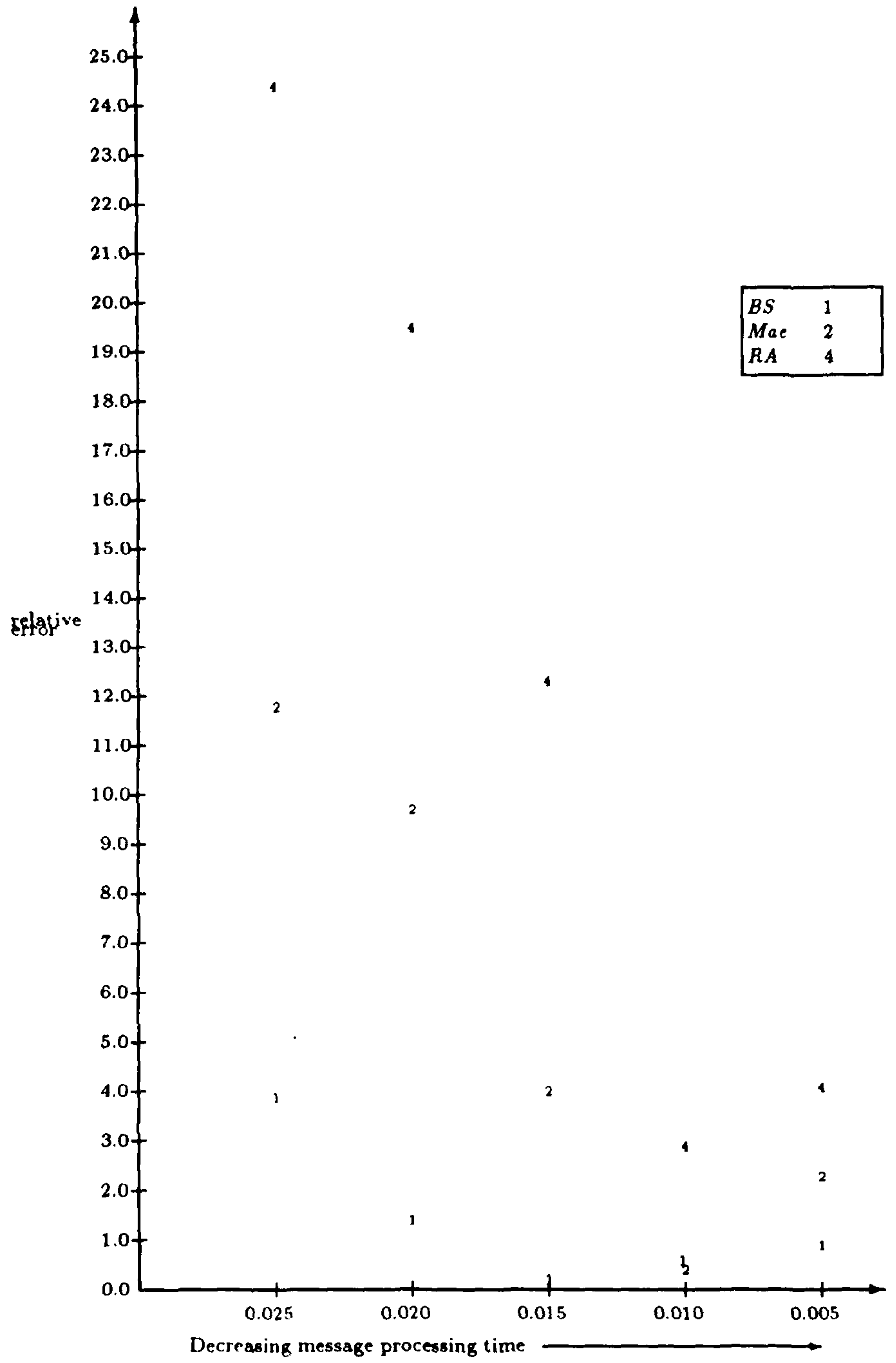


Figure 6.7: Relative Error in Analytical Approximation — 31 Processes in a Star Topology

We still have yet to factor in contention. Since the hub can only process messages sequentially, a message will not be sent until the previous message has been fully transmitted. If the previous message is a *request* message, there may be up to $m-1$ messages that must be transmitted until the subsequent message can be sent. If n processes are available to send *request* messages into the system, then for each *request* message sent there are $n-1$ *reply* messages sent also. If m is the number of processes in the system and Q_{cs} is the number of processes awaiting mutual exclusion, then $n = m - Q_{cs}$. Our measure for contention now becomes:

$$Q_{CIP} = Q_{CIP}(mt_p) + Q_{CIP}(m - Q_{cs} - 1)t_p$$

Since Q_{CIP} is defined in terms of itself, an iterative technique is the most obvious solution. To ensure that our results converged while using an iterative technique, we interpolated queue lengths for our M/M/1/∞/m formula, which takes only integer values.

