1

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

Efficient Protocols for Distributed Mutual Exclusion and Termination Detection

by

Martin Shepherd Chidyausiku

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfillment of
the requirements for the degree of
Master of Computer Science

Ottawa-Carleton Institute of Computer Science

**Carleton University
Ottawa, Ontario
March 29 1993**

© Copyright
1993, Martin Shepherd Chidyausiku

National Library
of Canada

Acquisitions and
Bibliographic Services Branch

395 Wellington Street
Ottawa, Ontario
K1A 0N4

Bibliothèque nationale
du Canada

Direction des acquisitions et
des services bibliographiques

395, rue Wellington
Ottawa (Ontario)
K1A 0N4

Author - L'auteur

Title - Notre référence

The author has granted an irrevocable non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of his/her thesis by any means and in any form or format, making this thesis available to interested persons.

L'auteur a accordé une licence irrévocable et non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de sa thèse de quelque manière et sous quelque forme que ce soit pour mettre des exemplaires de cette thèse à la disposition des personnes intéressées.

The author retains ownership of the copyright in his/her thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without his/her permission.

L'auteur conserve la propriété du droit d'auteur qui protège sa thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

ISBN 0-315-84103-6

Canada

Name John Doe University of Michigan

Dissertation Abstracts International is arranged by broad, general subject categories. Please select the one subject which most nearly describes the content of your dissertation. Enter the corresponding four-digit code in the spaces provided.

Communication 0000

SUBJECT TERM

0000

SUBJECT CODE

U·M·I

Subject Categories

THE HUMANITIES AND SOCIAL SCIENCES

COMMUNICATIONS AND THE ARTS

Architecture 0729
Art History 0377
Cinema 0900
Dance 0378
Fine Arts 0357
Information Science 0223
Journalism 0391
Library Science 0399
Mass Communications 0708
Music 0413
Speech Communication 0459
Theater 0465

EDUCATION

General 0515
Administration 0514
Adult and Continuing 0516
Agricultural 0517
Art 0273
Bilingual and Multicultural 0282
Business 0688
Community College 0275
Curriculum and Instruction 0727
Early Childhood 0518
Elementary 0524
Finance 0277
Guidance and Counseling 0519
Health 0680
Higher 0745
History of 0520
Home Economics 0278
Industrial 0521
Language and Literature 0229
Mathematics 0280
Music 0522
Philosophy of 0998
Physical 0523

Psychology 0525
Reading 0535
Religious 0527
Sciences 0714
Secondary 0533
Social Sciences 0534
Sociology of 0340
Special 0529
Teacher Training 0530
Technology 0710
Tests and Measurements 0288
Vocational 0747

LANGUAGE, LITERATURE AND LINGUISTICS

Language
General 0679
Ancient 0289
Linguistics 0290
Modern 0291
Literature
General 0401
Classical 0294
Comparative 0295
Medieval 0297
Modern 0298
African 0316
American 0591
Asian 0305
Canadian (English) 0352
Canadian (French) 0355
English 0593
Germanic 0311
Latin American 0312
Middle Eastern 0315
Romance 0313
Slavic and East European 0314

PHILOSOPHY, RELIGION AND THEOLOGY

Philosophy 0422
Religion
General 0318
Biblical Studies 0321
Clergy 0319
History of 0320
Philosophy of 0322
Theology 0469

SOCIAL SCIENCES

American Studies 0323
Anthropology
Archaeology 0324
Cultural 0326
Physical 0327
Business Administration
General 0310
Accounting 0272
Banking 0770
Management 0454
Marketing 0338
Canadian Studies 0385
Economics
General 0501
Agricultural 0503
Commerce Business 0505
Finance 0508
History 0509
Labor 0510
Theory 0511
Folklore 0358
Geography 0366
Gerontology 0351
History
General 0578

Ancient 0579
Medieval 0581
Modern 0582
Black 0328
African 0331
Asia Australia and Oceania 0332
Canadian 0334
European 0335
Latin American 0336
Middle Eastern 0333
United States 0337
History of Science 0585
Law 0398
Political Science
General 0615
International Law and Relations 0616
Public Administration 0617
Recreation 0814
Social Work 0452
Sociology
General 0626
Criminology and Penology 0627
Demography 0938
Ethnic and Racial Studies 0631
Individual and Family Studies 0628
Industrial and Labor Relations 0629
Public and Social Welfare 0630
Social Structure and Development 0700
Theory and Methods 0344
Transportation 0709
Urban and Regional Planning 0999
Women's Studies 0453

THE SCIENCES AND ENGINEERING

BIOLOGICAL SCIENCES

Agriculture
General 0473
Agronomy 0285
Animal Culture and Nutrition 0475
Animal Pathology 0476
Food Science and Technology 0359
Forestry and Wildlife 0478
Plant Culture 0479
Plant Pathology 0480
Plant Physiology 0817
Range Management 0777
Wood Technology 0746
Biology
General 0306
Anatomy 0287
Biostatistics 0308
Botany 0309
Cell 0379
Ecology 0329
Entomology 0353
Genetics 0362
Immunology 0724
Microbiology 0410
Molecular 0307
Neuroscience 0317
Oceanography 0416
Physiology 0433
Radiation 0821
Veterinary Science 0778
Zoology 0472
Biophysics
General 0786
Medical 0780

EARTH SCIENCES

Biogeochemistry 0425
Geochemistry 0006

Geodesy 0370
Geology 0372
Geophysics 0373
Hydrology 0388
Mineralogy 0411
Paleobotany 0345
Paleoecology 0426
Paleontology 0418
Paleozoology 0985
Palaenology 0427
Physical Geography 0368
Physical Oceanography 0415

HEALTH AND ENVIRONMENTAL SCIENCES

Environmental Sciences 0768
Health Sciences
General 0566
Audiology 0300
Chemotherapy 0992
Dentistry 0567
Education 0350
Hospital Management 0769
Human Development 0758
Immunology 0982
Medicine and Surgery 0564
Mental Health 0347
Nursing 0569
Nutrition 0570
Obstetrics and Gynecology 0380
Occupational Health and Therapy 0354
Ophthalmology 0381
Pathology 0571
Pharmacology 0419
Pharmacy 0572
Physical Therapy 0382
Public Health 0573
Radiology 0574
Recreation 0575

Speech Pathology 0460
Toxicology 0383
Home Economics 0386

PHYSICAL SCIENCES

Pure Sciences

Chemistry
General 0485
Agricultural 0749
Analytical 0486
Biochemistry 0487
Inorganic 0488
Nuclear 0738
Organic 0490
Pharmaceutical 0491
Physical 0494
Polymer 0495
Radiation 0754
Mathematics 0405
Physics
General 0605
Acoustics 0986
Astronomy and Astrophysics 0606
Atmospheric Science 0608
Atomic 0748
Electronics and Electricity 0607
Elementary Particles and High Energy 0798
Fluid and Plasma 0759
Molecular 0609
Nuclear 0610
Optics 0752
Radiation 0756
Solid State 0611
Statistics 0463

Applied Sciences

Applied Mechanics 0316
Computer Science 0967

Engineering
General 0537
Aerospace 0538
Agricultural 0539
Automotive 0540
Biomedical 0541
Chemical 0542
Civil 0543
Electronics and Electrical 0544
Heat and Thermodynamics 0348
Hydraulic 0545
Industrial 0546
Marine 0547
Materials Science 0794
Mechanical 0548
Metallurgy 0743
Mining 0551
Nuclear 0552
Packaging 0549
Petroleum 0765
Sanitary and Municipal 0554
System Science 0790
Geotechnology 0428
Operations Research 0796
Plastics Technology 0795
Textile Technology 0994

PSYCHOLOGY

General 0621
Behavioral 0384
Clinical 0622
Developmental 0620
Experimental 0623
Industrial 0624
Personality 0625
Physiological 0989
Psychobiology 0349
Psychometrics 0632
Social 0451

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis,

**Efficient Protocols for Distributed
Mutual Exclusion and Termination Detection**

submitted by

Martin Shepherd Chidyausiku

in partial fulfilment of the requirements
for the degree of Master of Computer Science

Director, School of Computer Science

Thesis Supervisor

Carleton University

April 19, 1993

Abstract

This thesis is concerned with protocols in two areas of Distributed Computing, *Mutual Exclusion* and *Termination Detection*. We briefly address some protocols that have been published for these areas and then we present new, efficient protocols for both problems. We present a mutual exclusion protocol which needs a very low number of messages for each entry into the critical section. The assumptions we make for the protocol are: a complete network, no link failures, a maximum of $(n - 1)$ node failures at any one time and the fact that the failure and recovery of any node is detectable by every non-faulty node. Our algorithm is shown to be always correct and guarantees *CS* access to any non-faulty process regardless of how many and how frequently other processes may fail and recover.

The Shavit-Francez protocol for distributed termination detection can be regarded to be an optimal protocol and it has a message complexity of $b + n.e$, where b is the number of messages transmitted in the actual computation, n is the number of nodes available in the system and e is the number of links between the nodes in the system. We present a protocol with a message complexity of $b + 3(n - 1)$ using the same assumptions as those of the Shavit protocol. Protocol evaluation criteria have been defined in many previous works and we assert that the aspects of local storage and fault tolerance should be included in the evaluation. We also touch on the problem of defining *links* between related problems of Distributed Computing.

Acknowledgments

I would like to thank my wife Doris, my children Aisha and Wesley for the patience and support they gave which enabled me to complete this work. I would also like to express my deepest gratitude to my advisor Prof. Nicola Santoro for the complete academic and financial support he gave me. He allowed me to lean on his experience and was always ready to give me quality time even at unscheduled moments. I wish to express my gratitude to Ekow Otoo for his support. I also thank many others who I have not listed here for their support, both academic and otherwise.

Contents

Abstract	iii
Acknowledgments	iv
List Of Tables	x
List Of Figures	xi
I Introduction	1
1 The Distributed Environment	4
1.1 Why Distribute	5
1.2 Fundamental Differences of Various Systems	6
1.3 A Basic Model for A Distributed System	6
1.4 A Distributed Protocol	7
1.4.1 Nature of Protocols	9
1.4.2 Used Techniques	10
1.4.3 General Assumptions	11
1.5 General Aspects on Evaluating an Algorithm	11
1.6 Desirable Qualities	12
1.7 Definition of Desirable Qualities	12

2	Motivation	14
2.1	Mutual Exclusion: an introduction	14
2.2	Termination Detection: an introduction	15
2.3	Organization	15
II	Distributed Mutual Exclusion	16
3	Study of Previous Works	17
3.1	The Problem of Mutual Exclusion	18
3.2	General Notes on Previous Works	19
3.3	Permission Based Protocols	20
3.3.1	The Maekawa Protocol	20
3.3.2	The Sanders Protocol	22
3.3.3	The First Singhal Protocol	23
3.3.4	The Rangarajan-Tripathi Protocol	24
3.3.5	The Woo-Newman-Wolfe Protocol	25
3.3.6	The Raymond Protocol	26
3.3.7	The Bar-Noy et.al. Protocol	28
3.3.8	The Second Singhal Protocol	29
3.3.9	Summary	30
3.4	Token Based Protocols	30
3.4.1	The Suzuki-Kasami Protocol	31
3.4.2	The Helary et.al. Protocol	31
3.4.3	The Nishio-Li-Manning Protocol	32
3.4.4	The Chang-Singhal-Liu Protocol	34
3.4.5	The Srimani-Reddy Protocol	35
3.4.6	The Satyanarayanan-Muthukrishnan Protocol	36
3.4.7	The Naïmi-Trehel Protocol	37

3.4.8	The Bernabeu-Auban et.al. Protocol	38
3.4.9	Summary of Token Based Protocols	39
3.5	Summary of Studied Literature	40
4	An Efficient Distr. Mutual Exclusion Protocol	42
4.1	Introduction	42
4.2	Assumptions	43
4.2.1	Basic Assumptions	43
4.2.2	Message Transmission Assumptions	43
4.2.3	Fault Assumptions: Fault Model	44
4.3	The Protocol: An Informal Description	45
4.3.1	Scenario in the Absence of Faults	46
4.3.2	Scenario in the Presence of Faults	48
4.4	Description of the Protocol	49
4.4.1	Message Types and Sizes	49
4.4.2	General Rules	51
4.4.3	Fault Tolerance Rules	52
4.5	Correctness	56
4.6	Number of Messages per Event	58
4.7	Storage Considerations	63
4.8	Assessments, Extension and Possible Generalizations	63
4.8.1	Assessment	63
4.8.2	Extension	64
4.8.3	Possible Generalization	64
4.9	Summary	65

III	Termination Detection	66
5	Survey of Related Work	68
5.1	Classification of Termination Detection Techniques	69
5.1.1	Diffusing Computation	69
5.1.2	Time Stamping	69
5.1.3	Circulating Token	70
5.1.4	Credit Sharing	70
5.1.5	Basic Assumptions for Termination Detection Protocols	70
5.2	Brief Survey of Termination Detection Algorithms	71
5.2.1	The Dijkstra-Scholten Algorithm	71
5.2.2	The Misra Algorithm	73
5.2.3	The Mattern Algorithm	74
5.2.4	The Huang Algorithm	75
5.3	A Brief Discussion on other Related Issues	76
5.4	Complexity Measurements	77
5.5	Summary	78
6	An Efficient Termination Detection Algorithm	79
6.1	Principles of the Algorithm	79
6.2	Assumptions and the Basic Model of the Algorithm	82
6.3	The Algorithm	82
6.3.1	Message Types	83
6.3.2	Local Storage	84
6.3.3	Rules of the Algorithm	84
6.4	Correctness	85
6.5	Message Complexity	88
6.6	Summary of the Algorithm	89
6.6.1	Assessment of the Algorithm	89

6.6.2	Fault Tolerance Considerations	90
6.7	Summary of Termination Detection	90
7	Thesis Conclusions	92
7.1	Future Work	93
A	Pseudocode Examples	96
A.1	Coordinating the Mutual Exclusion	96
A.2	Going Into Critical Section	98
A.3	Process Failures	99
A.4	Recovery of A Crashed Process	100
B	Code for the Optimal Termination Detection Algorithm	101
	References	104

List of Tables

3.1	Worst-Case Complexities of Some Algorithms (in absence of faults)	40
4.1	Number of Messages Per Event	61
5.1	Worst-Case Complexities of Some Termination Detection Algorithms	78
6.1	Worst-Case Complexities of Optimal TD Algorithms	88

List of Figures

1.1	Examples of topologies	8
1.1	Scenario in the absence of faults	47
1.2	Process states	50
1.3	Failure and recovery of processes	55
1.4	Fault tree	59

*Ah, for just one time I would take the
Northwest Passage
To find the hand of Franklin reaching for
the Beaufort Sea
Tracing one warm line through a land so
wild and savage
And take the Northwest Passage to the Sea !*

- Stan Rogers -

Part I

Introduction

One of the main tasks in developing a computer system both in terms of hardware and software [15] is the definition and development of *control structures* or *control primitives* around which the system components are built. In this thesis we discuss control primitives for a distributed computing environment. A *Distributed Computing environment* is one where the various, not necessarily homogeneous components are physically partitioned and communicate solely via *message passing* [18, 34, 36]. The definition and development of control primitives for a distributed environment is made more complex by the fact that the components involved do not share any resources [8, 20, 14]. A control primitive for such an environment would have to take many things ¹ into consideration including faults that can occur. There are control primitives which can be regarded as traditional distributed control primitives, such as Deadlock Detection. A lot of research has been dedicated to these control primitives resulting in very efficient algorithms. Links have been established between some aspects of distributed computing algorithms such that an efficient algorithm in one area would yield an efficient algorithm in the other ² but the aspect of linkage has been relatively less researched. The area of distributed computing is rapidly expanding and becoming central in computer science and establishing links between the various aspects becomes an issue of primary importance in order to limit the otherwise inherent developmental redundancy. In this thesis we focus on certain aspects of control in distributed computing, namely *Distributed Mutual Exclusion* and *Distributed Termination Detection*.

Mutual Exclusion (*ME*) is a situation where different sites in a distributed system compete for a certain privilege which can only be granted to a maximum of one site (in some cases k sites) only at a time [17, 23, 29]. Different protocols have been presented to deal with the problem of distributed mutual exclusion and depending on the assumptions that have been made for them, efficient results have been obtained (see Part II). In this thesis we present a protocol for distributed mutual exclusion which improves on most results and has an integral fault tolerance in built

¹The different aspects are dealt with in the following chapters

²An example is the link between Garbage Collection and Graph Marking [45-30-66]

into it. Our protocol achieves mutual exclusion in the worst case (in the absence of faults) with 3 messages per request.

Termination Detection (*TD*) is the situation where each site should be able to become aware of the situation where all the sites have completed a certain predefined computation or have all become passive with no messages in transit [22, 26, 47]. Tel[66] shows that the protocol by Shavit and Francez[60] to be an optimal protocol for termination detection. In this thesis we present a termination detection protocol which improves on the Shavit-Francez results. The Shavit and Francez protocol can detect termination using $b+n\cdot\epsilon$ messages and our protocol uses $b+3(n-1)$ messages. In both cases b stands for the messages that are transmitted within the monitored computation, n stands for the number of nodes in the system and ϵ stands for the number of links between the nodes. Note that the words *site*, *node* and *process* are used interchangeably in this thesis. Also the words *algorithm* and *protocol* mean the same thing in this thesis.

Chapter 1

The Distributed Environment

The components of a distributed environment do not have access to shared resources like shared memory. They only communicate with each other through the exchange of messages. The message propagation is bound to finite time delays and can either be made within *FIFO* or *non-FIFO* channels. The communication subsystem may also be faulty where sites can crash or links can fail. Protocols depend on the *underlying*¹ topology of the distributed network, types of message channels involved and the types of faults that can occur. However, the fact that nodes or processes can fail is an integral part of the distributed system model. Any protocol, be it on distributed mutual exclusion or on termination detection or on any other control aspect should and must take process or node failures into consideration.

Failures occur at random and in order to clearly express the ideas behind protocols that we later describe we state the following in connection with faults:

Any fault that may take place is detectable. When faults are not detectable then useful computation is impossible. The reason is that since the system is asynchronous, it is impossible to distinguish between a slow message and a failed link. This result was obtained in 1985 by Fisher Lynch and Paterson [21]. One possible general method of dealing with faults is that when a process detects the failure of another process, it logically blocks off the link to stop sending messages to it. A

¹Control primitives are similar to the kernel that runs underneath and applications are above it, the word *underlying* can be misleading but it has been adopted for the actual computation in this area

process can still receive messages from a link that it has logically blocked off. When a process detects the recovery of a process it unblocks the link to it.

Failure detection is simultaneous. When a failure occurs then all nodes detect this fault simultaneously. When a process receives a message at the same time that it detects a failure then the handling of the failure takes precedence over the message processing. The processing of a message or detection of a failure are atomic, i.e. they cannot be interrupted. This means that we have imposed an order in the concurrent access to the event queue. When we consider messages that arrive at a process alone, they arrive in any order. When we consider fault detection alone then faults are also detected in any order.

1.1 Why Distribute

The advantages of distribution can be viewed both from the user's and from the designer's point of view. Physical partitioning of the system allows processing to be done using the method of divide and conquer. This means that a complex problem is divided into smaller subsets which are then dealt with individually by the individual sites before the collective solution is submitted at one site or broadcast to all sites. This has a net effect of speeding up the overall computation. The existence of networks with shared peripherals such as printers necessitates the development of efficient distributed algorithms.

Distribution of a distributed system concerns both control and data. The aim of distributing data is to make data available to all sites with minimum retrieval cost and increase the system resilience. Distributing control is a means by which all sites play any role without any particular site exercising more control than the others, i.e. it does away with central control.

1.2 Fundamental Differences of Various Systems

In stand-alone or centralized systems, all processing is done by a single processor which forms the bottleneck of the system [7]. The alternative is a parallel system. A parallel system refers to one where the processing is done concurrently by more than one processing unit. In this case one or more resources such as memory, global clock (hardware or logical), long term storage and communicating channels are shared. The different processing units communicate and synchronize via these shared resources (especially memory access primitives and the global clock) and the components are usually homogeneous. Access to any of the shared resources involves synchronization with the other peers. Shared resources make the passing of explicit messages between the different units redundant but the synchronization points become the ultimate bottleneck of these parallel systems. Methodologies that are designed for a distributed system can be simulated using either a centralized or a parallel system at acceptable costs. The reverse is not true. This makes distributed algorithms highly portable to any environment, centralized or parallel.

1.3 A Basic Model for A Distributed System

The subject of *distributed systems* represents various systems at different levels of abstraction. In fact, a distributed system can be compared to the 7 layer *OSI* stack model [6, 7] because each layer represents a level of abstraction. Just as in distributed systems, different nodes communicate through the exchange of messages. In the *OSI* model different layers communicate with each other through the exchange of messages. Each layer of the *OSI* reference model receives a packaged message from its underlying layer and unpacks the message so as to understand the message from a communicating peer layer. For each layer, a different type of protocol is specified since different problems occur at different layers or different specifications have to be met at different levels. In the distributed system the same kind of problems occur at the different level of abstraction and therefore the solutions that are defined generally apply to all the levels of abstraction. For example, an efficient sorting

protocol in the distributed system is just as relevant to the hardware designer as it is to the systems programmer and the software designer. The graphic model of the *OSI* reference model usually takes the same form of two stacks directly linked at the bottom and logically linked at each level with the number of levels in the stacks showing the layers being considered. A distributed system is generally depicted as a graph with restrictions imposed on it. In the Graph $G = (v, e)$ the vertices (v) represent the sites and the edges (e) represent the communicating channels.

Generally the communicating channels are asynchronous because synchronous channels are not yet fully feasible in a distributed environment. The restrictions imposed on a graph can take the form of imposing directional links or defining a fixed topology, such as a star or a ring or a complete graph. Some restrictions are a subset of others, e.g. an efficient protocol designed for a ring ought to yield better results in a complete graph and not vice versa. The reason is because a ring is a subset of a complete graph. Figure 1.1 shows some of the commonly used network topologies. There are some other topologies that are not depicted, such as *tori*, *hypercubes*, *grids*, ...etc. Normally more restrictive assumptions are made so that solutions to the less restrictive are included. Furthermore, distributed protocols designed for particular functions can solve other functions, though not as efficiently as protocols designed with the other functions in mind. For example, a distributed election protocol (all cooperating sites in the system unanimously agree on electing only one of them as a leader) always offers a derivative mutual exclusion solution for allowing one site at a time to perform a certain function, though this derivative solution might be less efficient (less fair)² compared to a dedicated solution.

1.4 A Distributed Protocol

Essentially, a distributed protocol or a distributed algorithm is a collection of definitions of processes and messages to be exchanged between the linked processes. Each process can receive a message from another process and can send messages to

²fairness is one main requirement to an efficient mutual exclusion algorithm

Figure 1.1: Examples of topologies

other processes. When a process receives a message, it processes the message, sends out output and enters a certain state. The processing of a message can involve the sending of messages to other processes which in turn act similarly. In general, a computation in a distributed environment can be described by the equation:

Process + Incoming-Message = Action + Outgoing-Message(s). Any system is subject to failures that can involve communication links, nodes or messages and the faults can be of a temporary or permanent nature.

1.4.1 Nature of Protocols

Algorithms are each dedicated to a particular control primitive. The control primitive can then be viewed as a perpetual function which is *ever-present* and ready to execute as often as it is called upon to do so. A control primitive should therefore not involve the storage of massive amounts of data or the spawning of large numbers of processes. Otherwise the control primitive begins to compete with the actual computation for resources. This makes it a liability to the system.

Distributed control primitives should be viewed as being distinct from an actual computation which performs a certain function and then terminates. It is continuously available services upon which computations can be designed. For example, an efficient implementation of the election primitive, one of the strongest primitives, has led to the development of numerous applications exhibiting very high performance and resilience in distributed systems. Each control primitive absolves a particular function. The functions may overlap or are heaped together in a certain manner to solve some larger problem. For example, one protocol can assume the existence of a group membership protocol that gives needed support to that protocol. Some control primitives are used to synchronize concurrent processes and others are used for global observation (or global snapshots) of processes in a system. A token passing system can even be made highly fault tolerant using control primitives for token regeneration and recovery. A protocol is said to exhibit symmetry [9], if any process can play any role in it. i.e. a centralized protocol is not necessarily symmetric because not every node might be capable of taking over from the central process. The

more symmetrical a protocol is the better because this improves on fault tolerance. Some literature define the symmetry of a protocol as the case in which the Id's of the processes are not used [45].

Generally the protocols use one of three techniques, *diffusing computations*, *token passing* and *time-stamping*.

1.4.2 Used Techniques

Diffusing computations [21, 47] usually involves a coordinating or starting process which initiates the computation. The computation then diffuses throughout the network, from one process to another through message passing. The messages can be broadcast³ [44] or simply sent to one other neighbor. There are two forms of broadcasting, *shouting* similar to talking to an audience where all listeners hear simultaneously and *whispering* similar to telephoning where the calls are conducted individually between two partners. The caller calls all his neighbors, one after the other.

Token passing [38, 12] is where a process must first be in possession of a token before it can perform a certain function. The token(s) is passed between the participating processes fairly. Generally a cyclic path for passing on the token has to be specified but this does not always have to be the case.

Time stamping [37, 35, 41] is a case where each process keeps track of either the number of transactions it is making or the number of transactions each participating process is making. This is done by defining a pseudo-clock for each process and whenever a message is sent, it is tagged with the clock value of the sender-process and the receiver process compares that clock value to its own in order to establish precedence. Usually the receiver process updates its clock if it is behind that of the sender process. This means that events can be sorted topologically and synchronization can be effected.

³each process receives from one neighbor and sends to all the other neighbors

1.4.3 General Assumptions

The description of any protocol must be preceded by fixing assumptions about the topology, the types of links available, the type of transmission⁴, size of messages, size of local storage and types of faults that can transpire. If any of these is not specified but is relevant then the worst case situation is assumed.

1.5 General Aspects on Evaluating an Algorithm

Normally a protocol is measured against other protocols by the number of messages it takes to perform a particular computation. This is referred to as the *message complexity*. Three different types of message complexities are: the *best case* in which the minimal possible message complexity is considered, the *worst case* in which the maximal possible number of messages is considered and the *average case*. The average case usually involves a foreknowledge of the running of the system. We only consider the worst case except where explicitly stated otherwise. In cases where a comparison is made between *near perfect* protocols or between protocols with the same message complexities other aspects have to be considered before either protocol can be declared an improvement on another. There are cases also where message complexity alone does not suffice to qualify a protocol as being superior to others because if one were to disregard message size and local storage one would be able to come up with algorithms of remarkable message complexity. It is this point that makes *Local Storage* more important.

1.6 Desirable Qualities

We are of the opinion that a correct protocol that has low message complexity, low local storage, high fault tolerance has all the desirable aspects of a good protocol. We are aware of the fact that these aspects do conflict in some cases, so the aim is to

⁴Examples are: *FIFO*-channels, message duplication, time-out, acknowledgement of message arrival

strike an optimal balance. Some protocols are developed with one main objective, e.g. to solve a long standing problem. In such cases aspects such as fault tolerance or local storage might remain understandably marginal.

1.7 Definition of Desirable Qualities

The main properties that any protocol must exhibit are:

Liveness : the protocol reacts to any situation promptly or within finite time and performs the task it was designed to perform. This means that the protocol eventually yields results in finite time [3].

Safety : the protocol does not jump to false conclusions, i.e. the protocol has an in-built guard against false interpretations leading to wrong results. In short, the protocol does not do anything wrong. Fulfilling *liveness* and *safety* means that no *deadlock* can occur.

Fault Tolerance/Resilience : this is an aspect which should be an integral part of any protocol [19, 25]. A protocol which is not fault tolerant might help to understand a problem by eliminating all the fault tolerance bulk but is not of any practical use because faultless networks are not feasible.

Low Message Complexity : this is the traditional measuring rod for protocols but in some cases it has to be traded off with local storage and fault tolerance. For example, a token regenerating protocol needs a lot of messages to initiate a non-redundant token generation and it needs a lot of messages to inform interested parties of a successful regeneration of a previously lost token. If this token regenerating mechanism were part of a token based termination detection protocol then the message complexity would go up, but so would the fault tolerance. Keeping track of tokens with local storage using only *FIFO* channels takes less messages than without local storage (e.g. in a directional ring) using k tokens. Another example is one where fault tolerance is enhanced

by replication and partition. In this case the loss of a node does not lead to a breakdown of the system because the data lost with the crashed process is also resident at other sites. This increases local storage and at the same time the number of messages needed to maintain consistency increase and this tarnishes the message complexity.

Low Local Storage : the protocol keeps its local storage at a minimum or at an acceptable level. Since this aspect has to be traded off with other aspects, the acceptable level is not fixed in any way but left to be within certain thresholds in comparison with other similar protocols. Local storage can be reduced by having highly symmetrical protocols [9, 10, 56] and having little or no reference to process names or Id's.

Chapter 2

Motivation

2.1 Mutual Exclusion: an introduction

Mutual Exclusion (*ME*) can be compared to an hourglass which allows k grains of sand to trickle down to the lower part without any blockage or deadlock. The accuracy or correctness of an hourglass depends on the fact that the rate of trickle of the grains of sand is maintained by having the grains be of predictable size and having the opening be of a fixed size. We can define fairness in *ME* as being similar to not rotating the hourglass until all the sand has passed on to the lower part i.e. each grain of sand has played its part in finite time although the order in which this happens is not of paramount importance. In *ME* the grains of sand represent the processes and the passage of each grain of sand from the upper part to the lower part represents a process entering the critical section. The comparison is limited by the fact that in *ME* one process can have more demand for *ME* than another one meaning that that process would be able to enter the critical section more often than another one. A control primitive for Mutual Exclusion should be correct, efficient in terms of the message complexity and should be fair by treating all sites equally, while guarding against deadlocks or faults. Message complexity and fault tolerance seem to always work against each other but a healthy compromise between the two is the desirable goal. Note that fault tolerance should be an integral part of every protocol because faults do exist. In fact, the basic model for mutual exclusion stipulates that

a node that fails outside of its critical section (*CS*) should not prevent any other node from entering its *CS*[34].

2.2 Termination Detection: an introduction

Normally each site in a distributed system is not permanently aware of the global states of the system otherwise global clocks would be implemented. Sometimes each site is only aware of its immediate neighbors and not the entire system's topology. It is the task of a Termination Detection primitive to make each site aware of a certain type of global state. Termination Detection can be compared to a Petri Net where a place only fires if all the necessary tokens have arrived. The number of tokens can be compared to the number of sites in a distributed system. The arrival of a token can be seen as the termination of a single site. The fact that a terminated site can restart due to a message arriving from another terminated site can be modeled into a Petri Net by introducing loops. Both *TD* and *ME* deal with communication and synchronization.

2.3 Organization

In Part II we study various previous algorithms that have been published in the area of Mutual Exclusion and then we present a new algorithm in the same area. In Part III we study some previous Termination Detection algorithms and then we present a new optimal algorithm for this area of distributed computing. In Part IV we summarize and point out some directions for future research.

Part II

Distributed Mutual Exclusion

Chapter 3

Study of Previous Works

In this part we discuss issues related to *Distributed Mutual Exclusion*. As was mentioned in Part I above, the area of mutual exclusion plays a central role in distributed systems. First we study some of the previous research that has been carried out on distributed mutual exclusion and then we introduce a new distributed mutual exclusion protocol.

The basic prerequisites for a mutual exclusion protocol were specified by Dijkstra in 1965 [18] and require that at most one process be in its *critical section* (*CS*) at a time and that the history of a process outside its (*CS*) has no influence on the mutual exclusion mechanism as a whole. Further it is also required that all processes be treated fairly in entering *CS*. Note that the words *process*, *node* and *site* are used interchangeably in this thesis. Also the words *algorithm* and *protocol* are used synonymously in this thesis. In the early stages of this research on mutual exclusion, this was regarded as a mechanism whereby one of two processes was allowed access to the critical section (*CS*) [40]. The research has since been generalized to allowing k processes among a total of n processes ($k < n$) to go into *CS* at a time. First we give a brief recapitulation on the problem of Mutual Exclusion, then we consider some of the previously published protocols on of distributed mutual exclusion.

3.1 The Problem of Mutual Exclusion

In a distributed environment, processes can only communicate through message passing and often compete for resources. Some resources are such that they can be accessed or used by one process at a time. Each process needs to access them at indefinite times (with non-deterministic frequency) during its computational history. The access of such a resource is generally critical. In order to prevent faults such as deadlocks and starvation, the following has to be ensured:

Mutual Exclusion (ME): Only one process can be in $C'S$ at any time. In some generalized cases the number of processes allowed to be in $C'S$ at any time is k where $(k < n)$.

Liveness: Any process that needs to access a resource can do so in finite time (a.k.a. *non-starvation* or *liveness*).

Fairness: All processes are treated equally and all have the same probability for entering $C'S$.

Basic requirement: The behavior of any process outside of its $C'S$ should not affect the requesting and processing of $C'S$ by any other process. This means that any proposed protocol must still execute correctly in spite of some nodes failing outside of the $C'S$. This minimal fault tolerance must be an integral part of every mutual exclusion protocol.

For each protocol some basic assumptions are made in order to incorporate the above mentioned criteria. The assumptions that are made for a protocol must always be stated clearly, otherwise the worst case must be assumed and the protocol has to perform as stated. Some of the important assumptions that have to be made concern message transmission, topology and fault types. Some protocols make more constraining assumptions than others and some do not fulfill the above mentioned requirements. For example, Martin[10] published a protocol whose execution does not guarantee fairness. For situations where fairness is not important, such a protocol is applicable.

3.2 General Notes on Previous Works

In 1986 Raynal[54] published a book on mutual exclusion. In 1988 he again wrote a book[55] on distributed protocols. Chapter 2 of the second book is dedicated to election and mutual exclusion algorithms. Both these books can be regarded as containing surveys of distributed mutual exclusion algorithms from the early stages of the research up to 1984. We address work reported after 1984. We address milestone protocols, i.e. those that introduced a new notion and other algorithms which illustrate a particular method in dealing with distributed mutual exclusion. We discuss protocols and their message complexities without restating the proof of their correctness. Interested readers may consult the referred publications.

Mutual exclusion protocols can be grouped into two different classes, the *permission based* and the *token based*. Some are time-stamping based and others are tree based. There has been a surge of tree based protocols recently and the technique used is to first create a logical spanning tree, impose it on the network and then manipulate it. Logarithmic message complexities have been achieved because an ideal tree has logarithmic height.

In the literature, e.g. [51], it has been stated that a protocol in which processes do not bear the same responsibility in controlling mutual exclusion is not distributed. We claim that a protocol can still be distributed although the momentary responsibilities of the processes are not necessarily equal. We claim that the main prerequisite for a distributed protocol is the ability or potential for every process to play some role in the system. This means that one process can momentarily play the role of a leader and in the next instance play a different role while another process plays the role of a leader. In the next chapter we present a protocol which is truly distributed but in which a single node at a time plays a somewhat extended role.

3.3 Permission Based Protocols

In permission based protocols a node needs to gain the permissions of a majority of the other nodes before it can go into CS. Nodes can crash and if a node were to wait for approval to come from a fixed set of nodes then the potential for deadlock in this case increases. It therefore has to be ensured that if a node crashes outside of a CS it does not affect the rest of the system. Improvement of permission based protocols involves optimally reducing the subset of nodes from which a requesting node has to seek approval. Some methods use quorums and some use hierarchies to achieve this. For simplicity, we divide the permission based protocols into two groups in our study. The first group use quorums or groups of processes from which permission must be obtained before entering CS. The second group request permission from all the other processes.

3.3.1 The Maekawa Protocol

The protocol by Maekawa[39] is the first to use quorums or groups. It has a worst case message complexity of $5\sqrt{n}$ in the absence of faults, where n is the number of nodes in the system. The principle of the protocol is to divide the n processes in the system into k intersecting groups of size $\sqrt{n}$, i.e. any pair of groups has a pairwise non-null intersection. A process that needs to go into critical section must first obtain permission from all the members of one group. A precondition is that each process can only issue a permission to one process at a time. Since the groups have a non-null pairwise intersection it is clear that mutual exclusion is guaranteed. The protocol assumes a *fault-free* complete network with FIFO channels and each process is assigned a unique Id.

The main focus of the protocol is the optimal creation of the groups such that each process is assured fairness as well as equal responsibility in the computation. The method used to determine the groups is equivalent to finding a finite projective plane of n points. From this it is deduced that the optimal size of each group is $\sqrt{n}$. This implies that each process needs to receive permissions from only $\sqrt{n}$ processes

before it can enter *CS*. A process is allotted one group to which it belongs and from which it must gain permissions to go into *CS*. Entering *CS* a process sends time-stamped requests to all the members of its group except to itself. The time-stamping method is a logical clock mechanism defined by Lamport [37] in 1977. Each process locks itself for the first request that it receives while it is yet unlocked (including its own request). The process then returns a *locked* message to the requesting process. In that case the process is locked for the requesting process.

If the process is already locked for another process it then buffers the request in a queue of outstanding requests. A test is performed to see whether the current locking request is the eldest request, i.e. the older the request the higher the priority. This ensures fairness. A request can be preempted by another request of higher priority if the lower-priority request has not yet succeeded in locking all members of its group or has not yet entered the *CS*. If two requests bear the same time-stamp then the Id's or the processes are used as the tie breaker in which case the process with the smaller Id has higher priority. A request which contends with another one of higher priority is replied to with a failed message. In that case the process yields to all other contending requests of higher priority.

The yielding process releases all the locks it has on any member of the group to which it is yielding. The released node undoes the locks and then lock itself for the eldest request in its queue. It then sends a locked message to the node for which it has locked itself. Each requesting process enters *CS* upon receiving locked messages from all members of its group. After leaving its *CS* the process sends release messages to all members of its group. A process which receives a release message deletes the releasing process' request and then locks itself for the eldest request in its queue (if any). It then sends a locked message to the process for which it has locked itself and await either a release message or a yielding one. The protocol has a global storage complexity of $O(n^2)$ process Id's. The protocol deals with fault tolerance by considering the removal of failed processes but the fault model is incomplete in that it does not deal with all possible process failures.

The groups from which the processes must obtain permission are of fixed size and any process failure would skew the groupings and even lead to a deadlock. Although process failures are considered, no recoveries are discussed. No realistic fault situations are presented and the algorithm needs an additional mechanism to reset the groups in cases of node failures. The protocol is highly symmetrical in the sense that all nodes execute the same code, but there is reference to the node Id's.