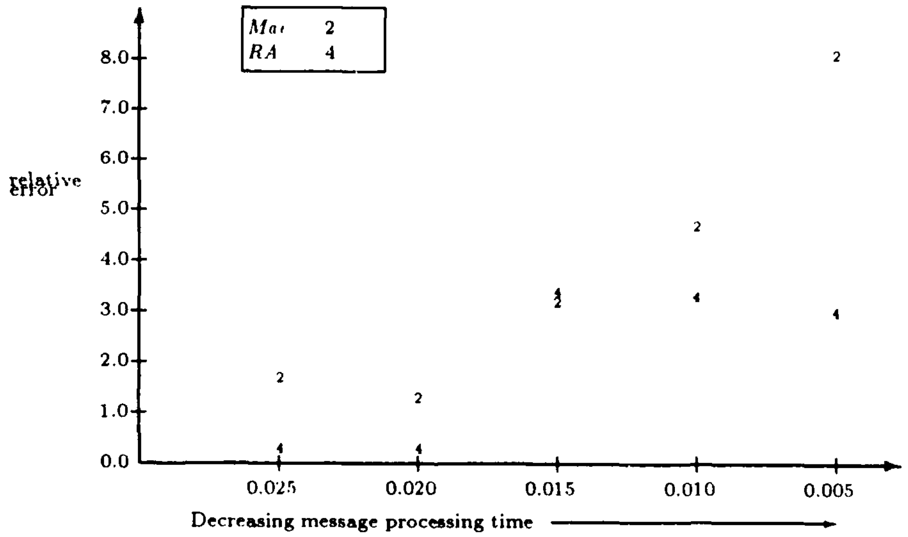


Figure 6.8: Relative Error in Analytical Approximation — 21 Processes in a Star Topology with Broadcast

We found that this same iterative heuristic using interpolation was also necessary to successfully estimate the performance of the *Maekawa* algorithm. Figure 6.6 and Figure 6.7 show the relative errors computed using these heuristics for a system without broadcast capability. Figure 6.8 and Figure 6.9 show the relative errors for a system with broadcast capability. We see that for systems without broadcast, we had difficulty deriving the performance of the *RA* algorithm at large message processing times. For systems with broadcast, we derived excellent results for virtually all of the algorithms.

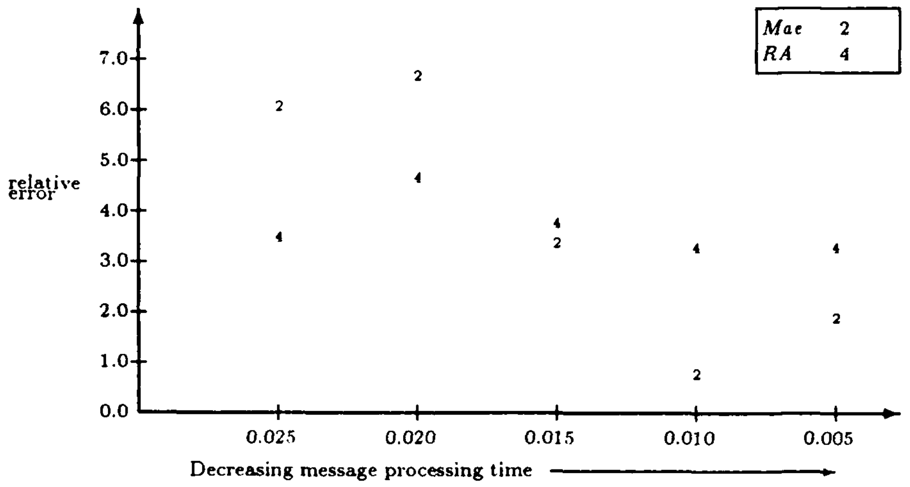


Figure 6.9: Relative Error in Analytical Approximation — 31 Processes in a Star Topology with Broadcast

6.3.4 Applying the Model to New Algorithms

We applied the methodology just described to a recently published distributed mutual exclusion algorithm [Raym]. We overlayed the radiating star topology suggested in the paper onto a fully-connected network. Relative error of our estimates to actual simulation results was less than 4.0% for a 21-process network for all message processing times. These results confirmed the usefulness of the model.

6.4 Catalog of Fail-free Simulation Results

We have two parameters that vary on a distributed system that does not experience failure. These parameters are message processing time, t_p, and the number of processes, N, in the system. Our goal in this section is to present the results in such a way that one can view graphically which algorithm is most