3.3.2 The Sanders Protocol

The protocol of Beverly Sanders[58] is similar to that of Maekawa in that it also uses intersecting groupings from which requesting nodes must receive permissions. This, however, is one of the first algorithms to explicitly use the notion of an *information structure* to solve distributed mutual exclusion. An information structure is composed of two parts: one part contains all processes from which permission must be sought. The other contains processes to be informed upon leaving *CS*, is defined. Each process also maintains a set of processes about which it keeps information locally. This is referred to as the *status-set*. The idea is similar to that of Maekawa in the sense that a fixed set of intersecting groups are defined and each process is associated to one group from which it must receive permissions. The main difference is that in Maekawa's protocol a process informs the same group that granted it permission when it leaves *CS*. In this protocol the group from which permission must be obtained is different from the group that must be informed upon exiting *CS* for each process. The protocol has a worst case message complexity of $(a + 2b + \sum_{j \in R_i} r_j)$ in the absence of faults, where a is the size of the *Inform-set* (the set of processes that are informed after exiting *CS*), b is the size of R_i , the *Request-set* of process i (the ones that granted i permissions), $r = 1$ if process i keeps information about process j , or $r = 4$ otherwise. The protocol has a global storage complexity of $(n - 1)^2$ request messages, where each request message contains an Id of the requesting node and a time-stamp. The fault tolerance discussed in the paper uses a fault model where orderly faults occur. Time-outs and continual auditing methods are suggested as methods of making the protocol resilient.

3.3.3 The First Singhal Protocol

Towards the end of the paper, Sanders considers how the static information structure in her algorithm can be made dynamic. This idea was developed by Mukesh Singhal[61] and he published a protocol based on a dynamic information structure, i.e. a structure that is continually evolving to reflect the current global status of the system. The advantages of this approach is that the processing of the algorithm does not remain the same at all times. Less processing is done if less processing is needed to perform mutual exclusion. A typical case is one in which some processes are not interested in entering C'S. In that case it is not necessary to include the uninterested processes in the computation, hence less work needs to be done. A static information structure algorithm would compute in the same way even though this would not be necessary in all cases. The main focus of this algorithm is the construction of a *dynamic-information-structure (DIS)*. The paper deals with the two questions:

What is the minimum set of sites to request permission from ? and

How does a site decide which sites constitute the minimum set at any time ?

The protocol is similar to that of Sanders except that the information structure is dynamic, only the request set is explicitly defined and the inform set is formed by incoming requests from other nodes. It is also assumed that each site has a unique Id. The protocol has a worst case message complexity of $O(3^{\frac{(n-1)}{2}})$ in the absence of faults. It also has a global storage complexity of $O(n^2)$ node Id's and it does not consider fault tolerance. No sites may crash. This restriction violates the basic requirement of mutual exclusion as stated in Chapter 1.

The algorithm also has a flaw: there is a *suspended-request-queue* which is never emptied. This means that the queue continues to grow unboundedly, this cannot be intended because it is an error. We feel that the error is due to an oversight.

3.3.4 The Rangarajan-Tripathi Protocol

Sampath Rangarajan and Satish Tripathi[51] published a protocol whose main focus is fault tolerance. The paper gives an extensive discussion of resilience. The protocol is permission based and uses the grouping method similar to that used by Maekawa[39]. The main difference to the Maekawa protocol is that this defines the groupings as follows:

The n processes in the system are divided into m subgroups each of size $\frac{n}{m}$. Out of these groups quorum groups, each composed of $\sqrt{\frac{n}{m}}$ processes, are made. If a square-root is not an integer then the nearest square (whose root is an integer) is used, this yields some dummy processes which serve no other purpose than filling up the groups. Every pair of quorum groups intersect at one subgroup. Each process is associated to a quorum subgroup and it is from that quorum that it must receive the permission of the majority before it can go into CS. Each process can only grant permission to just one process at a time. Request are time-stamped to allow for the ordering of events. We assume that node Id's are used as final tie-breakers since this is not made clear in the paper. Each subgroup is deemed to have granted its permission if a majority of processes in it grant their permission. If all quorum groups grant their permission then a requesting process can safely enter CS.

Note: The protocol does not consider the case when more than one simultaneous requests are made, this gives a wrong impression on the message complexity. Under such conditions where requests are generated one at a time, such that at any one time there exists only one request (globally) in the system, the protocol has a message complexity of $O(\sqrt{N \log N})$. The protocol is restricted to one request at a time and, as such, there is little need for local storage except that of the quorum groups.

The only fault tolerance discussion in the paper is concerned with maintaining functional quorum groups in spite of node failures. There is no clearly defined fault model in the protocol. The following questions are not addressed:

- How does a process handle a request ?
- What does a process do when it leaves CS in order for another to enter ?

- What types of faults can occur ?
- How does a process react to the failure of another processes ?
- Are processes allowed to recover after they have failed ?
- How many messages are needed to rectify a faulty set up ?

All the above questions need to be addressed if the protocol is to be considered for simulation or implementation.

3.3.5 The Woo-Newman-Wolfe Protocol

This algorithm by Woo et al.[68] uses a Huffman tree structure to optimize the number of processes from which permission must be received before a process can enter CS. The algorithm is similar to the Maekawa one in the sense that the processes have intersecting groups from which they must obtain permissions. The main difference is that each group for each process is composed of the nodes that lie along a path from the node as a leaf to the root of a dynamically constructed *Huffman* tree. This means that each process must obtain permissions from a maximum of $h \leq \log(n)$ other processes. The root node is the one that grants permission, the other nodes along the path grant or deny permission by passing on or blocking a request respectively. The main focus of the paper is to show how the Huffman encoding can be mapped to the CS demand of the processes in the system and thereby reduce the number of messages needed for each CS. Some aspect of fault tolerance is mentioned. The protocol assumes a complete network, *FIFO*-channels and a foreknowledge of the probabilities of all the processes' CS-demand. Each node has a unique Id.

A Huffman tree is created using the probabilities of the processes entering CS. Processes with higher demand are placed closer to the root of the Huffman tree. When a process wants to go into CS, it sends a request to its parent in the tree. Each process handles requests in a *FCFS* manner. A process that receives requests from its children passes on the first request that comes after sending an acknowledgement

to the sender of the request. If a process was requesting on behalf of its child then it passes on the received acknowledgement to the child. It buffers a second request thereby effectively blocking it. Since each internal process has only two children, each process does store at most one request. When the root receives a request it grants a permission to the requesting node if it has not yet granted a permission to another child. If the root has already granted a request then it sends a standby message to the requesting process.

It is clear that the number of messages needed for each CS is proportional to the height of the Huffman tree ($h \geq \log(n)$) and the position of a process in the tree, i.e. a process closer to the root causes less messages than a process at the bottom of the tree. In the absence of faults each CS request needs h -requests, h acknowledges, h -standbys, h -permissions and h -release messages. This gives $5h$ messages for each request in the worst case. In the worst case $h = \log(n)$.

Each node stores the Ids of its parent and both children, a boolean variable to show whether a request has been made or not and its own Id. The protocol deals with single entries into CS . When processes have dynamic probabilities of CS -demand or when all processes have the same probabilities then the protocol does not perform equally for each process. Fault tolerance is not considered in this protocol.

3.3.6 The Raymond Protocol

The previous discussion centered on protocols that use the idea of quorums or groups from which permission must be obtained. We present other permission based protocols that use a different technique. The Ricart-Agrawala[57] is one of the earliest permission based algorithms that did not use groups or quorums explicitly, in it each process broadcasts its request to all other processes and awaits the granting of permission by all other processes. Priority is dependent on Lamport clocks and node Id's. The algorithm has a message complexity of $2(n-1)$ and uses $n \log_2(n-1)$ global storage.

The algorithm of Raymond [52] extended Ricart and Agrawala's algorithm to allow for multiple (k) entries into CS but retaining the same message complexity of $2(n - 1)$. In this algorithm, a node broadcasts its requests to all the other nodes and may enter critical section if it secures permission to do so from at least $(n - k)$ other nodes, i.e. it goes into CS as soon as the first $(n - k)$ replies arrive. A node that is currently executing CS defers replies to other requests until it exits CS . The algorithm assumes a complete network. It also assumes that no messages can get lost. Each node that wants to go into CS broadcasts requests to all the other nodes. Each node, upon receiving a request responds with a reply message if it does not want to go into CS itself and it is also currently not in CS . Otherwise it defers the reply until after it has exited its own CS . Each request has a sequence number which is similar to the Lamport clock values[37], with this sequence number it is possible to order all the requests in the system. A request with a lower sequence number has a higher priority than another with a higher sequence number. Since it is possible for a node to receive more than one request from the same node¹ a process buffers the requests from each node in a list but sends a single reply to each requesting node after exiting CS . The reply specifies the number of requests for which the reply is being sent. Each node keeps track of the outstanding replies yet to be received from each node.

The protocol has a global storage complexity of $n(n - 1)$ requests and deferred replies. Each request contains an Id and a time-stamp. The protocol uses $(n - 1)$ request messages and $(n - k)$ reply messages for each process to go into CS . The total number of messages incurred per CS is $2(n - 1)$ since each request message generates a reply message, whether it is deferred or not. When the demand for CS is high then the number of total messages decreases because a process can send bundled reply messages to those processes that have sent it more than one request before a reply could be sent to them. It is mentioned in the paper that the reduction of messages

¹ This happens when a node has been receiving enough permissions to go into CS from other nodes besides the deferring one

by bundling up deferred reply messages is negligible. Therefore the protocol has a message complexity of $2(n - 1)$. Fault tolerance is not considered in this protocol.

3.3.7 The Bar-Noy et.al. Protocol

This protocol by Bar-Noy et al.[4] also uses a similar technique to that of Ricart and Agrawala and it explicitly addresses fault tolerance. This protocol allows for multiple entry into *CS*. The main focus of the algorithm is fault tolerance and the definition of an optimal priority rule according to which processes wanting to go into *CS* are granted permission to do so. The number of nodes that can fail in the system is bounded. Each process maintains a counter which reflects the number of times it has accessed *CS*. All processes that are currently interested in entering *CS* keep track of the number of times every other process has entered *CS*.

Each process wanting to go into *CS* sends requests to all the other processes and await acknowledgements. Each requesting process is ranked by every other process according to the number of times it has already entered *CS*. If one of the incoming acknowledgements indicates the existence of a request of higher priority then the process aborts its attempt to secure permission to go into *CS* and starts all over again, other wise the process broadcasts a second time and then goes into *CS* as soon as its rank is k or less. At most k processes may be in *CS* simultaneously. The protocol assumes *FIFO* channels. This is needed in order to correctly associate outgoing messages with incoming acknowledgements. *FIFO* channels are not needed if a mechanism to keep track of messages is available. The number of processes that can fail is bounded and each process knows the number of processes in the system. A second variation of the algorithm is presented in the paper which can cope with Byzantine faults. In the absence of faults the algorithm needs at least $5(n - 1)$ messages per *CS* in the best case. The storage complexity of the algorithm is $O(n^2)$. This protocol is optimal when one is considering highly fault tolerant systems but not when low message complexity is desired.

3.3.8 The Second Singhal Protocol

The protocol of Mukesh Singhal[62] also uses the notion of a *dynamic information structure* similar to that used by Beverly Sanders above. An information structure composed of two parts, one part containing all processes from which permission must be sought and another containing processes to be informed upon leaving *CS* is defined. As soon as a process gains permission of all the processes in its request set it enters *CS*, after it has left *CS* it then informs all the processes in its *inform-set*. The algorithm also makes use of the state information to continuously update its information structure. A requesting process that receives a request message from another process of higher priority adds the requesting site to its *request-set* and respond with a *request* message right away, otherwise it adds the requesting site to its *inform-set* and proceeds. If a process is already in *CS* when any request arrives then it just adds the requesting process to its *inform-set*. A non-requesting process that receives a request from another process after it has granted permission to another process adds the requesting process to its *request-set* and send it a permission. A process that receives a reply or permission message removes the sender process from its request set. The protocol has a message complexity of $3(\frac{n-1}{2})$ and a global storage complexity of $O(n^2)$ node Id's. A timeout mechanism is used to make the protocol resilient against message losses.

To make the protocol resilient to site failures each site removes a failed process from its request set, thus removes it from the system. This is what is intended but the protocol has a serious flaw. Each process that detects a failure just removes the failed process from its *REQUEST-set*. This is all that is described in the paper. This means that if the failed process is in the *Inform* set then it remains there. According to **Step 3** of the algorithm (Part C.) each process automatically moves all processes in its *Inform* set into its *Request* set. This means that a deadlock is inevitable because the process will now be requesting from a crashed process, this leads to a deadlock.

3.3.9 Summary

We noticed that none of the quorum or group based protocols that are also permission based has been generalized to allow for multiple entry into CS. The definition of a universal priority rule is of utmost importance in permission based protocols, some protocols give priority to those processes that have entered CS less than others while some give priority to processes that requested for CS before others.

We have seen that in permission based protocols several techniques can be used. Time stamping is an alternative to the *first come first served (FCFS)* principle and each protocol can optimally utilize one or the other. Where time-stamps are used and there is the possibility that more than one process have the same time-stamp (Lamport clocks are used), the Id of each process then plays the role of tie breaker according to the definition. Among the protocols that use the voting scheme some use the weighted voting, method while others use the majority voting method. In the weighted voting processes are assigned differing voting weights such that there can only be one majority in the system at any time but each majority group can have a different number of participating processes. This means that a requesting process could receive permission from a majority of processes after replies from just a few other processes. The majority voting method is one where each site holds only one vote and the above mentioned quorum or group protocols use this method.

3.4 Token Based Protocols

In token based protocols a token (a special object or message) is passed on from one process to another (either upon request or as a matter of course) and a process must be in possession of the token before it can go into CS. The improvement of token based protocols has now focused on providing quicker token location and transfer methods. Some tokens are just symbolic and some are complex data structures. Tokens can get lost and safeguards against deadlock in such situations have to be provided.

3.4.1 The Suzuki-Kasami Protocol

In 1985 Suzuki and Kasami[65] published a token based protocol for mutual exclusion. The algorithm is simple to understand and is based on the idea that a requesting process broadcasts its request to all the other processes and then waits until it receives a special privilege message, the *token*, before it can go into *CS*. The protocol assumes a complete network.

There is a token which initially resides at an arbitrary process. A process that is in possession of the token can enter *CS* at any time. Each process keeps a Lamport clock counter which is updated in the same way proposed by Lamport[37]. A requesting process broadcasts its time-stamped request to all the other processes. Each request is composed of a time-stamp and a process-Id.

The token is a queue containing all processes that have made requests. The entries are made in a FCFS manner. A token-holding process appends incoming requests to the back of the queue and it sends the token to the process at the head of the queue after it exits *CS*. A process that receives the token removes its entry from the head of the queue before entering *CS*. Fault tolerance is not considered in this protocol.

The protocol has a message complexity of $O(n)$ in the absence of faults. The protocol can be modified so as to bound the clock counters and to allow a distinction between different requests by the same node. This modification causes the protocol to have a message complexity $\leq n + (n - 1)$. The protocol has a global storage complexity of $O(n^2)$ entries where each entry is made up of a time-stamp and a node Id.

3.4.2 The Helary et.al. Protocol

The protocol by Helary et al.[32] is similar to the Suzuki-Kasami one except that no topological assumptions are made. Only connectivity assumptions are made. This is the main difference between the two algorithms. A process is only aware of its neighbors and every process has its own Id and messages are transmitted in their

integrity. A process that wants to go into *C/S* broadcasts its time-stamped request to all its neighbors and waits until the token arrives then it goes into *C/S*. Each request contains the originator, the time at which the request was made, the Id's of all processes that have seen the request already and the immediate sender of the request, i.e. the process that relayed it. A process that receives a request while it is not holding the token broadcasts the request to all processes whose Id is not yet in the request, i.e. those processes that have not yet received the request. It does this after adding the missing processes' Id's into the request. The process then appends the request to its list of pending requests. The list of pending requests also shows from which neighbor the request was received.

A token holding process sends the token to the process at the head of the token queue through the neighbor from whom the corresponding request came. This is done after the process has exited its *C/S* (if it was in *C/S*). The process first removes the request from the head from the token. The request at the head of the token queue is the eldest request because new requests are always appended to the back of the queue. The request shows the path through which it arrived at the token holder from the originator and the path is used to send the token. The token shows the processes to which it must be sent and in what order. When a process receives the token when it is the current destination then it enters *C/S*, any other process just passes on the token towards the destination according to the request path.

In the absence of failures the protocol has a worst case message complexity of $2(n - 1)$. The protocol has a global storage of $O(n^2)$ node Id's. Fault tolerance is not the main focus of the paper but fault tolerance is mentioned. In the case of a complete network the protocol does not offer a better solution than that of Ricart and Agrawala[57] and in the case of a tree topology the protocol does not have a logarithmic message complexity.

3.4.3 The Nishio-Li-Manning Protocol

The paper by Nishio et al.[49] discusses a resilient mutual exclusion algorithm which uses a circulating token based on the time out mechanism. The algorithm is based

on the Suzuki-Kasami algorithm[64] of 1982. A token-loss detection and a token regeneration procedure are defined to make the algorithm resilient. The main focus of the protocol is fault tolerance. The fault model that is used considers three different kinds of faults, the *process-failure*, the *communication-controller-failure* and the *communication-link-failure*. The basic idea is that a requesting process broadcasts its request to all the other processes. The token holder updates the token, which is a list of pending requests. The priority is given to the process which has used CS the least number of times and is nearest to the current token holder. The algorithm assumes a complete network where each process has a unique Id. It is also assumed that a process and its communication controller are disjoint such that one can fail while the other is still functional. Obviously if the communication controller fails then the corresponding process cannot communicate. It is further assumed that a site with a failed process can receive and send messages and the token. Communication controller failure does not cause any loss of information. Any failure is followed by recovery in finite time. The process and communication controller pair at each site are aware of each other's state.

Each site holds an array of length n into which it records the number of times each site is known to have made a CS request messages. Each time a site makes a request it increments its request counter by one and this counter is also part of the request. The count of CS-accesses for each process is always incremented by one each time a process leaves CS. A process wanting to go into CS broadcasts its request to all the other processes and then awaits the arrival of the token. A process that does not receive the token within the expected time initiates an investigation about the possible loss of the token. Each token generation has an age (an index) and only the process proposing to regenerate a token with the highest age succeeds if indeed the token is missing. Every non-faulty process and controller participate in the regeneration of a new token. This ensures the existence of at most one token in the system at any time. Each site also has a variable which shows its age. A previously failed and now recovering process inherits the age of one of its neighbors. A process that receives a token with a lower age than its own simply deletes it.

The method of time-out is used to detect failed links, processes, communication controllers and lost tokens.

The message complexity is not considered in the protocol and the authors acknowledge that the message complexity is high hence they do not even go into determining it. The protocol has a global storage complexity $\leq n^3$. The paper provides a method for introducing general fault tolerance. The principles that are applied to make the algorithm fault tolerant are applicable to other algorithms.

3.4.4 The Chang-Singhal-Liu Protocol

This protocol by Chang and Singhal and Liu [12] is a dynamic token-based distributed mutual exclusion protocol also similar to the Suzuki-Kasami one. Each process has a set of other processes to which it sends its request when it wants to enter *CS*. This set is dynamically updated in the course of the computation so as to reduce the number of processes to which each one must send its requests for *CS*. Reducing the number of processes in this way also reduces the number of messages exchanged for each *CS*. Each process has a list of processes that it knows to be potential holders of the token. In this case, the token is a queue of requesting processes. There are different ways of initializing this list. One way is to make every process append all processes whose Id is greater than its own into the list. Each process that wants to enter *CS* broadcasts requests to all processes in its list of potential token holders, re-initializes the list to empty and awaits the arrival of the token. When the token arrives the process goes into *CS*. Upon leaving *CS* the process adds to the token-queue all outstanding requests that have arrived but are not yet recorded in the token. All processes with pending requests, as shown by the token queue, are added to the site's set of potential token holders.