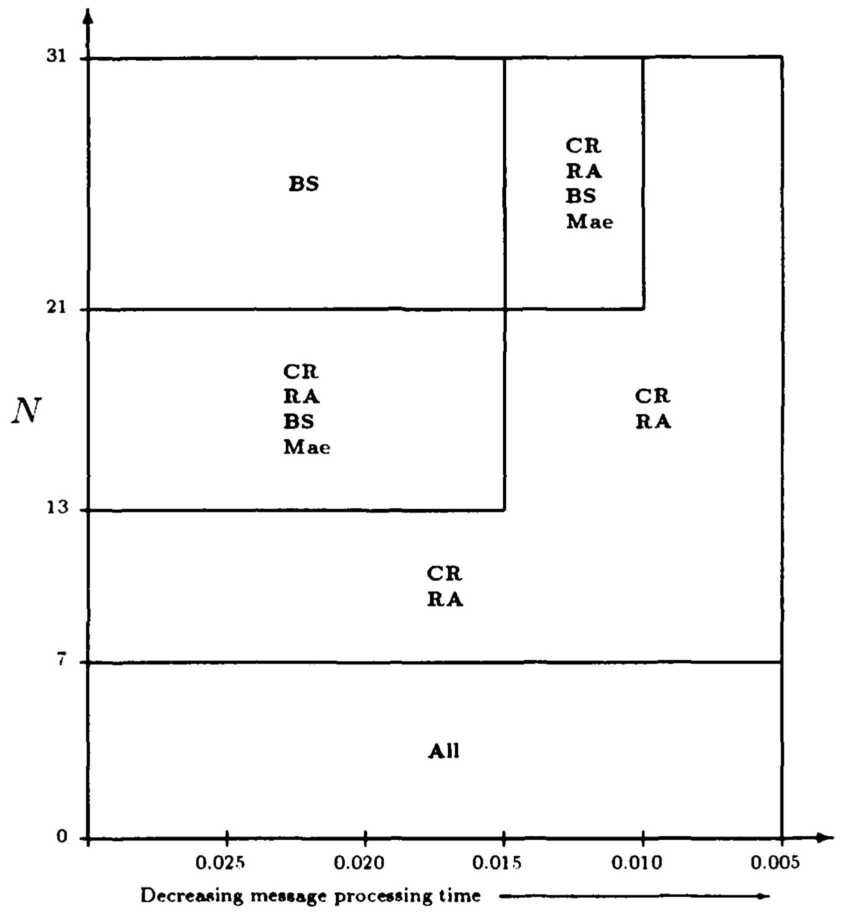


Figure 6.10: Algorithms in Fail-free Environment on Fully-connected Topology

suitable for a particular system. We also varied topology when examining the *BS*, *RA* and *Maekawa* algorithms in steady-state systems with 21 and 31 processes.

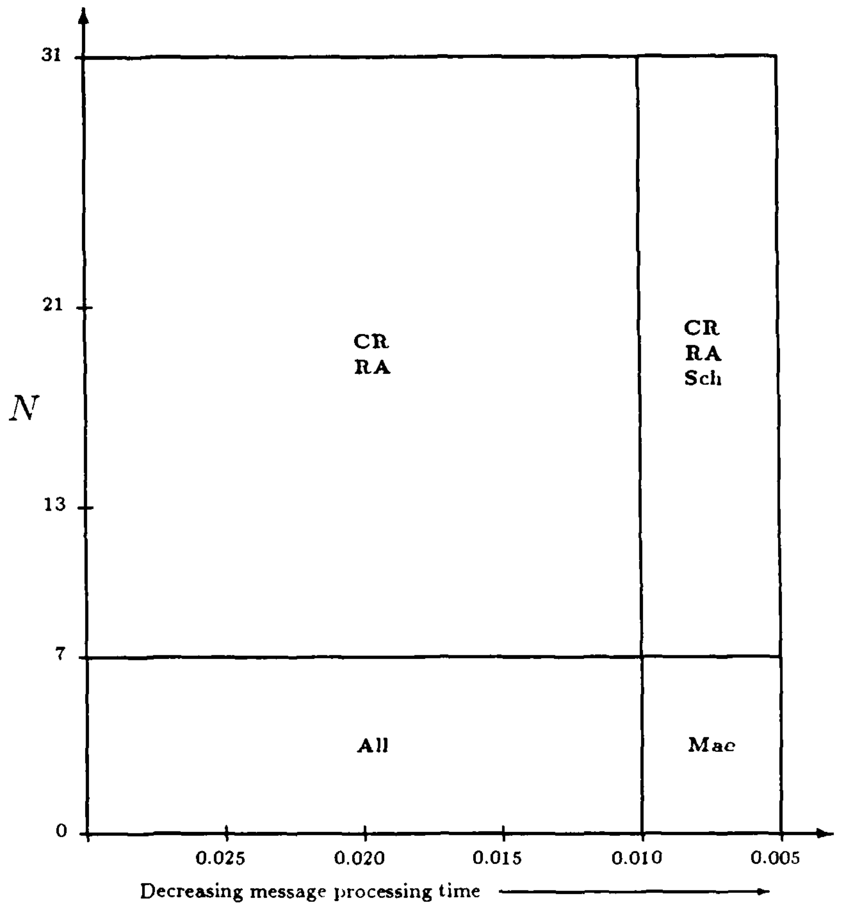


Figure 6.11: Algorithms in Fail-free Environment on Fully-connected Topology with Broadcast

In our figures, if more than one name appears within a block, it is because there is negligible difference in performance between the algorithms. We conducted a finite number of simulations. Therefore, the boundaries of the areas marked on our graphs may not occur exactly at 7, 13, 21 and 31 processes, nor exactly at message processing times of 0.005, 0.010, 0.015, 0.020 and 0.025.

6.4.1 Fully-connected Topology

The majority of our results were for the fully-connected topology. In general, in systems without broadcast, the centralized algorithm performed best in large systems with large message processing times; fully distributed algorithms perform best in almost all other systems. In small systems, no algorithm performed noticeably better than the others. Only as the number of processes increased past seven did the fully distributed implementations begin to perform more efficiently. With broadcast, fully distributed algorithms perform most efficiently virtually all of the time. For very small broadcast systems, the *Maekawa* algorithm appeared to be most suitable for small message processing times.

Figure 6.10 presents no surprises. During light contention, we expect the algorithms to perform with similar efficiency, since each process in all of the algorithms must request permission and receive a reply in order to enter the critical section. As contention increases, the mean length of the chain of messages approaches one for the fully distributed algorithms, and their performance begins to become more efficient than the others. As message processing time increases, the efficiency of the various algorithms become nearly equivalent for all but the *Schneider* algorithm. As message processing time and the number of processes continues to increase further, the fewer messages sent by the *BS* algorithm makes it most efficient in spite of its longer chain of messages.

Figure 6.11 reveals the efficiency of the fully distributed algorithms in fail-free broadcast systems throughout the range of message processing time and number of processes. One of the more interesting results is that the *Maekawa* algorithm is most efficient in small systems with small message processing times. Since the *Maekawa* algorithm does not necessarily enforce a total order on the FIFO mutual exclusion queue, it appears that with 7 processes and $K = 3$ that it captures the best of the centralized and fully distributed implementations. As the number of processes increases, however, the *Maekawa* algorithm generally exhibits the negative characteristics of the two implementations.

6.4.2 Ring Topology

The fully-distributed implementation of the *RA* algorithm performs most efficiently on a ring topology throughout the spectrum of message processing time. The *Maekawa* algorithm is a distant second in efficiency with the *BS* algorithm a close third. The ring topology never generates significant contention, since the system allows multicasting. Hence, the distributed implementations never generate the large number of messages that might otherwise impede their performance.

6.4.3 Star Topology

The centralized implementation of the *BS* algorithm runs most efficiently on a star topology. The star topology is a virtual reflection of the centralized implementation of mutual exclusion, and any other result would have been surprising. In addition, since the other algorithms send numerous messages across the hub of the network (even with multicasting), a bottleneck develops amidst the contention which grows rapidly as message processing time increases. Not surprisingly, the *Maekawa* eventually performs more efficiently than the *RA* algorithm as message processing time grows.

Adding broadcast capability to the system does not change our result with the *BS* algorithm performing most efficiently across the spectrum of message processing time. However, with broadcast, the *RA* algorithm performs more efficiently than the *Maekawa* algorithm.

6.5 Catalog of Fail-prone Results

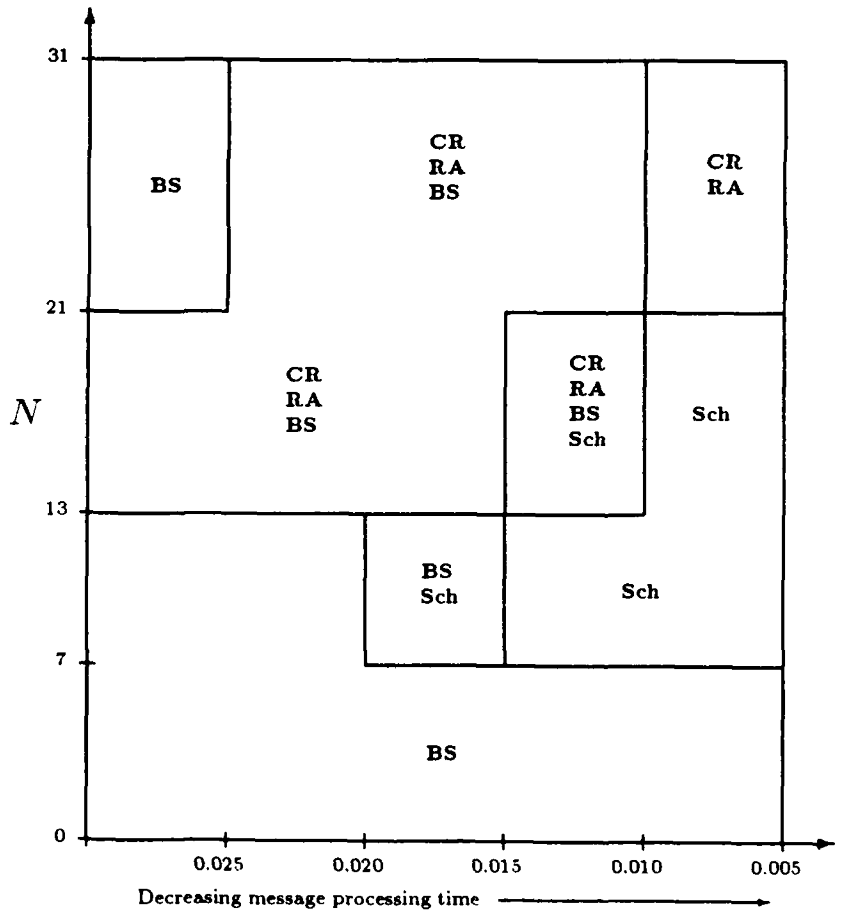


Figure 6.12: Algorithms in Fail-prone Environment on Fully-connected Topology

Our fail-prone results were derived on a fully-connected topology. We found that in small systems, the centralized implementation of mutual exclusion performed most efficiently. Since those processes not seeking access to the critical section can fail transparently to the system, we see a large advantage in the efficiency of the centralized algorithm compared to the others. Only when