If the token-queue is not empty then the process removes the first entry from it and sends the token to the process whose entry has just been removed from the token. Sequence numbers which are updated according to Lamport clock rules are maintained at each process. When a process receives a request and the requesting process is not part of the known potential token holders then it is appended to

that list. Note that processes not interested in CS are automatically removed from competition and this reduces the number of total messages. The protocol has a worst case message complexity of $O(n)$ and a global storage complexity of $O(n^2)$ process Id's. Fault tolerance is not considered in this protocol.

3.4.5 The Srimani-Reddy Protocol

This paper by Srimani and Reddy[63] discusses a generalized algorithm that allows for a maximum of k processes to enter CS at a time. Again, this algorithm is based on the algorithm by Suzuki and Kasami[64]. A process wanting to go into CS broadcasts its request to all the other processes and then awaits the arrival of the token. There exist k tokens in the system and any process must be in possession of a token before it can enter CS . Each token is a 2 dimensional array which holds the sequence number and an Id for each requesting process. The current token holder always sends it to the token at the head of the queue after it has exited CS and its own entry has been removed from the token. The protocol assumes a faultless complete network. Fairness is ensured by maintaining a number showing the frequency of CS entry by each process, the higher the frequency, the lower the priority. A process wanting to enter CS does so upon receiving a token, otherwise it awaits the arrival of pending messages (if any) before sending request messages to all the other processes. The request contains the Id of the process and an index showing how often the requesting process has already been in its CS .

A requesting process eventually receives all k tokens although it enters CS upon receiving the first one. A process that receives a token updates it according to its locally stored information after using it (if it needs it) and before sending it to other requesting processes. If there are no known requesting processes then the token remains at the process that had it last. This means that the information held by the tokens is synchronized to the information held at different processes. This prevents starvation, deadlock and incorrectness. It ensures that at any time, a maximum of k processes are executing their CS . The protocol has a worst case message complexity of $(n + k - 1)$ and a global storage complexity $\leq n^3$ node Id's.

Fault tolerance is not considered in this protocol. We conclude this study of previous works by discussing some token based protocols that use a combinatorial structure such as a tree or a heap to achieve mutual exclusion.

3.4.6 The Satyanarayanan-Muthukrishnan Protocol

Satyanarayanan and Muthukrishnan[59] published this tree structure token based algorithm in 1992. The protocol uses the *least executed* priority criterion and it is based on the token-based Raymond[53] protocol. A logical spanning tree is constructed in the network where each process is aware of the tree structure and its neighbors in it. Each process maintains a queue into which it appends incoming requests and numbers showing how often each process has executed CS. The queue entries are sorted according to how frequently a process has executed CS. The processes that have executed CS less times come first. The process assumes a complete network with *FIFO* communication channels. There is a process (the root) in the system designated as the leader and it initially creates a token. Each process wanting to go into CS sends its request towards the root of the spanning tree or towards the path where the token has been transmitted (if known). Each process that receives a higher priority request appends it to its local storage and then passes it on towards the token holding process (if it does not hold a token itself). If there are no pending requests then an incoming request is handled as a higher priority request. A process (not token holder) that receives a lower priority request just stores it in its local storage and does not pass it on because it is handled after the higher priority ones. When the token holder receives a high priority request it sends the token towards the requesting process. A process that receives the token first updates the token using its local pending-requests-queue and then deletes each entry from the local queue. It then forwards the token towards the process whose entry is at the head of the token-queue. If its own entry is at the head of the queue it then goes into CS after removing it. Upon leaving the CS the process increases its usage count and sends the token towards the requesting process at the head of the token queue. In the absence of faults the protocol has a worst case message complexity

of $O(2D)$ where D is the diameter of the spanning tree. The protocol has a global storage complexity of n^2 node Id's. Fault tolerance is not considered.

3.4.7 The Naïmi-Trehel Protocol

This paper by Naïmi and Trehel[48] is an in depth analysis of the algorithm published previously by the authors. The algorithm was the first *dynamic* logical tree solution to the problem of mutual exclusion. This paper illustrates the application of different combinatorial structures to solve mutual exclusion. The method is based on the principle of path reversal. Each node maintains information about two other nodes, one to which it issues permission and another to which it sends *CS* requests. The method uses that fact that the combinatorial structure of ordered trees maps directly into different priority queue structures such as the *binary heap* and the *binary tree*. By using the reference to these structures the cost of path reversal can be reduced considerably. The token contains requests of processes in the order in which they arrived at the current token holder. There is a global structure which is also maintained in the system, a path. The path leads to the node that is known to be the last one to request *CS* in the system. Every new request by a process makes the requesting process become the last member of the path. The path is the route taken by the token to service *CS* requests. The creation of a new extension to the path is a logical transformation of path reversal and whenever the token traverses an edge the direction of the edge points in the direction of the token. The process at the head of the token queue is the one which is currently in *CS* and its entry is removed before it sends the token to the new head of the token as the token travels towards the end of the path. The protocol assumes a complete network and reliable communication links. All the processes in the network are arranged in a logical tree and the current root of the tree is the beginning of the path and is also the holder of the token. As the token moves from a requesting process to another the path becomes truncated. If a process receives a request for the first time it designates the sender of the request to be the next receiver of the token after itself, if it is also requesting. If the receiver of the request is not requesting then it passes on

the request along the path that it knows to lead to the token. A requesting process that receives other requests passes them on towards its successor in the token path. Every new request thus makes the requesting process become the new end of the path. A mechanism is designed to prevent the *chase condition*, i.e. one in which a request chases endlessly for the tail of the path which is continually being shifted at the same rate. A process which receives a request that is of higher priority yields to that request if it is not already in *CS*. The protocol has a worst case message complexity of $O(n)$ and a global storage of $n \cdot \log(n)$ bits. Fault tolerance is not considered in this protocol. The paper provides a waiting time and a path reversal cost calculation, the former, though present in all algorithms, is rarely handled, the latter is relevant to the method used.

3.4.8 The Bernabeu-Auban et.al. Protocol

This Bernabeu-Mustachi[5] protocol is a tree structure token based protocol with a logarithmic message complexity. The general principle of the protocol is that the nodes of the system are initially arranged in a logical tree (*star*) with the token holder as the root (*center*). Each process initially knows along which path the token is. A requesting process sends its request to its parent in the tree. A requesting process or one that has already sent a request buffers a second request. A process that is not requesting and has not received any pending request passes on the request to its parent. The token holder (which is also the root of the tree defers a response until it has exited its own *CS* (if it was in *CS*). It then sends the token towards the first requesting process. The path along which the token travels is reversed so that the requesting process is the new root of the tree. The method that is used is the path compression method. The protocol assumes a complete network, no message losses and no process failures. Each node maintains a number indicating how many times it has previously requested *CS* and another one indicating how often the process has executed its *CS*. Each request indicates how many times the sender has already executed *CS*. Each process also shows which messages are in transit between itself and every other nodes. A requesting process sends its request to its parent in the

logical tree (if it does not hold the token). If it holds the token then it enters *CS* if there are no pending requests of higher priority. A process that receives a request passes it on to its parent if it is no longer the root. This means that the previous token holder has already passed on the token to another requesting process. If the token holder is currently in *CS* then it defers the response until it exists *CS*. When a process sends or relays the token then it makes the process to which it has sent the token its new parent in the logical tree. The tree is thus dynamic and linked to the movement of the token. When concurrent requests are made then a forest is the result. Since each process maintains a list of indexed messages concerning current communication between itself and every other node these lists are used to bring a total ordering of events in the system. The protocol has a message complexity of $2 + 3\log(n)$ in the absence of faults. the protocol has a global storage complexity of $O(b)$ where b is the number of messages transmitted between the nodes in the *underlying* computation. Fault tolerance is not considered.

3.4.9 Summary of Token Based Protocols

From the brief study of token-based protocols above it is clear that token based protocols are mainly concerned in minimizing the number of messages needed to locate the token. Most of the algorithms allow only for single entry into *CS*. Judging from the message complexities, it seems that the tree-based (or the combinatorial-structure-based) approach is optimal for token based protocols. Fault models in token based protocols usually involve the detection of the loss of a token and its regeneration. Normally age counters in the form of sequence numbers are used in token based protocols to ensure non-starvation and fairness. These are normally based on Lamport clocks. There are other token based protocols that require the construction of a ring topology where the token traverses the ring and they perform better (in a ring) than protocols designed for arbitrary topologies.

<i>Alg. Ref.</i>	<i>Author</i>	<i>Message Complexity</i>	<i>CS Entry</i>	<i>Fault Tolerance</i>
[39]	Maekawa	$5\sqrt{n}$	Single	No
[58]	Sanders	$a + 2b + \sum_{r_j \in R_i} r_j$	Single	Yes
[61]	Singhal	$3\frac{(n-1)}{2}$	Single	No
[68]	Woo et al.	$2(n-1)$	Single	No
[52]	Raymond	$2(n-1)$	Multiple	No
[4]	Bar-Noy	$5(n-1)$	Multiple	Yes
[65]	Suzuki et al.	n	Single	No
[32]	Helary et al.	$2(n-1)$	Single	Yes
[12]	Chang et al.	n	Single	No
[63]	Srimani et al.	$n + k - 1$	Multiple	No
[59]	Satyanara et al.	$2D$	Single	No
[48]	Naïmi et al.	n	Single	No
[5]	B-Auban et al.	$2 + 3\log(n)$	Single	No

Table 3.1: Worst-Case Complexities of Some Algorithms (in absence of faults)
 D =diameter of spanning tree, a = InformSet, b =RequestSet, $r=1,4$

3.5 Summary of Studied Literature

We have looked at both permission-based and token-based protocols. Table 3.1 shows the worst-case message complexities of some of the protocols in the absence of faults.

Chang, Singhal and Liu[12] discuss the relationships between the two areas in their paper. They make the observation that for every non-token based protocol (not involving site locking) there exists a token-based algorithm (using broadcast communication) with the same initialization of request sets and similar rules to update the request sets. They cite the relationship between the Ricart-Agrawala[57] and the Suzuki-Kasami[65] algorithms as a typical illustration of this relationship. We observed the following:

- Each method has different priority rules. Permission based protocols tend to use time-stamps and the process Id's to make a total ordering of events in the system whereas token-based protocols seem to use similarly structured sequence numbers to achieve the same thing. The priority rules are not limited

to one type of protocol, rather, some are used in one type of protocol more than in another. Some protocols adhere strictly to the principle of *FCFS* while others solely depend on time-stamps and process Id's.

- Token based protocols can be generalized to allow for multiple entries into *CS* more readily or using less complicated methods than permission based ones. To generalize permission-based protocols in this way augments local storage.
- The definition of common measuring standards for protocols is an important aspect. We believe that another measure of the quality of a protocol is the ease with which it can be generalized to accommodate multiple entries into *CS*.
- Generalizing for arbitrary topologies almost always causes loss of efficiency and most mutual exclusion protocols use a complete graph. Another factor that diminishes message complexity and is thus mostly avoided by most protocols is *fault tolerance*.
- There is a larger ratio of permission-based protocol that take fault tolerance into consideration than token-based ones. Recently more research has been put into permission based methods than in token based ones.
- Both permission and token based methods seem to benefit from the application of dynamically managed structures such as *trees* and *heaps*.

In the following chapters we present a token-based protocol. The protocol is optimal both in terms of message complexity and in terms of local storage. Fault tolerance is also an integral part of the protocol. Furthermore, the protocol can be generalized to allow for multiple entries into *CS* by changing just one parameter with no loss of efficiency and resilience.

Chapter 4

An Efficient Distr. Mutual Exclusion Protocol

4.1 Introduction

In distributed computing *a priori* centralization of control is normally undesirable because this works against the principles of local autonomy for the system processes. In this section we describe a dynamically coordinated protocol for managing a highly fault-tolerant critical section in a complete network. Faults such as site crashes and message losses can take place at any time and at any site including the one currently in the critical section. The protocol ensures the standard requirements for mutual exclusion protocols of *Exclusion*, *Fairness* and *Liveness*. The method used in this chapter is token-based and does not use vector clocks. The method is based on what we call the *dynamic coordinator*. The main goals of this approach are reducing communication overhead by centralizing certain aspects and enjoying the advantages of concurrent processing by having a dynamic structure. In comparison to other Distributed Mutual Exclusion protocols this approach fares quite well in terms of number of messages needed for each CS-entry, local storage and fault tolerance. Some distributed mutual exclusion protocols will behave differently if the demand for CS is not evenly distributed and some processes even have the potential for deadlock if some processes do not request CS for prolonged periods[12]. Some protocols need all their processes to know the number of processes available in the

system, our protocol has none of these constraints. Any number of new processes can be added on to the system as long as they each have a unique Id. Furthermore, this approach has an added advantage of being simple and straightforward.

4.2 Assumptions

4.2.1 Basic Assumptions

As we mentioned on Page 1, any fault that may take place is detectable. We also mentioned that the message that is sent by a process in response to failure detection can never precede the failure detection by the receiving process. As we shall see later, this condition can be relaxed without compromising the quality of the protocol. We assume that a mechanism whereby every node in the system becomes aware of failed nodes exists.

The underlying topology is a complete graph. All the processes of the system are connected by bi-directional links and can communicate directly. As we shall see later, this assumption can be relaxed as link faults are allowed. As a consequence it might be possible for the network to become partitioned.

Each neighbor is aware of all its neighbors on its incident links. Each neighbor is aware of its neighbors and of whether they are up or down.

Every process in the system has a unique Id. Each process can identify from which neighbor a request is coming and keeps track of faulty neighbors. Absence of a unique Id implies an anonymous network. Having an anonymous network means that symmetry cannot be broken. Inability to break symmetry implies the inability to elect a leader-process and this in turn implies that a mutual exclusion coordinator cannot be agreed upon.

4.2.2 Message Transmission Assumptions

In the absence of failures, transmitted messages are received in their integrity.

Each processor has a message buffer (unordered set). A processor is able to broadcast

messages to a selected set of its neighbors (possibly all of them). A non faulty processor will eventually be able to communicate with another non-faulty processor within finite time except in the case of a complete network partition.

Messages do not get lost in transit. When a process sends a message to another process and that node crashes during or soon after the event then the sender process is able to detect whether the message has arrived at the receiver or whether it got lost. If the message is lost in transit then the sender puts itself in a position as if it never had sent the message. When a process crashes then all the links that are incident to it are flushed and all messages that may have been in them are lost. It can be shown that this restriction can be relaxed without compromising the protocol's quality.

Redundant messages are not processed. If a process receives a message from a process which then crashes (just before the message is received or just after sending the message then, except for the standby message and the permit, it will ignore the message.

Note: We do not assume FIFO channels. Further note that some mutual exclusion protocols do require FIFO channels and thus are less general[68, 67].

4.2.3 Fault Assumptions: Fault Model

Node failures can be viewed as occurring at the highest level of abstraction, the application level. There are other faults that can occur within the network such as link failures. Such faults pertain to lower levels of abstraction and since we assume a reliable underlying structure (complete with routing mechanisms and group membership methodologies) we do not handle link failures.

Nodes can fail by crashing. As mentioned in Chapter 1, by definition, the mutual exclusion problem allows for process failures and the behavior of a process outside of CS does not affect any other processes that want to go into CS . It also follows that if a centralized coordinator is being used (as in our protocol) then it cannot prevent a process from entering CS when no other process is currently in CS and the process is next in line for CS . Our protocol can tolerate up to $n - 1$ process

failures, including the one that is currently in *CS*. This is a further generalization compared to protocols that allow for very limited (sometimes none) process failures. Permission granting protocols are more susceptible to this kind of limitation. Our protocol does not cover Byzantine faults. To date we have yet to see a protocol which handles Byzantine faults completely in distributed mutual exclusion. Our protocol does not handle Byzantine faults.

A crashed process can recover and rejoin the computation. We allow for crashed sites to be able to recover and rejoin the computation.

No preferential treatment of some processes. As described below, a process which fails while in the *CS*, upon recovery, has to rejoin the tournament tree and compete for *CS* in the normal way. This ensures Non-starvation and Fairness.

4.3 The Protocol: An Informal Description

At any one time there is exactly one process designated as the Mutual Exclusion Coordinator (*MEC*). The *MEC* is the only one that issues *CS*-permissions and the only one that can regenerate a token when the current token gets lost. Note that in this protocol *token*, *PERMIT* and *permit message* are synonyms.

Every process is potentially capable of becoming the *MEC*. There is a predefined mechanism for succession to the *MEC* (the *MEC succession chain*) in case of *MEC* failure: there is a total ordering (e.g. based on Id's) of the processes and a dynamic cyclic list based on this ordering, where the first element follows the last. At any time, the list is composed only of the non-faulty processes: processes that fail are automatically removed from the list, and recovered processes rejoin the list in the appropriate positions (i.e., according to the ordering). Initially the process with the smallest Id is the *MEC*. If the *MEC* fails, the process following it in the list will become the *MEC*. Note that it is possible for the process that is currently in *CS* to become the next *MEC*; this can occur because entering into *CS* is done at the application level and independent of any action taken at the lower level where the mutual exclusion protocol operates.

4.3.1 Scenario in the Absence of Faults

We now describe the workings of the protocol in the absence of faults; after that, we will discuss the scenario when faults can occur.

Each process wanting to enter *CS* issues a request to the *MEC*. The request is inserted by the *MEC* into a queue of all requests (including its own) not yet granted. The *MEC* replies either with the permit or with a standby message. The permit is granted if no process is currently in its *CS* or if the request is at the head of the *MEC*-queue. The standby message is sent otherwise. The standby message contains the Id of a process whose request precedes that of the requesting process (if any) and a rank which shows the request's position in the *MEC*-queue. Once a permit has been granted, the corresponding request is removed from the *MEC*-queue. See Figure 4.1. Notice that the *MEC* appends its own requests to the queue. This fact can be seen in Figure 4.1 where the rank numbers are increased from 2 to 4, the *MEC*'s rank must have been 3.

When a permission is granted to a process, the process will enter its *CS*. Upon completion, it will notify the *MEC*. Upon receiving such a notification, the *MEC* will grant a permit to the request now at the head of the queue (if any). With this it is clear that, when *CS* demand is very low and the requests occur sparsely then the standby messages are barely used and hence less messages are needed per *CS*-request.

Figure 4.1: Scenario in the absence of faults

4.3.2 Scenario in the Presence of Faults

We allow any number of processes to fail at any time for any length of time. This generality creates many different situations in the system; in the following we consider and describe how the situation is handled by the protocol.

- An inactive non-MEC process crashes. In this case no messages are sent anywhere and the computation is not affected. Other protocol need to exchange messages to deal with this situation [68, 4, 58, 32] and others will continue to send messages to the crashed process in the processing of CS [1, 62].¹
- A non-MEC process crashes after requesting but before receiving a permit. If the permit for that request had been sent, the MEC will proceed to assign a permit to the first request in its queue (if any). If no permit had been sent and the request is not in the queue (this would happen if the request is still in transit) no action is taken. If the request is still in the queue the MEC will await for a notification of a failure from the process *next* whose entry is next in the queue and then remove it. Upon receiving a process failure notification the MEC will send either a permit or a new predecessor to the notifying process. If no such *next* process exists in the queue then the MEC will just remove the queue entry of the failed process.
- A non-MEC process crashes while it is in CS. If a process crashed while it is in CS then the MEC will either await failure notification from the next process in the queue (if any) or it will await the next request to which it will send a permit right away.
- The MEC crashes. If all the other processes knew the MEC then all current fault handling computations are stopped. The next process in the MEC succession chain becomes the new MEC. If all non faulty processes do not

¹In the algorithm of Dolev et al. requests continue to be broadcast to every other process regardless of whether it is active or crashed. In the protocol of Singhal a crashed process can be a member of the *Inform-set* and hence will continue to receive messages.

know the *MEC* identity, a different method (described under Rule 9 below) is performed to determine the new *MEC*.

In all the above situations faults were occurring in an orderly manner and there was always enough time to reorganize before another fault occurred. This enabled us to clearly discuss what happens in the event of a fault. However faults can occur at any time. This means that even while in the process of recovery another burst of process failures can occur and while in the process of assessing the damage yet another burst of failures can occur.

Our protocol will be shown to be always correct. That is, any process wanting to enter *CS*, if it does not fail, it will gain access — this in finite time regardless of how many and how frequently the other processes fail and recover. reorganize is available. At this point we would like to stress that correctness in the presence of ongoing faults can only be assessed when there is a respite and the system is allowed to reorganize in order to do useful computation. If the *MEC*'s would continually fail in an endless fault cascade then finite delays are not guaranteed. In this case no protocol can work because every protocol needs the finite delays, i.e. no protocol can be proved to be correct in an environment where a cascade of faults does not cease.

4.4 Description of the Protocol

In this section we specify the rules of the protocol that are based on the above mentioned assumptions and messages.

4.4.1 Message Types and Sizes

In this protocol every process can be in one of 3 states as shown in Figure 4.2.

A process is *ACTIVE* if it is either in the *CS* or it has requested to enter *CS*.

A process is *FAULTY* if it has crashed. By definition, all its non-faulty neighbors are aware of the crash.

Figure 4.2: Process states

A process is *INACTIVE* if it is functional but is not currently interested in entering *CS*.

A process is *RECOVERING* if it was previously crashed and is now recovering and does not yet know the current *MEC*.

The transition between the various states is shown in Figure 4.2.

4.4.1.1 Types of Messages

The types of messages sent by processors are:

REQUEST This message is sent to the *MEC* by a process seeking *CS* permission. The request message carries the process Id of the requesting process.

PERMIT This message is a *token* which does not contain any other data, it is sent by the *MEC* to grant permission to enter *CS*. Initially no process has made a request for *CS* and the *MEC* has sent no token to any process.

FINISHED This message is sent to the *MEC* by a process which has completed its *CS*.

REPLY This message, sent by the *MEC*, is an acknowledgment to a request to enter *CS* at a time when the token has been sent to another process. This message contains a *MEC* ranking number and the Id of the process whose request precedes the current one in the *MEC* queue (i.e. the predecessor).

NEW-PRED This is a message sent by the *MEC* to a process whose predecessor in the *MEC*-queue has crashed. It is sent only if there are other processes ahead of it in the *MEC*-queue. The message contains the Id of the new predecessor.

SEARCH This message is sent either by a process which has recovered from a crash or by any process after the crash of the *MEC*. In both cases, the message is only sent to the next or the previous process in the *MEC*-succession chain. Initially the message is sent to the respective successors, this is called the *Search-down*. If the last process in the succession chain also does not know the Id of the *MEC* then it sends the *SEARCH* message in the opposite direction, the *Search-up*, a *Search-up* has precedence over a *Search-down*. Each process stores data which shows whether a *Search-down*, a *Search-up* or none of both has transpired. This data is used by the process to determine where to send the *FIND* message if at all.

FIND This message is sent by a process which knows the *MEC* (possibly the *MEC* itself) in response to a *SEARCH* (search-down or search-down) message. This message is sent only along the *MEC* succession chain.

4.4.2 General Rules

Rule 1. A process wanting to enter *CS* will send a *REQUEST* message to *MEC* directly.

Rule 2. Upon receiving a *REQUEST*, if there are no pending requests and the token is free, the *MEC* will send a *PERMIT* message to the requesting process directly. If the token is not free or the *MEC*-queue is not empty, the *MEC* will append it to *MEC*'s queue and will send a *REPLY* message to the process.

Rule 3. After exiting *CS* a process sends a *FINISHED* message to the *MEC* and becomes inactive (as far as the mutual exclusion is concerned) until it makes another request.

Rule 4. When the *MEC* receives a *FINISHED* message, it sends a *PERMIT* message to the process (if any) at the head of the queue and removes it from the *MEC*-queue.

4.4.3 Fault Tolerance Rules

Rule 5. If an inactive process fails, then it is removed from the *MEC* succession chain.

Rule 6. When the *MEC* detects the failure of a process in its queue then it sends a *NEW – PRED* message to its successor in the *MEC* queue (if any), the message contains the Id of the new predecessor of the receiving process in the *MEC*-queue. If the fault is detected just after the free token is available at the *MEC* and the successor of the crashed process is now at the head of the queue then the *MEC* sends it the *PERMIT* directly.

Rule 7. If an active process fails after the token has been sent to it but before a *FINISHED* message has been received from it, then the *MEC* will grant a *PERMIT* to the process now at the head of its queue and then remove it from the queue. Such a rule is activated in the following cases which, note, are indistinguishable for the *MEC*:

1. A requesting process crashed before it could receive the *PERMIT*,
2. a requesting process crashed while it was in *CS* and
3. a requesting process crashed after completing *CS* and sending the *FINISHED* message but the message had not yet arrived)

Rule 8. If a process makes a request and the *MEC* crashes before the process has received a rank then it assumes its request to be lost and has to make a new request.

Rule 9. If a process is recovering, it would not know the *MEC*. In order to know the *MEC* the process initiates the *MEC* seeking procedure (*swing scan*) by sending the *SEARCH* message to its own successor (*Search-down*). If the

successor is also recovering it will either have done the same already or will do it then. Any process that knows the *MEC*-Id and receives a *SEARCH* message will respond by sending a *FIND* message with the *MEC*-Id to the sender of the *SEARCH* message. The process that receives the *MEC*-Id with the *FIND* message will then start sending requests to the *MEC*. Each process stores information showing which process has sent it a *SEARCH* message.

If no process knows the *MEC*-Id, (all recovering processes started recovering just as the previous *MEC* and all previously non-faulty processes were crashing) then the sending of *SEARCH* messages goes on until the last process in the succession chain is reached (the end of the succession chain which is a non-faulty process with the largest Id). At this point the *SEARCH* message starts traveling in the opposite direction (*Search-up*) until it reaches the start-process which has the smallest Id of all the non-faulty processes or until it encounters a process which knows the *MEC*. If no *MEC* exists the first process in the chain then becomes the new *MEC* and returns the *MEC* Id to its successor. Each process that receives the *MEC*-Id will send it to another process that had sent it a *SEARCH* message (if any).

The search procedure can always be interrupted by the handling of a crashed *MEC*.

Rule 10. If the *MEC* crashes, the handling of all the other process failures that might be in progress are interrupted (except for the updating of the *MEC* succession chain). The *MEC* crash-handling has the highest priority and all the other faults will be handled when it wakes up.

The process that is the *MEC* successor according to the *MEC* succession chain described above will start sending *FIND* messages to all non-faulty processes which respond by sending their ranks to the new *MEC* to enable the new *MEC* to collect the current rankings of all the processes that were in the crashed *MEC*'s queue. The process that is currently in *CS* will send the rank -1 . With the collected ranks the new *MEC* can reconstruct the previous

MEC-queue and know whether to grant a *PERMIT* to the process at the head of the queue or to wait for a *FINISHED* message from the process that sent the rank of -1 . If no process sends the rank of -1 then the *MEC* knows that it can safely generate a *PERMIT* to the process at the head of its queue.

Processes that have no rank and are not interested in entering CS will send the *FIN*D message back to the *MEC*. When the new *MEC* receives a *FIN*D message (with its own Id) from another process it ignores it. The message serves only as an acknowledgement to the previously sent *FIN*D message.

Processes that have no rank but are interested in entering CS will send their requests to the new *MEC*. The new *MEC* will append the new requests to the queue so that they come in *FCFS* order after the reconstructed queue.

The collection of ranks can be optimized as follows:

The *MEC* does not have to wait until the whole queue is complete in order to start issuing permits. The issuing of permits can resume as soon as a certain number of ranks have been received. This is so in order not to allow slow links to slow down the whole process. If a high-priority rank is received then it is appended to the *head* of the *MEC*-queue so that it will be the next one in line for CS.

Figure 4.3 shows how the failure of processes is handled and how successive *MEC*'s are installed by the protocol. In part (a) we see just the *MEC*'s successive chain where process k is the current *MEC*, indicated by the shading. In part (b) the previous *MEC* has just crashed and the next process in the chain assumes the role of the *MEC*. All the processes that were not failed at the time of the crash start sending their ranks to the new *MEC* as soon as the predecessor has failed, this is indicated by the arrows 1, 2, 4 and 5. In the meantime, other processes that had previously failed (can even include the previous *MEC*) start recovering and in this state they send messages to their successors to find out the Id of the new *MEC*, indicated by the arrows 2 and 4. As soon as such an inquiry message gets to

Figure 4.3: Failure and recovery of processes

a process that knows the current *MEC*, it responds with the Id of the *MEC*. At the same time, the new *MEC* is also sending messages to all the other processes to collect their previous ranks (if any), indicated by arrows coming out of the node at 6. In Part (c) we see the recovery of only two processes. These inquire about the Id of the current *MEC* from their successors in the succession chain as indicated by the arrows 1 and 2. In Part (d) all the processes have previously crashed and they are now in the process of finding out the Id of the *MEC*. This is a complete *swing scan* in which each process sends a message to its successor. This goes on until the last process (process n) has received such a message. Process n then sends the same message to its predecessor in the chain and the swing goes upwards until it reaches the first process in the chain (in this case, process 1). At this stage, process 1 realizes that currently there is no *MEC* process in the system and declares itself to be the *MEC*. It just broadcasts an announcement message to all the non-faulty processes. After this Part (e) shows the processes beginning to send their requests to the new *MEC* after the announcement.

4.5 Correctness

In this section we will prove the correctness of the algorithm; that is, the protocol exhibits fairness, liveness and is deadlock free.

Theorem 4.1 *The Dynamic Coordinator protocol executes correctly.*

We prove this theorem by means of two lemmas:

Lemma 4.1 *Mutual exclusion is guaranteed.*

Proof:

A process can only enter *CS* after it receives the token. In order to prove that mutual exclusion is guaranteed we only need to show that there is always a single token in the system and if it is lost it is regenerated.

We prove the lemma by contradiction.

Let us assume that there are more than one token available. Initially, by definition, there was only one token in the system and there was only one process designated as the *MEC*.

Also by definition, only the *MEC* can generate another token. Consider the moment t in time when the second token is generated. This token is generated by the *MEC*; the older token in the system might have been generated by any (previous or current) *MEC*. The generation of the second token can only have taken place according to one of the rules 2, 4, 6, 7, and 10 since, according to the protocol, a *PERMIT* can only be granted under them. We shall consider each of those five cases individually.

Generation According to Rule 4: In this case, the *MEC* had generated the second token after receiving a *FINISHED* message; such a message is sent after exiting *CS* and, thus removing the token from the system. But the existence of two tokens at time t implies that there must have been two tokens in the system prior to time t – a contradiction.

Generation According to Rule 6: In this case, the second token was sent to a process whose predecessor had crashed and as a result, its request is now at the head of the *MEC*-queue. This could only have been done after the *MEC* had received a *FINISHED* message from the previous holder of the token. Therefore if there are two tokens now then there must have been more than one token in the system prior to time t - a contradiction.

Generation According to Rule 7: In this case, the second token was generated after the process to which a token had been sent crashed, then that token got lost, by definition. If that is the case then there must have been two tokens prior to time t - a contradiction.

Generation According to Rule 10: In this case, the second token is generated as a result of a new *MEC* succeeding a crashed *MEC*. The new *MEC* can generate a token only after collecting the ranks (if any) from all processes. In doing so, the *MEC* must have received a message from the process holding the other token. But according to the protocol, the *MEC* would not generate a token in such a case - a contradiction.

Generation According to Rule 2: This is the only rule left whereby the second token could have been generated. If the *MEC* had generated the token upon the receipt of a request then this could only have been done because the *MEC*-queue was empty and the token was free. If the generated token is the second token then there must have been a second token (unknown to the *MEC*) in the system prior to time t . This is a contradiction.

Thus, a second token cannot be generated as long as there is a token in the system. This implies that mutual exclusion is indeed guaranteed. $\square$

Lemma 4.2 *Both Deadlock and Livelock are impossible.*

Proof:

We will show that every requesting process, if it does not fail, will enter its CS within finite time.

By definition, the *MEC* will not prevent any process that wants to enter *CS* from doing so in finite time. The perpetual existence of the *MEC* is guaranteed by the *MEC*-succession method. We defined an assumption whereby there is always enough time to reorganize after failures so as to do useful computation. This is the only case in which that assumption is needed.

This means that even a cascade of failures will halt in finite time, long enough for the system to reorganize. As long as the active set of processes (known to the *MEC*) is non-empty then each of the processes is guaranteed to enter *CS* in finite time.

Should the token get lost, the *MEC* guarantees the regeneration of another one in finite time. An active process that crashes will have to make a new request for *CS*. These factors guarantee that no deadlock will occur. The *MEC* will grant permits to the requests that come to it on a *FIFS* basis and this is enforced by the queue which does not allow for preempting of requests by other requests. As long as a functional *MEC* exists in the system then fairness will be guaranteed. If the *MEC* crashes then the new *MEC* will collect data from all processes to enable it to reconstruct the previous *MEC* queue and pick up where the previous *MEC* left. This guarantees the continuation of the *MEC* queue ordering hence guarantees fairness. □

4.6 Number of Messages per Event

In this section we consider the number of messages needed by the protocol. We count the actual number of messages transmitted, not received. This is an upper bound on the total number of messages, since to be received a message must have been transmitted but not all transmitted messages are received.

We take the initial situation where no *CS*-requests have been made as the starting point of our measurement and an arbitrary point t after some requests have been made and some even granted. We are considering the worst case, thus we assume that the demand for *CS* is high.

Figure 4.4: Fault tree

There are a number of faults that can transpire in our model. Each type of fault has a different priority. Figure 4.4 shows the different faults that are dealt with in the protocol. The root and left children in the fault tree depict the fault as being more severe than the ones on the right children. Of course when the faults occur they can be any combination of the fault types.

In the cases of faults we count or *charge* the messages as follows:

- If the *MEC'* crashes we charge all lost requests of processes that had not yet obtained a rank to the recovery of the *MEC'*. In this case the *MEC'* might even have sent off the ranks which then get lost when the *MEC'* crashes. This means that when the *MEC'* crashes we loose a maximum of $2(m-2)$ messages in the worst case, where m is the number of active processes ($(n-1) \geq m$).
- If a new *MEC'* is installed this costs $2(m-2)$ messages, where m is the number of non faulty processes and $(n-1) \geq m$. This is because the new *MEC'* has to poll all the other non-faulty processes for their ranks (if any) and hence there are two messages exchanged with each other non-faulty process. If all the other processes besides the crashed *MEC'* knew the crashed *MEC'* then

they start sending their ranks to the new one (by definition they know it), this is also charged to the installation of the new *MEC'* and we remain with $2(n - 2)$ messages. The recovery of the *MEC'* is just a special case of the recovery of a faulty process.

- If a process is recovering we charge the cost of it finding out the new *MEC'* to its own recovery which is 3 messages for each such process.

Note that we have described how we charge the messages in cases of faults. Some of the messages that we charge for separately could actually be one and the same message appearing in different scenarios. For example we charge for the recovery of each process and we charge for the installation of the new *MEC'* if an old one crashes. Obviously we are counting some of the messages twice. Another example is that we charge for lost ranks for all the processes in the case of a crashed *MEC'*. This case can hardly arise because the *MEC'* deals with incoming requests, one at a time and it is not the case that it would have issued ranks to all other processes all at once. Either it fails before it is able to accomplish this or some of the ranks arrive at the processes before the *MEC'* crash is detected.

This shows that our message counting method is quite generous and should be seen as an upper bound. In reality the number of messages needed for each event will be lower.

Theorem 4.2 *The number of messages needed by the protocol for each event are as follows:*

- for entering CS, 3 messages,
- for handling a crashed token holding process, 1 message,
- for handling a crashed non-*MEC'*, non-token-holding process, 0 messages,
- for the recovery of a crashed process, 3 messages,
- for handling the crashing of the *MEC'*, $3n$ messages

We will prove this (see Table 4.1) with the following lemmas.

Type	Event	Nbr of Messages
1	Entry into CS	3
2	Crash of token holder process	0
3	Crash of process in MEC queue	0
4	Crash of Non-active process	0
5	Recovery of a process	3
6	Crash of MEC	3n

Table 4.1: Number of Messages Per Event

Lemma 4.3 *The number of messages needed by the protocol for each CS-entry is 3.*

Proof:

This protocol needs 3 messages per CS, one request, one reply and one permit. $\square$

Note: The number of messages needed by the protocol at low CS-demand is 2 for each CS entry. This is the case when the requests are generated one at a time and no other request is generated until after the current request has been services and the process has left the CS. This means that to every request message there is a permit message and the reply message is not used. *No distributed mutual exclusion protocol can do better than this because 2 is the absolute minimum communication in message passing systems.*

Lemma 4.4 *The number of messages needed to handle the crashing of a process to which the token has been sent but before it sends back a FINISHED message is 0.*

Proof:

If a process crashes after the token has been sent to it but before a *FINISHED* message is received from it then the permit is granted to the next protocol in the queue. We charge this granting of the token to the action of requesting a token and not to the handling of the crashed process. In other words, if there were no other processes in the MEC-queue then the MEC would just wait for the next request to arrive. $\square$

Lemma 4.5 *The handling of the crash of a non-active process takes 0 messages.*

Proof:

If a non-active process crashes then only the *MEC*' succession chain needs to be updated. This is done by the underlying mechanism which lies outside of this protocol, hence we charge no messages for this. $\square$

Lemma 4.6 *The number of messages needed for the recovery of a previously crashed process in the worst case is 3.*

Proof:

A recovering process will first send a *SEARCH* message to its successor in the succession chain and then it will receive the same message from that process. The recovering process then send the *SEARCH* message to its predecessor in the same chain. After this message, the next message will announce the Id of the current *MEC* and the process is fully recovered upon receiving this message. $\square$

Lemma 4.7 *The number of messages needed to handle the crash of the *MEC* in the worst case is $3n$*

Proof:

The worst case scenario is one in which the *MEC* crashes and while it is recovering, all the other processes also crash and begin to recover immediately.

In this case we need 3 messages for the recovery of each of the $(n - 1)$ processes. We also need 3 messages for the recovery of the previous *MEC*. Altogether we need $3n$ messages, the maximum number of messages of the protocol but the occurrence of this worst case has a probability which is admittedly very small in a normal system.

For example, if just the *MEC* crashes while all the other processes survive then all the surviving processes know the successor, by definition. They all need to send their rankings to the new *MEC*. This takes $(n - 2)$ messages. The crashed *MEC* may be recovering and it takes two additional messages for it to change from recovering to inactive. This makes a total of (n) messages in the case of just the *MEC* crashing. $\square$

4.7 Storage Considerations

The total amount of storage involved in this protocol is a sum of the local storage, the message contents and the *MEC*' storage. The message contents were discussed above. Each process will store its *MEC*'-successor and its *MEC*'-predecessor Id's to enable the determination of the new *MEC*' if the current *MEC*' crashes, it will store a bit to show which process sent it a *SEARCH* message and it will also store the *MEC*'-Id, if known. All the other storage is dynamic and is as follows:

The *MEC*' will store a queue of dynamic length into which requests are appended. The queue shows the Id of the requesting process, its rank and the status of the request. Every other process (including the *MEC*') will store the following data which is generated upon becoming active: a predecessor Id, a successor Id and a *MEC*'-rank. Some protocols that have been published in this area do not address the issue of storage completely.