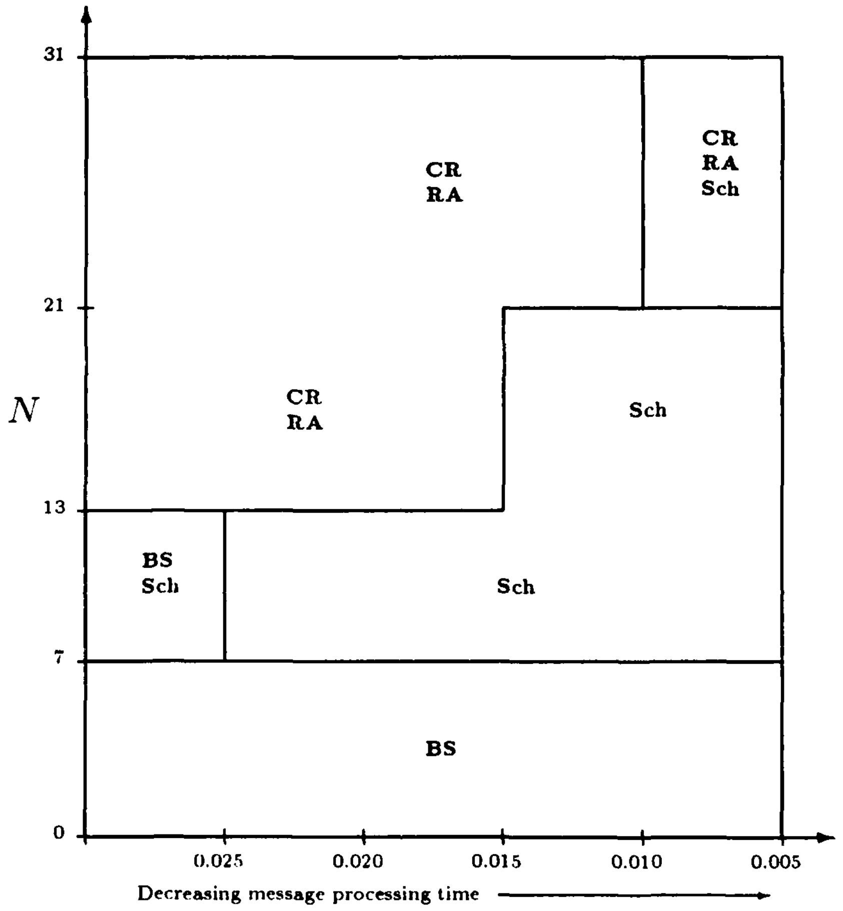


Figure 6.13: Algorithms in Fail-prone Environment on Fully-connected Topology with Broadcast

the number of processes exceeds 13 and approaches 21 do the fully distributed implementations begin to perform more efficiently; we see that as the number of processes increases, it becomes less likely that a process waiting near the head of the mutual exclusion queue will have a process ahead of it fail. Therefore, the *RA* algorithm can capitalize on its shorter chain of messages between executions of the critical section, resulting in greater efficiency. However, as message processing time continues to increase, the *BS* algorithm once again becomes most efficient in an environment without broadcast. In one of the more surprising results, the *Schneider* algorithm performs most efficiently in systems with 13 processes and message processing times of below 0.015 and in systems with 21 processes and a message processing time of below 0.010. The two-level failure detection scheme of the *Schneider* algorithm works effectively at these particular values. Processes that are waiting for entry to the critical section and then fail are primarily detected before they reach the head of the queue. That is, failure is detected when the failed process does not acknowledge a message, not when it has failed to release the critical section. In contrast, the *RA* and *CR* algorithms rely exclusively on detecting failure of processes in the critical section, and they run less efficiently in systems of this size. However, when the number of processes increases to 31, the *Schneider* algorithm begins to thrash as large numbers of *ack-probe* messages are sent into the network. Even when message processing time is small, the *Schneider* algorithm proves itself inefficient in relation to the other algorithms in large systems. The window where the *Schneider* algorithm is most efficient in non-broadcast systems is a fairly small one.

With broadcast, the centralized algorithm still dominates in small systems, while larger systems run more efficiently with fully distributed algorithms. The *Schneider* algorithm again proved the advantage of detecting failure with two separate mechanisms. It has surprising efficiency throughout all message processing times when there are 13 processes in the system and when message processing time is small in 21 and 31-process systems.

6.6 Summary

Our results indicate that the centralized implementation of mutual exclusion is superior in performance to its fully distributed counterparts in numerous types of networks. In systems with frequent process failure, the centralized algorithm can execute more efficiently in networks with small numbers of processes. Even in small systems with broadcast capability, the *BS* algorithm exhibits superior performance in the presence of frequent process failure. Though we have examined only systems that experience frequent process failure, it is significant that the centralized implementation proves to perform more efficiently than its distributed counterparts in the presence of failure in systems where the distributed algorithms performed most efficiently without failure. We have also shown that in broadcast systems with 21 or more processes, a distributed implementation is preferable in the presence of frequent process failure; we can expect that this result holds for all levels of failure as the distributed implementations also proved most efficient in identical systems in fail-free environments. The surprising results noted for the *Schneider* algorithm appear to indicate that some sort of failure detection scheme that augments detection of process failure in the critical section may be desirable in some systems. Topologies may also prove decisive in the selection of an algorithm. The centralized algorithm is far more efficient on a star topology than its hybrid or fully distributed counterparts; this observation can possibly be generalized to other systems with heavy contention. In contrast, the fully distributed algorithms are most efficient on a ring topology which has little contention and where the interval required for a message to travel from sender to receiver is large. The length of the chain of messages is shorter in fully distributed implementations, and they will be more efficient when contention is not a factor. These results all indicate that many variables must be prudently weighed before a decision is made as to which algorithm performs best in a particular environment.