4.8 Assessments, Extension and Possible Generalizations

We have presented a protocol for mutual exclusion which allows for single entry into *C'S* even in the presence of faults. In this section we conclude by assessing the protocol (general comparisons) then we show how the protocol can be extended to allow for multiple entry into *C'S*, finally we discuss the possible generalizations of the protocol.

4.8.1 Assessment

We have not dealt with all possible faults that can occur, especially those faults that occur at lower levels of abstraction such as faulty hardware which can cause link failures. This would have gone beyond the intended scope of this thesis. We have actually gone further in addressing the important issue of fault tolerance than most protocols, some of which do not even discuss this aspect, for example, [65, 31,

5, 67, 68, 59]. Our protocol is relatively easier to understand. The simplicity of this protocol is demonstrated by the simplicity of the proofs compared to other protocols [48]. Our protocol exhibits lower local storage and smaller message sizes compared to many others, for example, [39, 58, 52].

4.8.2 Extension

One extension which can be made is to allow for multiple *CS* entry instead of a single entry into *CS*. There are two possible ways of doing this: One method is by allowing the creation of k *MEC*'s in the system. Each *MEC* has its own running which is that described above. A method of distinguishing the *MEC*'s is needed and can be easily implemented by tagging the messages with a different tag for each token. This method can become very expensive. Another method is by changing the single *MEC*. All that needs to be done in this case is to allow the *MEC* to issue k tokens. The rest of the protocol remains the same, the number of messages remain the same and the fault tolerance remains the same as well. This is a demonstration of the protocol's simplicity compared to the complex process of effecting a similar generalization in other protocols[59].

4.8.3 Possible Generalization

The protocol can be generalized a number of ways: Link faults can be allowed to occur. One problem with this kind of fault is that one might have to broadcast along all links to assess link status and partition has to be reckoned with. Bursts of faults can be allowed to occur in any order and at any time. In fact, Byzantine faults can be allowed to occur. The problem with loosening all restrictions on faults is that no protocol can cope with the worst case. The topology can be allowed to be an arbitrary graph. In this case the problems of partition and merge emerge and they have to be efficiently handled first and foremost, especially in the light of link faults. When partitions occur the possibility that a functional token is still available

in one of the partitions is given. The number of partitions can be as high as n , a decision has to be made as to which partition inherits useful computation.

Some methods used to cope with partitions use *time-outs*. The use of *time-outs* requires prior knowledge of the system. Taking these into considerations reduces the generalization of any protocol. We did not allow link faults and arbitrary topologies for obvious reasons. But we believe that one good strategy to solve the problem is by superimposing a spanning tree on the graph and then manipulating the graph. A group membership protocol may have to be used also in this case.

4.9 Summary

In this part we started out by addressing some mutual exclusion protocols. We went on to present a mutual exclusion protocol which uses a dynamic coordinator to manage mutual exclusion. This protocol requires $O(1)$ messages per entry in $C'S$. We showed that fault handling is also not costly in this protocol, ranging from 0 to $O(n)$ messages. Not only is this protocol efficient and fair, it is also very resilient and uses small amounts of local storage. We did not handle random and arbitrary failures in this chapter. We believe that the study of the protocol in the presence of arbitrary faults brings benefits not only to this particular problem but for the whole area of distributed computing. We intend to look at this aspect in the course of our future research.

Part III

Termination Detection

After considering the area of mutual exclusion, we now focus on the area of distributed termination detection in this part. Some of the issues on termination detection have already been introduced in Part I. This is very important in multi-phase algorithms where the order of execution of the various phases has to be in a fixed sequence; here it is important to detect when one phase is completed before the next phase can be initiated. The *2 Phase Commit* protocol in Distributed Databases is one such computation where one phase may only begin when another has terminated, if deadlocks and errors must be avoided.

A termination detection algorithm is viewed as operating at a lower level than the application that is being monitored for termination in the sense of *OSI*-layers. This means that the termination detection protocol has full access (read-only) to all the messages and data sent out and received by the application. The application is normally referred to as the *underlying computation* although in this sense it is actually above the termination detection algorithm.

We discuss the basic techniques of some termination detection algorithms that have been published. The survey that we present here is not meant to be a comprehensive study of termination detection algorithms but rather a brief discussion of some of the general techniques that have been used in this area. A more elaborate study would go beyond the intended scope of this thesis. For more information on termination detection protocols we refer the reader to works by Andrews[1], Tel[66] and Raynal[55].

Chapter 5

Survey of Related Work

One important characteristic in the design of termination detection algorithms is the guarantee that they do not interfere with the monitored computation. This is generally referred to as the *non-interference* of an algorithm. This aspect was not considered in early termination detection algorithms. Next to *non-interference* is the ability to detect termination in distributed systems were the problem of ensuring global termination is relatively more complex because of the existence of independent and concurrent processes running on different processors. The conditions under which termination can be assured and detected need not only to be viewed at the local level of each process but at the global level where an overview of the whole system is possible. The technique used to detect termination should not be an integral part of the observed process, rather it should be as independent and as small as possible, running beneath the observed process. All conditions or states that have a global bearing on the whole distributed system are known as global states. The observation of global states, also known as computing snapshots, is non-trivial. This is because of two main factors,

- the lack of a global clock, valid for all processes and
- the non-determinism inherent to arbitrary transfer delays of messages in an asynchronous environment. The quest for a solution of the above mentioned problem has some benefits or desired spin-offs, these are:

the detection of stable properties (e.g., termination) in a distributed environment.

the recovery of a system after a failure and

the design of a fundamental mechanism for distributed application control.

5.1 Classification of Termination Detection Techniques

There are four main techniques used to solve distributed termination detection problems: *Diffusing Computations*, *Vector Clocks or Time Stamping*, *Circulating Token* and *Credit Sharing*.

5.1.1 Diffusing Computation

Here the control messages are propagated through repeated triggering such that a node that receives a message broadcasts the same message to selected neighbors. This method requires topology awareness and that channel types be specified, e.g., as being *FIFO*. Each process keeps track of all incoming messages and all outgoing messages and thus is able to tell whether all messages it has sent have been acknowledged and whether it still has to respond to any incoming messages. When there are no outstanding incoming and outgoing messages at every node termination is said to occur. This method can be used in conjunction with other methods such as *graph embedding* and *vector clocks*. The message complexity is optimized by ordering the propagation of messages.

5.1.2 Time Stamping

In this approach vector clocks described in [37] are used to mark events and by synchronizing the clocks time cuts (consistent sub-cuts) can be obtained and evaluated. This method involves an optimal sampling (at optimal intervals) of events in the

system and the interpretation of the sampled data to determine the global state. This method usually requires considerable local storage at each node.

5.1.3 Circulating Token

In this case either markers are stored at each node or a token traverses the network along a circular path that visits each node. When markers are used then the token can set the markers according to certain conditions being met at each node and using the markings termination can be determined. This method requires the processing of the *Euler* cycle in the network or the creation of a depth first search (*DFS*) tree. The detection of the loss of a token has to be an integral part of the protocol and token regeneration must be guaranteed in case of token loss.

5.1.4 Credit Sharing

This method involves the distribution of a divisible resource (either large integers or bits) by an initiating node to the others. The other nodes then return fractions of the resource as they become terminated. The initiating node (which knows the original quantity) can detect whether all the shared out resource has been recovered and can detect termination. The method of dividing the resource has to be simple and care has to be taken that none of the fractions can get lost. A mechanism of regenerating more resources when necessary has to be ensured. This method usually requires that the nodes keep track of messages (either all or a subset) that are transmitted so as to tell if any messages are in transit.

5.1.5 Basic Assumptions for Termination Detection Protocols

In general all termination detection protocols make the following assumption:

- all processes have two main states, either they are *passive* or they are *active*
- an active process can become passive at spontaneously

a passive process can only become active upon the reception of a message from another process

only an active process can send messages to other processes

Some protocols make the following assumption depending on whether they are meant for a *synchronous* or an *asynchronous* environment:

the sending and receipt of a message occur within the same *atomic action*. This assumption is only needed for synchronous environments. Most protocols, including the one we present, deal with asynchronous environments.

The following is the fundamental theorem (of Termination Detection):

Theorem 5.1 *Once a system is terminated, it is always terminated.*

For termination to take place two conditions have to hold:

1. *Every process is in the passive state and*
2. *There are no messages in transit*

A brief study of some of the Termination Detection protocols follows.

5.2 Brief Survey of Termination Detection Algorithms

As we have already mentioned, this is not intended to be a comprehensive survey of termination detection protocols but rather an illustration of the different techniques in form of examples.

5.2.1 The Dijkstra-Scholten Algorithm

This algorithm[22] represents algorithms that use the diffusing computations techniques. This is a classic algorithm referred to by many researchers of this area of distributed computing. The basic idea is as follows: There is a directional network of strongly connected processes (the topology is arbitrary) in which there is a process designated as the *environment* (*ENV*) which has the task of detecting

termination. The *ENV* does not have incoming edges. All the other processes are known as *internal* nodes. Each process regards the process from where incoming edges start as the *predecessor* and the processes where its outgoing edges end as its *successors*. Processes can receive messages and signals from their predecessors and successors respectively. Processes can send messages and signals to their successors and predecessors respectively. All the communication takes place within the same channels. The number of transmitted signals in the system always balances the number of transmitted messages at the end of the computation.

Two deficits are defined for each process. First, a deficit C along incoming edges which is the difference between received messages and sent out signals. The sent out signals never exceed the received messages because signals are sent in response to incoming messages, therefore $C \geq 0$. For the *ENV* this deficit is always 0 because it has no incoming edges. The second deficit D is the difference between sent out messages and received signals. The number of received signals never exceeds the number of sent out messages because the processes can only respond with signals after receiving messages, therefore $D \geq 0$. All internal nodes are reachable from the *ENV* via directed paths, all edges of which have non-negative deficits. A process keeps track of all outstanding signals by using the deficit mechanism. A process is not allowed to send a signal to its predecessor as long as it is active or as long as its D -deficit is greater than 0.

A bounded number of steps after the diffusing computation has terminated the *ENV* will have returned to a neutral state where there are no outstanding messages or signals. Note that this protocol does not have any topological constraints. No fault tolerance aspects are considered and the message complexity is $O(b.n)$. Besides having a large message complexity the protocol also has other drawback:

- the deficits C and D are unbounded for each process and
- the algorithm must be initiated at the starting of each computation.

The ideas in the algorithm have been adopted and used for various applications such as [13, 16, 11, 17].

5.2.2 The Misra Algorithm

This algorithm[15] represents algorithms that use the *circling token* method to detect distributed termination. This algorithm is for a strongly connected network of processes and it used some ideas inherited from the Francez-Rodeh algorithm [27]. A single *marker* or *token* repeatedly traverses all edges until it detects termination. The protocol requires *FIFO*-channels but does not make use of the process Id's. The latter fact is sometimes referred to as the proof of symmetry.

The principle of the algorithm where the token visits all processes can be illustrated by assuming a directional *ring* topology. In this case the token begins at a terminated process and goes marking all subsequent process if they are terminated. Encountering an active process means that the token waits at the process until the process has terminated. When the token returns to the starting point and finds the process still terminated then termination of the whole system can be concluded. Two factors are important in this.

1. the FIFO channels ensure that when the token has gone through a channel then it has flushed out any messages that might have been in transit along that channel and
2. every edge has to be traversed at least once in the detection process.

The idea is generalized for an arbitrary network by introducing an edge-counter which is carried by the token. The number of edges in the system are known and every time the token traverses an edge to find a terminated process it increases this counter by one. A path (*Euler cycle*) is determined in the system which allows the token to traverse every edge and thus also visit every process. If the token encounters an active process then it reinitializes the counter and starts the traversal as soon as the current process terminates.

Another aspect of graph coloring is also introduced: Before a token departs from a terminated process it colors the process *white* and increases its edge counter by one. The token does not stay permanently at an idle process. When a message arrives at a process, the process colors itself *black* before beginning its computation. Whenever

the token arrives at a black process it begins the detection all over again. When the edge counter reaches the total number of edges in the system then termination is concluded. This method has a further benefit in that the moment when termination occurred in the system can be determined to lie between the starting of the last detection traversal and the instant of detection. The drawback is that for arbitrary strongly connected networks a preprocessing of the *Euler* cycle is required.

A *DFS* (depth first search) method can be used for arbitrary networks where the traversal is initiated by the current root of the *DFS* tree. If an active or *black* process is discovered then the traversal starts all over again with the root being the previously active process when it has terminated. In this case no special *ENV* process is needed. The paper shows how the same method is used to solve mutual exclusion and the *Token Regeneration* problem. The protocol has a message complexity of $O(b \cdot e)$, where e = total number of edges.

5.2.3 The Mattern Algorithm

This algorithm [42] represents algorithms that use the *credit distribution* technique to detect distributed termination. This algorithm requires a special process (the *ENV*) which initiates the underlying computation and then becomes detached from that initiated computation. This *ENV* is then used to detect termination. The principles of the protocol are as follows:

Upon initiation, the *ENV* distributes a credit resource (whose sum total is known to itself) among all the processes. The *ENV* then maintains a set of *debts* to keep track of missing shares. An active process holds a credit share > 0 and an activation message also has a credit share > 0 . Each process can send part or all of its credit to the *ENV* via control messages or signals. Before becoming passive or before terminating, a process must have sent off all its credit either via activation message to other processes or straight to the *ENV* via signals. When the *ENV* has recovered the original amount then termination is concluded because there can be no active process in the system (all have surrendered their credit shares) and there can be no message in transit (an activation message would have to have a credit share).

Normally the number of activation messages equals the number of control messages (signals), this means that the message complexity of this algorithm would be equal to that of Dijkstra and Scholten [22]. The number of messages needed for termination detection in this case can be reduced by allowing a process to accumulate shares that are being sent by other processes until it has terminated itself. This method, though reducing the message complexity involves the complicated issue of adding the fractions at each process. This algorithm has the potential problem of having to define an efficient (loss-less) credit partition method. Such a method which uses binary operations is provided in the paper. The *ENV* can be eliminated by allowing local autonomy whereby any process can initiate termination and be the detecting process as well. The protocol has a message complexity of $3b(n - 1)$.

5.2.4 The Huang Algorithm

This algorithm[33] represents algorithms that use the *distributed snapshots* or *time stamping* technique to detect distributed termination. The algorithm does not assume a special process or an *ENV*. The messages that are sent in the system are divided into two groups, *basic* messages (concerning the underlying computation) and *control* messages (also known as *signals*). The algorithm uses clocks based on the Lamport clocks [37]. The method is composed of a sampling of processes and channels for any activity, the lack of which indicates termination. Each message is time stamped. Each process begins with a clock value of 0 and it is incremented every time the process terminates or becomes idle. The processes update their clock values according to the Lamport clocks. The aim is to obtain a consistent cut which shows the global state of the system.

A process that becomes idle broadcasts a time-stamped sampling request to all other processes. Through the time stamp the receivers are able to determine whether the sampling request is the youngest request that has arrived. Older requests are silently ignored. Each recipient of a sampling request updates its clock value if necessary and then send its status to the requesting process (if the request is valid). The tie of having equal time stamps is broken by the process Id's. Eventually,

the last process to terminate detects termination in finite time. Fault tolerance is also not considered in this algorithm. The message complexity in this algorithm is $(2b + 2)(n - 1)$.

5.3 A Brief Discussion on other Related Issues

As we mentioned above, all the current termination detection algorithms can be grouped into four main categories, the *shared credit* based, the *graph marking (circulating-token)* based, the *diffusing computations* and the *consistent cut (vector clock)* based. In 1989 Mattern and Huang [12, 33] separately (apparently without prior knowledge of each other's work) came up with two very similar algorithms for termination detection in an arbitrary graph with a single initiator. Both algorithms let the termination monitoring node (*EM*) share out some values among all nodes and messages involved in the computation and detection work. The *EM* then starts recollecting these shared out values. The collection is done by letting nodes that become passive and messages that became irrelevant (e.g. an activation message arriving at an already *active* node) send their shares back to the *EM*. The *EM* concludes termination if the original sum is reached. The methods used by both authors have many things in common and there is one aspect which seems less desirable in both, the unbounded counters. Each has either an integer or a number of bits which are not bounded in the sense that the integer or the bits are continually subdivided, i.e. the number of fractions is allowed to grow and is limited only by the number of basic communications involved and the division of shares becomes somewhat complex. We show that termination can be detected without having to use such unbounded variables.

Ideally, the best method for termination detection would be one in which all participating nodes can detect global changes at the same time without a prior investigation by a central process but such an approach would be difficult to realize without increasing the message overhead and local storage (this is a subject of future research). There are different methods by which the nodes' states are audited. One

extreme is to do so at regular intervals. This is not a good strategy because the interval has to be optimized for each system, depending on its characteristics. Another extreme is to have each node report its every termination to the coordinating node. This has the disadvantage of having to distinguish between final terminations and temporary ones because terminated nodes could still restart after they receive messages from other *live* nodes, even after previously reporting termination.

5.4 Complexity Measurements

It has been shown[66] that the message complexity of all termination detection algorithm is composed of two parts, the basic messages (b) and the detection wave part. This means that no termination detection algorithm for the asynchronous and distributed environment can have a message complexity less than b , in fact the optimal termination detection to date was the one of Shavit and Francez[60] which has b as the basic message component and $(n \cdot c)$ as the detection wave component, where c is the number of edges or communication links in the system. Some termination detection algorithms require that every basic message be acknowledged while some do not. Requiring an acknowledgement when the system itself is a restriction on the system, but an acceptable one. Our algorithm and that of Shavit and Francez[60] require the acknowledgement of basic messages. Some algorithm require that the communication channels between *nodes* be *FIFO* and some do not. Usually this restriction is imposed because of the negative effects of a message being overtaken by another one along the communication channel, if the algorithm depends on the order of message arrival. Imposing the *FIFO* condition on the system is therefore more restrictive and both our algorithm and that of Shavit and Francez do not require *FIFO* channels.

Table 5.1 shows the message complexities of some termination detection algorithms. There are some other termination detection algorithms[16, 50] that are not

<i>Alg. Ref.</i>	<i>Message Complexity</i>	<i>Channel Type</i>	<i>Global Storage</i>	<i>Comp. Initiation</i>	<i>Comments</i>
[31], 1988	$(\frac{1}{2}b + 1)(n - 1)$ (ring)	FIFO	$O(n-1)$	Single	testing cycle every $\frac{1}{2}$ nodes
[26], 1980	$4b(n - 1)$	FIFO	$O(n)$	Single	Freezing + 1 Waves
[22], 1980	$b.n$	FIFO	$O(1)$	Single	Diffusing Comp.
[27], 1982	$2b.n + n$	FIFO	$O(n)$	Multiple	3 Waves, no Freezing
[45], 1983	$b.e + 2(n - 1)$	FIFO	$O(1)$	Single	Circulating token
[12], 1989	$3b(n - 1)$	FIFO	$O(b)$	Single	Credit sharing
[33], 1989	$(2b + 2).(n - 1)$	-	$O(n)$	Multiple	Time Stamping

Table 5.1: Worst-Case Complexities of Some Termination Detection Algorithms
ring=directional ring, e=number of edges, b=number of basic communications

mentioned here because they assume *synchronous* communication. As we mentioned in Part I, synchronous communication is not yet feasible.

5.5 Summary

From the above mentioned termination detection algorithms and from Table 5.1 one fact does emerge, the new algorithms focus on robustness, fault tolerance and reduction of the sizes and numbers of messages. Most termination detection protocols (unlike those for mutual exclusion) do not address the aspect of presenting the message complexity of the protocol.

Another observation is that most termination detection algorithms require a special process to detect the termination. A few algorithms do not require this but having a central detecting process is not a disadvantage as long as any process in the system can play this role when required to do so or in case of the failure of the previous detecting one. We now present an optimal termination detection protocol.

Chapter 6

An Efficient Termination Detection Algorithm

In this chapter we present a new algorithm for Distributed Termination Detection.

Our algorithm uses a bounded number of messages, bounded local storage and does not require *FIFO*-channels. The number of messages exchanged in the worst case is $b + 3(n - 1)$ where b is the number of *basic* messages exchanged in the monitored computation and n is the number of nodes in the network. This improves on the algorithm of Shavit and Francez[60] which has a similar structure but requires $b + n\epsilon$ control messages, where ϵ is the number of *edges* or *links* in the network. Because of the $\Omega(b + n)$ lower bound on the Termination Detection problem, our algorithm is optimal with respect to the number of messages used to detect termination.

The algorithm we present reduces the message overhead by eliminating the need for multiple audits such as in [16, 50, 31]. The issue of fault tolerance is also mentioned at the end.

6.1 Principles of the Algorithm

In our approach we have a spanning tree, the root process of which is the *Detector* and it eventually detects termination when it does occur. We allow the *Detector* to start the detecting process only *after* the spontaneous awakenings have transpired

or *after* the initiations of the basic computations. **Note** that we do not restrict the monitored computation to have only a single initiator such as in [22, 26, 45].

After the awakening, the *Detector* waits for its children in the spanning tree to report termination. Once all the immediate children of the *Detector* report termination, the rest of the system is truly terminated but before that, processes can be activated then terminate and then be reactivated again.

We use a (*spanning tree*) superimposed on to the network that is being monitored for termination. Each node in the monitored computation is associated with a process which is part of the termination detection mechanism. The process Id and the *node* Id are always the same for each site.

We show the association of nodes as follows, $process_i$ is the i^{th} process in the logical spanning tree and $node_i$ is its corresponding node in the monitored computation. When we mention *nodes* we are referring to the monitored computation and when we mention *process* we are referring to the termination detection process. When a node sends or receives a message then the corresponding process has *read-only* access to the message, but a process can send or receive a message without the corresponding node becoming aware of it.

Initially we assume that there is a rooted spanning tree which is used only to transmit particular types of control messages. If one does not exist in the network, it can be constructed using $O(c + n \cdot \log(n))$ messages [28]. The root of this spanning tree is called the *Detector*.

Whenever a computation is started in the monitored computation, the process which detect this broadcasts a special (WAKE UP) messages on the spanning tree, this causes the spanning tree to go into the initial stage within finite time and then the termination detection begins. The beginning of a computation in the monitored computation can involve numerous spontaneous awakenings and multiple initiator. This means that we need at most $2(n - 1)$ messages of initiation in the case of multiple activations and exactly $(n - 1)$ in the case of single activation.

Any process that becomes initiated in the spanning tree regards its children in the tree as being active (even if it has not sent them the initiation messages yet) until they send *DISCONNECT* messages.

Each *node* has two states, *active* and *passive* and each associated termination detection process also has two states, *live* and *inactive*. When *node_i* is *active* then *process_i* is also *live* but not vice versa. When *process_i* is inactive then *node_i* is also *passive* but not vice versa.

All the nodes in the spanning tree are initiated with a commitment such that they have to send a *DISCONNECT* message to their respective parents, this means that the *Detector* expects *DISCONNECT* messages from all its immediate children.

A process only sends the *DISCONNECT* message to its parent after it has received *DISCONNECT* messages from its children (if any) and after it has become *inactive*, i.e. its corresponding *node* has also become *passive*.

If *node_i* activates *node_j* ($j \neq i$) then *process_i* expects a *DISCONNECT* or an acknowledgement message from *process_j*. But *process_j* only sends the *DISCONNECT* or an acknowledgement message either if it is already *live* when it receives the message or after it has received all its outstanding *DISCONNECT* messages and *node_j* has become *passive* respectively. The main principle of the algorithm is as follows: If *node_i* sends a message which causes *node_j* to become *active* and this leads to *process_j* becoming *live* then *node_j* sends an acknowledgement to *node_i* which causes *process_i* to expect a *DISCONNECT* message from *process_j*. This means that even though *node_i* becomes *passive*, *process_i* can only become *inactive* after it has received the *DISCONNECT* message from *process_j* and *process_i* sends the *DISCONNECT* message when it becomes inactive.

As we now prove, when the *Detector* has received *DISCONNECT* messages from all its children termination can be concluded.

The principles of our algorithm are somewhat similar to the processes of the UNIX¹ operating[2] system in the sense that in unix, a parent process can *sleep*

¹UNIX is a registered trade mark of AT&T

m wait to synchronize with the exiting of its children processes and in that case the parent process is removed from the process table only after all its children have exited.

6.2 Assumptions and the Basic Model of the Algorithm

We make the following assumption, we assume that no new computation can be started before the termination detection process is completed. This assumption can be easily lifted, for example, by associating a unique index to each computation.

Our basic model is as follows:

- **Activation:** Initially any number of *nodes* can become *active* spontaneously as mentioned above. After that a *passive node* can only become *active* upon the receipt of a message.
- **State change:** Each *active node* may become *passive* spontaneously. Only *active nodes* may send messages.
- **Links:** Every link between any pair of processes in the spanning tree is a bidirectional communication channel.

Note that we do not require *FIFO* channels and we also do not require knowledge of the topology by the nodes.

6.3 The Algorithm

In this section we describe the algorithm in detail. First we consider the types of messages and the local storage required then we define the rules of the algorithm.

2 OF 2

PM-1 3 1/2" x 4" PHOTOGRAPHIC MICROCOPY TARGET
NBS 1010a ANSI/ISO #2 EQUIVALENT

6.3.1 Message Types

There are only two types of messages between any pair of processes, the initialization message (*INIT*) and the disconnection or acknowledgement message. The acknowledgement to the message that makes a process to change state from *inactive* to *live* is called the *DISCONNECT* message. There is no actual difference between a normal acknowledgement and a *DISCONNECT* message but we differentiate them here to make the principles of the algorithm appear more clearly.

Every *node* expects an acknowledgement for every message it sends to another *node* and hence each termination detection process keeps track of all unacknowledged messages that have been sent to other nodes by using a counter, *cc*.

A *passive node* that receives a message becomes *active* but the corresponding process does not acknowledge the message if it also caused it to change state from *inactive* to *live* until it becomes *inactive*. This means that each process can receive at most one such message that it defers to acknowledge at a time, this implies also that each process can have a maximum of $(n - 1)$ commitments, or messages that are yet to be acknowledged.

A *node* can become *passive* while its associated process is still *live*, this happens if the *node* has activated other *nodes* which are still *active* or committed. If a *live* process receives a message then it responds with an acknowledgement right away. The process to which a process has to send a *DISCONNECT* message is called its *parent* and the processes to which a process is committed (expects *DISCONNECT* messages from them) are called its children.

A process only sends a *DISCONNECT* message to its *parent* if it has no unacknowledged messages, i.e. its $cc = 0$ and also if its associated *node* is *passive*. The sending of such a message means that the process and the node have terminated and they remain that way until an activating message (if any) arrives.

6.3.2 Local Storage

We now focus on the local storage involved for each node. Each process in the algorithm stores the following operating data locally:

cc_i This is a counter with which process i keeps track of unacknowledged messages that it has sent. If $cc_i = 0$ then process i expects no *DISCONNECT* or acknowledgement messages.

CO_i This is a single bit. If cc_i is active then $CO_i = 0$, otherwise $CO_i = 1$.

PA_i This is a process Id of the $process_{p_A}$ that caused process i to become *live*. If the $process_i$ is *inactive* then this variable is empty. Also when the $process_i$ sends a *DISCONNECT* message then PA is reinitialized to empty.

Each process stores the following static data locally:

PAR_i This is a process Id of $process_i$'s parent in the initial spanning tree.

SON_i This is a counter which indicates the number of $process_i$'s children in the initial spanning tree.

STT_i This is a single bit set to 0.

Initially, for each process i the operating data is identical to the static data, that is, initially,

$$PA_i = PAR_i; cc_i = SON_i; CO_i = STT_i$$

The static data is only used for the reinitialization of the termination detection algorithm.

6.3.3 Rules of the Algorithm

The *liveness* and *safety* of the algorithm are guaranteed by the following rules:

- Rule 1.** A process responds to any incoming message with an acknowledgement, if before receiving the message it was already *live*. If it was *inactive* it now becomes *live*, makes the sender of the activating message its *parent* by appending its Id to its *PA*.
- Rule 2.** When $node_i$ sends a message to another one then $process_i$ increments cc_i by one and when an acknowledgement arrives it decrements it by one.
- Rule 3.** When a $node_i$ becomes *passive* $process_i$ sets $CO_i = 1$. If $cc_i > 0$ then it sends a *DISCONNECT* message to $process_{p_A}$. It then sets *PA* to empty and enters the *inactive* state. This shows that the process has terminated and is no longer committed to any other process. When $node_i$ becomes *active* $process_i$ sets $CO_i = 0$ and becomes *live* if it was *inactive*.
- Rule 4.** A process i cannot change its state from *live* to *inactive* until all its local variables are set as follows:
 $cc_i=0$, $CO_i = 1$ and $PA_i=\text{empty}$.

6.4 Correctness

In this section we present the correctness of the algorithm in terms of *safety* and *liveness*. The spanning tree composed of *live* processes = T .

Theorem 6.1 (Safety) *When the Detector has entered the inactive state then termination is concluded.*

Proof:

In this case we need to prove the following:

1. The spanning tree structure of *live* processes is always maintained and there cannot be a *live* process disjoint from the spanning tree.
2. The *Detector* can never become a child of any other process.

3. When the *Detector* has entered the *inactive* state then there are no messages in transit and all processes are also *inactive*.

Firstly we define an invariant of the algorithm which is maintained by all processes in all states shown in the code (see the Appendix).

$$Q \Leftrightarrow \text{SpanningTree} = T \Rightarrow \text{root}(T) = \text{Detector} \quad (6.1)$$

$$p - \text{state}_i = \text{live} \Rightarrow \text{process}_i \in T \quad (6.2)$$

We prove the rest of the theorem using the following lemma: □

Lemma 6.1 *Q is an invariant of the presented algorithm.*

Proof:

We prove this by induction.

Base Case: Initially (by construction) all the processes are *live*, they are in *T* and *T* is rooted in the *Detector*, this preserves *Q*. Also by definition,

$$\text{node}_i = \text{active} \Leftrightarrow \text{process}_i = \text{live}.$$

In *T* we only have two types of actions: *delete-process* and *insert-process*.

Deletion of a process In this case, for *process_i* to be deleted from *T* it must have received all pending *DISCONNECT* messages from its children (if any), *cc_i* must be = 0 according to *Rules 2* and *3* and *node_i* must be *passive*.

This means that in order to send the *DISCONNECT* message, a process must be a leaf (no children) in the spanning tree. It also means that (*node_i* is *passive*) after *process_i* sends such a message to *process_{PA}* it then becomes *inactive*. *Q* is still maintained because *T* is still rooted in the *Detector* and all live processes are still in *T*.

Insertion of a process i In this case, the process i must be currently *inactive* according to *Rule 1* because it makes the process that activated it its new parent. Since initially all the processes belonged to T and were all *live*, i must have previously become *inactive* and had been deleted from T . The $node_i$ must have received an activation message from another $node_j$ ($i \neq j$). The $node_j$ must have been *active* in order to send a message, according to the basic model described above. This implies that $process_j$ must have been also in T . The $process_i$ now makes $process_j$ its parent in T . The activated $process_i$ is thus become *live* and in T . In all this, Q is still preserved.

Inductive Hypothesis: We assume that Q is preserved in all different configurations in the running of the algorithm. In this case, all that is happening is that the *branches* of T are randomly, becoming short and long in a systaltic fashion.

Inductive Step: In this case, we discuss the situation where the *Detector* has become *inactive* and we prove that Q is still preserved. We prove this by *contradiction*. Lets assume that the *Detector* has become *inactive* and there is still a message in transit. Lets assume that this message was sent by $process_j$. The $process_j$ must have been *live* in order to send the message. PA_j must either be non-empty (a contradiction because the path along the parent of j must be rooted at the *Detector* but the *Detector* has no current children) or j must be the *Detector*, also a contradiction because the *Detector* is *inactive*. Therefore, when the *Detector* becomes *inactive*, there are no messages in transit and the whole computation has terminated.

□

Theorem 6.2 (Liveness) When termination occurs then it is detected in finite time.

Proof:

When all the processes that $process_i$ has caused (directly or indirectly) to become *live* have become *inactive* then $process_i$ receives all the outstanding *DISCONNECT*

<i>Alg. Ref.</i>	<i>Message Complexity</i>	<i>Channel Type</i>	<i>Global Storage</i>	<i>Comp. Initiation</i>	<i>Comments</i>
[60]	$b + n\epsilon$	-	$O(n)$	Multiple	Waves
This thesis	$b + 3(n - 1)$	-	$O(n)$	Multiple	1 Wave

Table 6.1: Worst-Case Complexities of Optimal TD Algorithms

messages in finite time according to Rule 3. If $node_i$ has also become *passive* then $process_i$ sends a *DISCONNECT* message to $process_{PA}$ (if not empty) and becomes *inactive* in finite time, according to the same rule. This activity continually converges towards the *Detector* as successive processes become *inactive*. This means that the *Detector* receives *DISCONNECT* messages from all its children in finite time, the order in which this happens is not important. It follows therefore, that when termination occurs, the *Detector* knows in finite time. $\square$

6.5 Message Complexity

In this section we discuss the message complexity of the algorithm.

Note: n =number of nodes, ϵ =edges and b =number of basic communications

Theorem 6.3 *The algorithm has the worst case message complexity of $b + 3(n - 1)$, shown in Table 6.1.*

Proof:

By definition, every message that is sent between any pair of *nodes* gets either an acknowledgement or a *DISCONNECT* message. Since b is the number of basic communication messages that are transmitted during a computation, the algorithm needs a total of b acknowledgement messages, this includes *DISCONNECT* messages for all terminations (state changes of processes from *live* to *inactive*) that transpire excluding the $(n - 1)$ that disconnect the initial spanning tree.

The initialization of the detection spanning tree takes a maximum of $2(n - 1)$ messages in the case of multiple initiators and $(n - 1)$ messages when we have a

single initiator in the monitored computation, we use $2(n - 1)$ because it is the worst case.

DISCONNECT messages are either sent along the original paths in the initial spanning tree or they are sent along the new dynamically created paths as the computation precedes. When the computation eventually terminates then *DISCONNECT* messages must have been sent also along the initial spanning tree paths to disconnect the initial commitments. This takes $(n - 1)$ messages.

In the worst case, the sum of all the above mentioned messages is $b + 3(n - 1) + 1$.

6.6 Summary of the Algorithm

6.6.1 Assessment of the Algorithm

The algorithm that was presented here is very similar to that of Shavit and Francez[60] and has the same advantages. The algorithm improves on the Shavit and Francez one with respect to the number of messages used for the detection of termination, the Shavit-Francez algorithm uses $n.c$ messages while this algorithm uses only $3(n - 1)$.

The following assessments apply to both the presented algorithm and the Shavit-Francez one. Other termination detection algorithms have the possibility of unsuccessful probes to detect termination, for example, the algorithm by Marzullo and Sabel [41] and that of Mattern [43]. The presented algorithm does not have the possibility of an unsuccessful termination detection probe. The presented algorithm is more optimal than most graph marking algorithms because they need to traverse all the edges in a graph, i.e. the Euler cycle of the network, for example [45].

Whereas most other algorithms use a number of different messages, this one uses only two types of messages e.g. [26, 27]. Most other algorithms have considerable global storage but this algorithm's total global storage is limited to $O(n)$ integers. Half of the global storage is static and is only needed for the initialization of the termination detection.

The algorithm requires no *FIFO* channels as do others, see Tables 5.1 and 6.1. The algorithm does not have a restriction on the network topology as others e.g. [31, 12] that require a specific network topology such as a ring or a complete network.

6.6.2 Fault Tolerance Considerations

We did not implicitly consider fault tolerance in this algorithm so as to bring out its basic principles. In this section we do consider the algorithm in the context of fault tolerance. We consider fault tolerance only for a complete graph in which the spanning tree is a *star* topology with the *Detector* in the center. We consider the same kinds of faults and therefore use the same techniques as those applied in the Mutual Exclusion protocol of Part II to make the algorithm fault tolerant.

6.7 Summary of Termination Detection

The general observation is that most algorithms do not address the issue of message complexity and a few address fault tolerance. Token-based algorithms deal more with the fault tolerance issue than the others. The technique of time-outs seems to be the main technique used in non-token-based algorithms to effect fault tolerance.

Judging from the published papers, it also seems that the research on Termination Detection has tapered off in the recent years. It has been stated that no new results can be expected in this problem and that research should focus on other problems. The new algorithm that we presented disproves this.

Termination detection has been shown to be related to the areas of Deadlock Detection, Election and Total Algorithms [30] and it seems as if the area of Termination Detection is related to many other areas of distributed computing. We believe that reducing any distributed computing problem to that of Termination Detection helps in understanding it and also in its verification because of the straight forwardness of the termination detection problem.

Thesis Conclusions

Chapter 7

Thesis Conclusions

This thesis dealt with two areas of distributed computing, namely, *Termination Detection* and *Mutual Exclusion*. A number of other related issues were considered and two new and optimal protocols were presented.

We started out by discussing the general distributed computing environment. This we did in preparation of the main areas of the thesis. We introduced a notion of standardizing the protocol-evaluation techniques by laying down aspects which each protocol must take into consideration.

Following that we crossed over to the area of distributed Mutual Exclusion. Here we briefly studied some of the protocols published in this area. We went on to present a new protocol for distributed mutual exclusion. We proved the protocol to be correct, efficient and fault tolerant.

The protocol we presented for mutual exclusion used the principle of a dynamic coordinator. Here we had a node playing the special role of Mutual Exclusion Coordinator. Every node was potentially capable of becoming the coordinator. The protocol exhibits very low worst case message complexities. The protocol was for single entry into the critical section and we showed the ease with which it is extended to allow for multiple entries into the critical section.

We discussed the protocol in the presence of faults. We had imposed a restriction on the nature of faults, i.e. the fact that faults subside to allow a reorganization. This is a necessary condition because an endless cascade of faults means that finite

delays cannot be guaranteed. If finite delays cannot be guaranteed no protocol can work. We then discussed the area of termination detection. We briefly considered previous work in this area. We went on to present a new efficient termination detection protocol.

In the area of Termination Detection we showed that an efficient protocol which does not use clocks or unbounded counters can be optimal. The protocol was based on a spanning tree and though it is similar to that of Shavit and Francez [60] we showed that it improved on their results by reducing the number of detection messages from $n\epsilon$ (where ϵ is the number of edges) to $3(n-1)$ in the worst case. We also considered the introduction of fault tolerance into the protocol.