Our model of computation enables us to reasonably predict the performance of distributed mutual exclusion algorithms. Networks with little contention prove easiest to solve analytically. Networks with heavy contention proved more difficult to solve accurately. As message processing time increased and contention grew, our results became less accurate. We applied the model to a newly-published tree-based algorithm. Our prediction of its performance was confirmed through simulation of the new algorithm. We believe it is possible to apply this model to any distributed mutual exclusion algorithm and make a reasonably accurate estimate of its performance on a given topology.

In any event, it is clear that there is much to be learned about the performance of distributed algorithms. Selecting appropriate metrics, predicting performance and simulating algorithmic behavior in fail-prone environments are a necessary part of the learning process.

6.7 Future Work

This section discusses some of the limitations of this work and identifies areas for future research.

6.7.1 Limitations

Our model of the underlying distributed system was designed specifically for distributed mutual exclusion algorithms. Other algorithms may introduce new parameters into our model and eliminate some of those we have identified. A more general model, which allows incorporation of new parameters into the model easily, will make it more attractive as a basis for studying distributed algorithms.

Our model of the underlying distributed system is also designed around process failure; the CIP does not fail and the network never partitions. The model does not provide sufficient detail to parameterize these important areas of concern.

Utilization is a useful metric for resource contention. It does not appear to be a general metric sufficient to measure the performance of all types of distributed algorithms. However, our metric of the "interval between successive executions of the critical section" can be generalized to the "interval between successive events in a distributed system". We have shown that this metric can be translated into utilization for the study of mutual exclusion; perhaps it can also be translated into meaningful metrics for other applications.

Our model for predicting the performance of distributed mutual exclusion algorithms provides excellent results for all topologies with little contention. When contention becomes a factor, our model still provides excellent results for systems with small message processing times. However, as message processing time and the accompanying contention grow, our estimates of their performance become less accurate.

Our fail-free simulation results provide insight into the performance of distributed mutual exclusion algorithms on varying topologies. However, we have limited ourselves to a small range of parameter values; there are many other values to be explored.

Finally, our fail-prone simulations assume worst-case intervals for determining the presence of failure within a system. Our simulations also only address frequent process failure, and they only examine fully-connected topologies. Other levels of failure should be inspected and other topologies used as a basis for results.

6.7.2 Enhancements

Several major areas of enhancement to the work are available. First, we can explore the performance of tightly-coupled networks. Tightly-coupled networks introduce new topologies: hypercube and grid to name two of them. They also introduce a host of different issues. Several of these issues are the queueing of messages at the host and message processing time that exceeds message transmission time.

Second, we have not attempted to predict the performance of our algorithms in the presence of process failure. With a given failure rate and the intervals identified for discovering failure, it may be possible to estimate the performance of our algorithms in the presence of failure.

Third, the intervals for determining when a process should query another process as to whether or not it has failed can be adjusted to reduce the number of probe messages sent into the system. Extending these intervals has the negative effect of allowing a failed process to go undetected for a longer period of time; however, it has the positive effect of reducing the number of messages sent into the communications network by the algorithm.

Fourth, the algorithms that we have simulated can all be improved both in the fail-free and fail-prone environments. It appears possible to apply all that has been discovered with our simulations and create an "optimal" algorithm for both a fail-prone and fail-free system. It may be the case that the optimal algorithm for a fail-prone environment is quite different for that of a fail-free one; it may also be the case that we can construct one that is the best compromise for both environments. This is an area that is wide open for research.

6.7.3 Epilogue

There are numerous distributed algorithms for which elegant solutions are proposed. Questions are still open as to their efficiency and robustness. Identification of metrics to measure their performance and methods to predict their efficiency are still largely unknown.

BIBLIOGRAPHY

[AB] Aahlad, Y. and Browne, J.C. (1987): Balanced protocols for sequencing distributed computations; *TR-87-39*, Dept of Computer Sciences, University of Texas at Austin.

[AHU] Aho, A.V., Hopcroft, J.E., and Ullman, J.D. (1974): *The design and analysis of computer algorithms*; Addison-Wesley Publishing Company.

[AL] Andrews, G. and Levin, G. (1982): On-the-fly deadlock prevention; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 165-172.

[AlSa] Albert, A.A. and Sandler, R. (1968): *An introduction to finite projective planes*; Holt, Rinehart and Winston, Inc.

[AS] Andrews, G.R. and Schneider, F.B. (1983): Concepts and notations for concurrent programming; *Computing Surveys 15 (1)*, 3-43.

[Bag] Bagrodia, R. (1987): On the design of high performance distributed systems; *PhD Dissertation*, Dept of Computer Sciences, University of Texas at Austin.

[BCM] Bagrodia, R., Chandy, K.M., and Misra, J. (1987): A message-based approach to discrete-event simulation; *to appear in IEEE Transactions on Software Engineering*, Dept of Computer Sciences, University of Texas at Austin.

[Bru] Brumfield, J. (1987): Concurrent programming in Modula-2; *Eighteenth SIGCSE Technical Symposium on Computer Science Education*, 191-200.

[BS83] Buckley, G. and Silberschatz, A. (1983): An effective implementation for the generalized input-output construct of CSP; *ACM TOPLAS 5 (2)*, 223-235.

[BS84] Buckley, G. and Silberschatz, A. (1984): A fault tolerant centralized mutual exclusion algorithm; *Fourth International Conference on Distributed Computing Systems*, 347-356.

[BT] Bracha, G. and Toueg, S. (1984): A distributed algorithm for generalized deadlock detection; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 285-301.

[CG] Chang, C. and Gouda, M. (1987): Bidding against conspirators; *in preparation*, Dept of Computer Sciences, University of Texas at Austin.

[Cha87] Chandy, K.M. (1987): A theorem on termination of distributed systems; *TR-87-09*, Dept of Computer Sciences, University of Texas at Austin.

[CL] Chandy, K.M. and Lamport, L. (1985): Distributed snapshots: determining global states of distributed systems; *ACM Transactions on Computer Systems 3 (1)*, 63-75.

[CM82] Chandy, K.M. and Misra, J. (1982): A distributed algorithm for detecting resource deadlocks in distributed systems; *Proceedings* *of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 157-164.

[CM84] Chandy, K.M. and Misra, J. (1984): The drinking philosophers problem; *ACM TOPLAS 6 (4)*, 632-646.

[CM87a] Chandy, K.M. and Misra, J. (1987): *Parallel program design: a foundation*; Addison-Wesley Publishing Company.

[CM87b] Chandy, K.M. and Misra, J. (1987): Conditional knowledge as a basis for distributed simulation; *5251:TR:87*, Computer Science Dept, California Institute of Technology.

[CR] Carvalho, O. and Roucairol, G. (1983): On mutual exclusion in computer networks; *CACM 26 (2)*, 146-147.

[CS] Chandy, K.M. and Sauer, C.H. (1978): Approximate methods for analyzing queueing network models of computing systems; *Computing Surveys 10 (3)*, 281-317.

[Dij65] Dijkstra, E.W. (1965): Solution of a problem in concurrent programming control; *CACM 8 (9)*, 569.

[Dij74] Dijkstra, E.W. (1974): Self-stabilizing systems in spite of distributed control; *CACM 17 (11)*, 643-644.

[Dij75] Dijkstra, E.W. (1975): Guarded commands, nondeterminancy and formal derivation of programs; *CACM 18 (8)*, 453-457.

[FY] Francez, N. and Yemini, S. (1985): Symmetric intertask communication; *ACM TOPLAS 7 (4)*, 622-636.

[Gar] Garcia-Molina, H. (1982): Elections in a distributed computing system; *IEEE Transactions on Computers C-31 (1)*, 48-59.

[GK] Gopal, I. and Kermani, P. (1986): Distributed resource sharing in computer networks; *Foundations of Software Technology and Theoretical Computer Science, Sixth Conference Proceedings*, 319-327.

[GLSZ] Graham, G.S., Lazowska, E.D., Sevcik, K.C., and Zahorjan, J. (1984): *Quantitative system performance*; Prentice Hall.

[Gou] Gouda, M. (1987): The stabilizing philosophers: asymmetry by memory and by action; *TR-87-12*, Dept of Computer Sciences, University of Texas at Austin.

[GS] Griefer, A. and Strong, R. (1988): DCF: distributed communication with fault tolerance; *Research Report RJ 6361*, IBM Almaden Research Center.

[GZ] Gusella, R. and Zatti, S. (1986): An election algorithm for a distributed clock synchronization program; *6th International Conference on Distributed Computing Systems, IEEE Proceedings*, 364-371.

[Hoa] Hoare, C.A.R. (1978): Communicating sequential processes; *CACM 21 (8)*, 666-677.

[KL] Kelton, W.D. and Law, A.M. (1982): *Simulation Modeling and Analysis*; McGraw-Hill, Inc.

[Kna] Knapp, E. (1987): Deadlock detection in distributed databases; *working paper*, Dept of Computer Sciences, University of Texas at Austin.

[Koo] Koo, R. (1987): Techniques for simplifying the programming of distributed systems; *TR-87-858*, Dept of Computer Science, Cornell University, Ithaca, New York.

[KS] Korth, H.F. and Silberschatz, A. (1986): *Database system concepts*; McGraw-Hill, Inc.

[Kum] Kumar, D. (1985): A class of termination detection algorithms for distributed computations; *Foundations of Software Technology and Theoretical Computer Science, Fifth Conference*, 73-100.

[Lak] Lakshmi, M.S. (1987): A study and analysis of performance of distributed simulation; *TR-87-32*, Dept of Computer Sciences, University of Texas at Austin.

[Lam78] Lamport, L. (1978): Time, clocks, and the ordering of events in a distributed system; *CACM 21 (7)*, 558-565.

[Lam84] Lamport, L. (1984): Solved problems, unsolved problems, and non-problems in concurrency; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 1-11.

[Lam86a] Lamport, L. (1986): The mutual exclusion problem: part I — a theory of interprocess communication; *JACM 33 (2)*, 313-326.

[Lam86b] Lamport, L. (1986): The mutual exclusion problem: part II — statement and solution; *JACM 33 (2)*, 327-348.

[LM] Livny, M. and Manber, U. (1987): μ — a system for simulating and implementing distributed and parallel algorithms; *CSTR-737*, Computer Sciences Department, University of Wisconsin at Madison.