7.1 Future Work

There have been papers published which discuss how a protocol to solve one problem of distributed systems can be transformed or migrated to solve another [66, 16, 30]. Unfortunately there has not been much research effort put into this area although the benefits are obviously invaluable. An ideal situation would be the discovery of a universal transformation procedure which could efficiently transform protocols from one area of distributed computing into any other area. This goal is still very distant in the light of current research but we believe that it is feasible. Message passing systems share many important aspects and they are bounded by similar conditions. Another reason why research in this area is important is the fact that distributed protocols are usually meant to be building blocks whereby the whole system can be made resilient and more efficient. In a case where different protocols can be grouped together to form a larger more comprehensive protocol the elimination of redundancy becomes important.

New areas emerge as old areas become refined, split and specialized. In such cases it is often observed that similar patterns of research are repeated in different areas. Sometimes incorrect protocols are published because of limited tests performed on them or the lack of standardized acceptance procedures. If a universal

transformation procedure would exist then it would be easier to test new protocols and to generalize them.

We are aware that the realization of a universal distributed protocol linkage procedure cannot be done overnight but we wish to add on to this kind of research in this chapter. We believe that this kind of research brings about a standardization whereby it would be harder to publish incorrect or inefficient protocols. As part of our future work we wish just to present the linkage problem with respect to distributed mutual exclusion and distributed termination detection.

Most of the linkage research known to the author has involved the areas of *Termination Detection* and *GraphMarking*. These areas are implicitly related such that the transformation procedures used on them are hardly applicable to other areas. The research into the area of linkage is similar to the research into *NP - Complete* problems whereby a mapping procedure is defined such that a solution to one problem would imply a solution to another. In linkage we already know that solutions exist for each individual area. We wish to define a mapping procedure such that an efficient protocol in one area implies an efficient (or at least a viable) solution to another.

Termination Detection answers the question *Are you terminated ?* with a boolean answer of *Yes* or *No*. Mutual Exclusion does not exhibit a clear link to termination detection because it does not answer a question directly. Mutual exclusion returns a schedule or a sorted list of processes in which ideally each process appears only once although repetitions are possible. Establishing a link (or the lack of it) between the two areas is an areas of future research.

Sometimes it is easier to establish one-way links between different areas than to establish two-way links. We believe that even the establishment of one-way links is a step forward. We believe that it is possible to establish methods of migration through which protocols in one area can be transformed into those of another. Because it is easier to test protocols of some areas (e.g. Termination Detection) than it is to test others (e.g. Group Membership) a migration method could make it easier to verify complex protocols.

In the course of our research we encountered published protocol that had flaws in them (see both Singhal protocols above). This stresses the need for a universal protocol verification and evaluation procedure. In future we wish to explore the possibility of the formal definition of such a procedure. As an example, one minor feature of such a procedure would require a protocol to show the local storage involved for each process and the size of messages involved. We believe that the establishing of links between different distributed control aspects can speed up the process of developing and verifying new protocols. We believe that any work in the above mentioned future research is of vital importance in finding more robust and more efficient solutions.

Appendix A

Pseudocode Examples

In this chapter we present pseudocode examples of the *Dynamic Coordinator* protocol and the *Termination Detection* protocol, both presented in Parts II and III respectively. For the mutual exclusion protocol we assume that the complete network topology is present and a spanning-star is already in place where the center is the current *MEC*. For the termination detection protocol we assume an arbitrary network topology. Please note that the processes and tasks (states) that we describe below are highly interleaved in the real situation. This is because faults can occur at any time, even while other recovery procedures are underway. We have not included this aspect of overlapping for the sake of clearness.

A.1 Coordinating the Mutual Exclusion

The current *MEC* plays a special role in the mutual exclusion. Here we present the code that pertains only to that special role. The *MEC* would also perform other tasks just as every other process and would therefore run the additional programs as well. Note that any node in the system is capable of becoming the *MEC*.

```
PROCESS coordinate-me()
```

```
begin
```

```
    /* local variables and message declaration */
```

```
    STATE initial:
```

```
queue = empty;
predecessor = mec;
rank = 0;
queue-head = 1;
mec-predecessor = X;
mec-successor = Y;
current-in-cs = NONE;
wait(request);
```

STATE handling-request:

```
predecessor = requesting-Id;
rank = rank+1:mod(n);
queue.id[rank] = requesting-Id;
queue.rank[rank] = rank;
if(length(queue)<2) begin
    current-in-cs = queue.id[queue-head];
    remove-entry(queue,rank);
    queue-head = queue-head + 1;
    send(current-in-cs,permit);
else send (requesting-proc,reply,rank,predecessor);
enter(granting-delayed-permit);
```

STATE granting-delayed-permit:

```
wait(finished(current-in-cs) OR
    failed(current-in-cs));
current-in-cs = queue.id[queue-head];
remove-entry(queue,queue-head);
send(current-in-cs,permit);
if(request-arrives) enter(handling-request);
```

STATE receiving-failed-msg:

```
remove-entry(queue[sender-Id-1]);
```

```

        if(queue-head < queue.rank[sender-rank]) begin
            find-new-predecessor;
            send(detecting-proc, new-predecessor);
        end else
            enter(handling-request((permit, current-in-cs)));
    end
end

```

A.2 Going Into Critical Section

We do not concern ourselves with the underlying computation that does not involve the going into the *CS*. We have grouped that sort of computation under the state of non-*CS* computation, although this lies within the task for going into *CS*.

```

PROCESS going-into-cs()
    begin
        /* local variables and message declaration */

        STATE requesting:
            send(mec, request, own-Id);
            enter(stand-by);

        STATE stand-by:
            if(receive(stand-by-msg)) begin
                store(rank, predecessor);
                enter(stand-by);
            end
            if(receive(permit)) begin
                enter <CS>;
                delete(rank, predecessor);
                send(mec, finished);
            end

        STATE non-ME computation:
            compute until CS is required;
    end

```

```
        enter(requesting);  
end
```

A.3 Process Failures

Any process can fail at any time. A cascade of *MEC*-failures can even take place. Each process starts to tackle the failures known to itself until that handling process is either interrupted by another failure or until the normal computation can proceed.

```
PROCESS another-process-fails()  
begin  
    /* local variables and message declaration */  
  
    STATE cs-predecessor-fails:  
        send(mec,lost-msg,own-Id,rank)  
        wait(receive(new-predecessor);  
  
    STATE mec-predecessor-fails:  
        poll backwards until  
            new mec-predecessor located;  
  
    STATE inactive-process-fails:  
        block link to failed process;  
  
    STATE mec-fails:  
        if(other-recovery-in-progress) interrupt;  
        if(own-Id = mec-successor(failed-Id) begin  
            become(mec);  
            announce(mec-successor,new-mec);  
            announce(mec-predecessor,new-mec);  
            collect(ranks-and-ids);  
            START coordinate-me();  
        end else if(exists(mec-successor)) wait(announce);  
end
```

A.4 Recovery of A Crashed Process

A recovering process does not initially know the current *MEC* and it has to find out by first inquiring from its successor in the *MEC succession chain*. If the successor is also recovering it first tries to find out the *MEC - Id* from its successor ...etc. There is a process which is the current end or *tail* of the succession chain. This process starts the *MEC*-search in the opposite direction. There is also a process which is the current head of the chain. This process assumes the *MEC* position as soon as it receives the search message. This means that all the currently active processes recovered at about the same time and all the previously active ones (including the then *MEC*) have just crashed. This is the worst case.

PROCESS recovery-of-failed-process()

begin

/* local variables and message declaration */

STATE seeking-mec:

send(mec-successor, mec-seek-msg);

wait(mec-Id OR seek-msg);

if(received(announce OR mec-Id) AND
(asked(predecessor)))

begin

send(predecessor, mec-Id)

end else begin send(mec-predecessor, mec-seek-msg);

wait(mec-Id OR announce);

end if(received(announce OR mec-Id) AND
(asked(mec-successor)))

send(mec-successor, mec-Id)

START (going-into-cs());

end

Appendix B

Code for the Optimal Termination Detection Algorithm

The following is the code executed by every process in the termination detection algorithm presented in Part III.

```
PROCESS td()
begin /* for process i and any other process is j,
      this same code is executed by every process */
    int cc=CHILDREN;
    Id PA=PARENT;
    bool state=active, p-state=live, CO=true;

    STATE initial:
        cc=CHILDREN; PA=PARENT; state=active;
        p-state=live; CO=true;

    STATE send:
        /* state = active, p-state = live */
        send(msg,i); cc = cc + 1; enter(local);

    STATE receive:
        /* (msg,j) arrived at i */
```

Appendix B. Code for the Optimal Termination Detection Algorithm102

```
receive(msg,j); state = active;
if(p-state==inactive) begin
    PA=j; p-state=live; enter(local);
end else send(DISCONNECT,j);
```

STATE inactive:

```
/* p-state = live */
if(state==passive) begin
    PA=empty; p-state = inactive;
    send(DISCONNECT,PA);
    if(NOT(Detector) enter(terminated);
    else enter(detected);
end else enter(local);
```

STATE live:

```
/* A DISCONNECT arrives at i */
receive(DISCONNECT,j); cc = cc - 1;
if(cc==0 AND state==passive) enter(inactive);
else enter(local);
```

STATE passive:

```
/* a node becomes spontaneously passive */
enter(local);
```

STATE detected:

```
/* announce termination */
```

STATE local:

```
/* local processing and other alternating states */
enter(send|receive|inactive);
```

STATE terminated:

Appendix B. Code for the Optimal Termination Detection Algorithm103

```
wait(arrival(msg,j));  
/* could be termination announcement or activation  
message, and enter appropriate state */  
end
```


References

- [1] Gregory R. Andrews. Paradigms for process intercation in distributed programs. *A.C.M. Computing Surveys*, 65-92, March 1991.
- [2] Maurice J. Bach. *The design of the UNIX Operating System*. Prentice Hall, 1986.
- [3] Giafranco Balbo and Giovanni Chiola. An example of modeling and evaluation of a concurrent program using colored stochastic petri nets: lamport's fast mutual exclusion algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 3(2), March 1992.
- [4] Amotz Bar-Noy, Danny Dolev, Daphne Koller, and David Peleg. *Fault Tolerant Critical Section Management in Asynchronous Networks*. Technical Report, Stanford University, April 1989.
- [5] Jose' M. Bernabeu-Auban and Mustaque Ahamad. *Applying a Path-Compression Technique to Obtain Efficient Distributed Mutual Exclusion Algorithm*. Technical Report, GIT Atlanta, 1988.
- [6] Uyless D. Black. *Data Networks, concepts, Theory and Practice*. Prentice Hall, 1989. OSI stack ...
- [7] Werner Brecht. *Verteilte Systeme unter UNIX: Eine Praxisorientierte Einführung*. Vieweg, 1992.
- [8] J.G. De Bruijn. Additional comments on a problem in concurrent programming control. *Communications of the A.C.M.*, 10(3):137-138, 1967.

- [9] J.E. Burns. Symmetry in systems of asynchronous processes. In *Proc. 22nd Annual Symp. on Foundations of Computer Science*, pages 169–171, October 1981.
- [10] O. Carvalho and G. Roucairol. On mutual exclusion in computer networks. *Communications of the A.C.M.*, 26(2):146–147, 1983.
- [11] K.M. Chandy and J. Misra. Distributed computing on graphs: shortest path algorithms. *Communications of the A.C.M.*, 25(11):833–837, November 1982.
- [12] Ye-In Chang, Mukesh Singhal, and Ming T. Liu. A dynamic token-based distributed mutual exclusion algorithm. In *Proc. Ninth Symp. Reliable Distributed Sys.*, pages 240–246, October 1990.
- [13] S. Cohen and D. Lehmann. Dynamic systems and their distributed termination. *A.C.M. SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, 29–33, August 1982.
- [14] P.J. Courtois and F. Heymans. Concurrent control with readers and writers. *Communications of the A.C.M.*, 14(10):667–668, 1971.
- [15] O.J. Dahl and E.W. Dijkstra. *Structured Programming*. Academic Press, London, 1972.
- [16] E. Dijkstra, W.H.J. Feijen, and A.J.M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 217–219, June 1983.
- [17] E.W. Dijkstra. Hierarchical ordering of sequential processes. *Acta Informatica*, 1(2):115–138, 1971.
- [18] E.W. Dijkstra. *Programming Languages: Cooperating Sequential Process*. Academic Press, New York, 1965.
- [19] E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the A.C.M.*, 17(11):643–644, 1974.

- [20] E.W. Dijkstra. The structure of the multiprogramming system. *Communications of the A.C.M.*, 11(5):341-346, 1968.
- [21] E.W. Dijkstra. Termination detection for diffusing computations. *Inf. Proc. Lett.*, 11(1):1-4, 1980.
- [22] E.W. Dijkstra and C.S. Scholten. Termination detection for diffusing computations. *Inf. Processing Letters*, 1-4, August 1980.
- [23] R.W. Doran and L.K. Thomas. Variants of the software solution to mutual exclusion. *Inf. Proc. Lett.*, 10(4):206-208, 1980.
- [24] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of A.C.M.*, 32(2):374-382, April 1985.
- [25] Michael J. Fischer. A theoretician's view of fault tolerant distributed computing. *Communications of the A.C.M.*, 448:1-9, 1987.
- [26] N. Francez. Distributed termination. *A.C.M. Toplas*, 42-55, January 1980.
- [27] Nissim Francez and Michael Rodeh. Achieving distributed termination without freezing. *IEEE Transactions on Parallel and Distributed Systems*, 287-293, May 1982.
- [28] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning trees. *A.C.M. Trans. on Programming Languages and Systems*, 5(1):66-77, 1983.
- [29] A. Gottlieb and B.D. Lubachevsky. Basic techniques for the efficient coordination of very large numbers of cooperating sequential processors. *A.C.M. Toplas*, 5(2):164-189, 1983.
- [30] G. Tel, R.B. Tan, and J. VanLeeuwen. The derivation of graph marking algorithms from distributed termination detection protocols. *Information Processing Letters*, 10:107-137, October 1988.

- [31] S. Haldar and D.K. Subramanian. Ring based termination detection algorithm for distributed computations. *Information Processing Letters*, 149-153, October 1988.
- [32] J.M. Helary, N. Plouzeau, and M. Raynal. A distributed algorithm for mutual exclusion in an arbitrary network. *The Computer Journal*, 289-295, 1988.
- [33] Shing-Tsaan Huang. Termination detection by using distributed snapshots. *Information Processing Letters*, 113-119, August 1989.
- [34] H. Hyman. Comments on a problem in current programming. *Communications of the A.C.M.*, 45, 1966.
- [35] A. Kaneko and Y. Nishihara. Logical clock synchronization method for duplicated data base control. In *Proc. of the Conf. on Distributed Computing Systems*, pages 601-611, 1979.
- [36] D.E. Knuth. Additional comments on a problem in concurrent programming control. *Communications of the A.C.M.*, 321-322, 1966.
- [37] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the A.C.M.*, 558-565, 1977.
- [38] G. Le Lann. Algorithms for distributed data sharing systems which use tickets. In *Proc. of the 3rd Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 259-272, 1978.
- [39] Mamuro Maekawa. A $\sqrt{N}$ algorithm for mutual exclusion in decentralized systems. *A.C.M. Trans. Computer Systems*, 3:145-159, May 1985.
- [40] Alain J. Martin. A new generalization of dekker's algorithm for mutual exclusion. *Communications of the A.C.M.*, 295-297, December 1985.
- [41] K. Marzullo and L. Sabel. *Using Consistent Subcuts for Detecting Stable Properties*. Technical Report, Cornell University, May 1991.

- [12] F. Mattern. Global quiescence detection based on credit distribution. *Information Proc. Letters*, 195-200, February 1989.
- [13] Friedmann Mattern. An efficient distributed termination test. *Information Processing Letters*, 203-208, May 1989.
- [14] P.M. Mellar Smith, Louise E. Moser, and Vivek Agrawala. Broadcast protocols for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 17-25, January 1990.
- [15] J. Misra. Detecting termination of distributed computations using markers. In *Proc. of the 2nd A.C.M. Conf. on Principles of Distributed Computing, Montreal*, pages 290-294, August 1983.
- [16] J. Misra and K.M. Chandy. A distributed graph algorithm: knot detection. *A.C.M. Trans. on Prog. Lang. and Syst.*, 678-686, October 1982.
- [17] J. Misra and K.M. Chandy. Termination detection of diffusing computations in csp. *A.C.M. Toplas*, 37-43, January 1982.
- [18] M. Naïmi and M. Trehel. *Analysis of an Efficient Distributed Algorithm for Mutual Exclusion (Average-Case Analysis of Path Reversal)*. Technical Report. Re. Rep., R.R. du Laboratoire d'Informatique de Besancon, 1988.
- [19] Shojiro Nishio, Kin F. Li, and Eric G. Manning. A resilient mutual exclusion algorithm for computer networks. *Information Processing Letters*, 163, October 1990.
- [50] S. P. Rana. A distributed solution of the distributed termination problem. *Information Processing Letters*, 43-46, July 1983.
- [51] Sampath Ranganathan and Satish K. Tripathi. *A Robust Distributed Mutual Exclusion Algorithm*. Technical Report, University of Maryland (College Park), 1991.

- [52] Kerry Raymond. A distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 189–193, February 1989.
- [53] Kerry Raymond. A tree-based algorithm for distributed mutual exclusion. *A.C.M. Trans. Comput. Syst.*, 7:61–77, 1989.
- [54] M. Raynal. *Algorithms for Mutual Exclusion. Series in Computer Systems*. MIT Press, Cambridge Mass., 1986.
- [55] Michel Raynal. *Distributed Algorithms and Protocols*. John Wiley and Sons, 1988.
- [56] G. Ricart and A.K. Agrawala. Author's response to 'on mutual exclusion in computer networks' by carvalho and roucairol. *Communications of the A.C.M.*, 147–148, 1983.
- [57] Glenn Ricart and Ashok K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *Communications of the A.C.M.*, 24:9–17, January 1981.
- [58] Beverly A. Sanders. The information structure of distributed mutual exclusion algorithms. *A.C.M. Trans. Computer Systems*, 5:284–299, August 1987.
- [59] R. Satyanarayanan and D.R. Muthukrishnan. A note on raymond's tree based algorithm for distributed mutual exclusion. *Inf. Proc. Letters*, 249–255, October 1992.
- [60] N. Shavit and N. Francez. *A New Approach to Detection of Locally Indicative Stability*. Technical Report, The Technion, Haifa, 1986.
- [61] Mukesh Singhal. *Construction of Optimal Dynamic-Information-Structure Mutual Exclusion Algorithms for Distributed Systems*. Technical Report, Ohio State University, 1990.

- [62] Mukesh Singhal. A dynamic information-structure mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 121-125, January 1992.
- [63] Pradip K. Srimani, L. Rachamalla, and N. Reddy. Another distributed algorithm for multiple entries to a critical section. *Information Processing Letters*, 51-57, January 1992.
- [64] I. Suzuki and T. Kasami. An optimality theory for mutual exclusion algorithms in computer networks. In *Proc. of the 3rd Conf. on Distributed Computing Systems, Miami*, pages 365-370, 1982.
- [65] Ichiro Suzuki and Tadao Kasami. A distributed mutual exclusion algorithm. *Communications of the A.C.M.*, 344-349, November 1985.
- [66] Gerard Tel. *The Structure of Distributed Algorithms*. PhD thesis, University of Utrecht, February 1989.
- [67] E.B. Weidman, I.P. Page, and W.J. Pervin. Explicit dynamic exclusion algorithm. In *Proc. Ninth Symp. Reliable Distributed Sys.*, pages 142-149, October 1990.
- [68] Tai-Kuo Woo and Richard Newman-Wolfe. Huffman trees as a basis for a dynamic mutual exclusion algorithm for distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 126-133, January 1992.

END

1 6 - 1 2 - 9 3

FIN