[LSP] Lamport, L., Shostak, R. and Pease, M. (1982): The byzantine generals problem; *ACM TOPLAS 4 (3)*, 382-401.

[LP] Levi, S. and Plateau, B.D. (1986): A distributed algorithm for deadlock and termination detection of distributed computations; *CS-TR-1750*, Department of Computer Science, University of Maryland at College Park.

[Mae] Maekawa, M. (1985): A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems; *ACM Transactions on Computer Systems 3 (2)*, 145-159.

[Mar] Marzullo, K. (1983): Maintaining the time in a distributed system; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 295-305.

[Mis83] Misra, J. (1983): Detecting termination of distributed computations using markers; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 290-294.

[Mis86] Misra, J. (1986): Distributed discrete-event simulation; *Computing Surveys 18 (1)*, 39-65.

[MM] Mitchell, D. and Merritt, M. (1984): A distributed algorithm for deadlock detection and resolution; *Proceedings of the ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 282-284.

[MOO] Maekawa, M., Oldehoeft, A. and Oldehoeft, R. (1987): *Operating systems, advanced concepts*; Benjamin/Cummings Publishing Company, Inc.

[MS] Mohan, C. and Silberschatz, A. (1981): Distributed control — is it always desirable?; *IEEE Symposium on Reliability in Distributed Software and Database Systems*, 102-106.

[PS] Peterson, J.L. and Silberschatz, A. (1985): *Operating system concepts (second edition)*; Addison-Wesley Publishing Company.

[RA] Ricart, G. and Agrawala, A. (1981): An optimal algorithm for mutual exclusion in computer networks; *CACM 24 (1)*, 9-17.

[Ray] Raynal, M. (1986): *Algorithms for mutual exclusion*; MIT Press.

[Raym] Raymond, Kerry (1989): A tree-based algorithm for distributed mutual exclusion; *ACM Transactions on Computer Systems 7 (1)*, 61-77.

[Sch] Schneider, F. (1982): Synchronization in distributed programs; *ACM TOPLAS 4 (2)*; 179-195.

[Schw] Schwetman, H. (1988): Process level parallel programming; *MCC-ACA-ST-055-88*, Microelectronics and Computer Technology Corporation, Austin, Texas.

[Sil81] Silberschatz, A. (1981): Port directed communication; *The Computer Journal 24 (1)*, 78-82.

[Sil84] Silberschatz, A. (1984): Characterization and analysis of distributed control schemes; *working paper*, Dept of Computer Sciences, University of Texas at Austin.

[Sin] Singh, A.K. (1989): The paradigm of rankers; *PhD Dissertation, in preparation*, Dept of Computer Sciences, University of Texas at Austin.

[SK] Spezialetti, M. and Kearns, P. (1986): Efficient distributed snapshots; *6th International Conference on Distributed Computing Systems, IEEE Proceedings*, 382-388.

[SuKa] Suzuki, I. and Kasami, T. (1985): A distributed mutual exclusion algorithm; *ACM Transactions on Computer Systems 3 (4)*, 344-349.

[Sta] Stankovic, J.A. (1988): A serious problem for next-generation systems; *IEEE Computer*, 10-19.

[Tan] Tanenbaum, A.S. (1988): *Computer networks (second edition)*, Prentice-Hall, Inc.

VITA

Samuel Grier was born on 21 April 1951 in Bethesda, Maryland, the son of Samuel Lycurgus Grier and Mary Bosserman Grier. Upon graduation from Fort Hunt High School in Alexandria, Virginia, in 1969, Samuel entered the U. S. Air Force Academy in Colorado Springs, Colorado. After graduating from the Academy, he attended Undergraduate Pilot Training at Williams AFB, Arizona, and was chosen to fly Boeing KC-135A aircraft out of Loring AFB, Maine. He flew as copilot, and then as pilot, of the KC-135A before being selected for Graduate School to study for a Masters Degree in Computer Science. In the Fall of 1979, Samuel entered Graduate School at the University of Colorado in Boulder, Colorado, under the sponsorship of the Air Force Academy. He did his thesis work under Doctor Lloyd Fosdick, resulting in a program for detecting plagiarism in Pascal programs. After graduation in December of 1980, he was stationed at the Air Force Academy as an Instructor and subsequently Assistant Professor of Computer Science. The Air Force Academy selected him to study for his Ph.D. in Computer Science; however, prior to his studies, Samuel was required to complete 11 years of Air Force flying. From 1983 until 1986, he flew the KC-135A aircraft as an aircraft commander, instructor pilot, and evaluator pilot at Fairchild AFB, Washington. His aircrew was recognized as the Best KC-135A Crew in the Strategic Air Command in the 1985 Bombing/Navigation Competition, and his crew was subsequently recognized as the best KC-135A Crew overall in 15th Air Force. In 1985, his introductory Pascal textbook, Pascal for the Eighties, was published by Brooks/Cole Publishing Company. In the Fall of 1986, Samuel entered the Graduate School of the University of Texas at Austin to obtain his Ph.D. in Computer Science.

Permanent address: 18423 Cedar Drive
Triangle, Virginia 22172

This dissertation was typeset[1] with LATEX by the author.

[1] LATEX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TEX program for computer typesetting. TEX is a trademark of the American Mathematical Society. The LATEX macro package for The University of Texas at Austin dissertation format was written by Khe-Sing The.