Studies in Systems, Decision and Control 116

Sukhendu Kanrar
Nabendu Chaki
Samiran Chattopadhyay

Concurrency Control in Distributed System Using Mutual Exclusion

Springer

Studies in Systems, Decision and Control

Volume 116

Series editor

Janusz Kacprzyk, Polish Academy of Sciences, Warsaw, Poland
e-mail: kacprzyk@ibspan.waw.pl

Sukhendu Kanrar · Nabendu Chaki
Samiran Chattopadhyay

Concurrency Control in Distributed System Using Mutual Exclusion

Springer

Sukhendu Kanrar
Department of Computer Science
Narasinha Dutt College
Howrah, West Bengal
India

Samiran Chattopadhyay
Department of Information Technology
Jadavpur University
Kolkata, West Bengal
India

Nabendu Chaki
Department of Computer Science & Engineering
University of Calcutta
Kolkata, West Bengal
India

ISSN 2198-4182 ISSN 2198-4190 (electronic)
Studies in Systems, Decision and Control
ISBN 978-981-10-5558-4 ISBN 978-981-10-5559-1 (eBook)
DOI 10.1007/978-981-10-5559-1

Library of Congress Control Number: 2017945702

Printed on acid-free paper

This Springer imprint is published by Springer Nature
The registered company is Springer Nature Singapore Pte Ltd.
The registered company address is: 152 Beach Road, #21-01/04 Gateway East, Singapore 189721, Singapore

Preface

The broad area of research presented in this book is designing operating systems (OS) for distributed systems. The advantages of a distributed system over a traditional time-sharing mainframe system depend a lot on the underlying operating system. The expected benefits of a distributed computing platform include distribution transparency, improved price/performance ratio, better system response through load distributing, higher dependability, etc. This research area has always been fascinating to explore. Hence, operating systems, as a whole, forms the broad domain of the research presented in this book while the book focuses on process synchronization for a distributed system.

Existing mutual exclusion (ME) algorithms are often either symmetric or token-based. Token-based approaches offer solutions at relatively lower communication cost. One major limitation of the token-based mutual exclusion algorithms for distributed environment like Raymond's well-known work is inability in handling the processes with pre-assigned priority values. Besides, the natural fairness in terms of first-come-first-serve allocation among equal priority processes too is not guaranteed in Raymond's algorithm. This has been the motivation of the first work discussed in this book. In the book, we discussed a modification of Raymond's well-known algorithm to ensure fairness in terms of first-come-first-serve (FCFS) order among equal priority processes. Subsequently, it was improved and named *Modified Raymond's Algorithm with Priority* (MRA-P). The solution considers the priority of the requesting processes and allocates resource for the earliest request when no such request from a higher priority process is pending. In MRA-P, we introduced a pair of local queues whose union would resemble a single global queue.

However, MRA-P suffered from some major shortcomings like lack of liveness, high message complexity, etc. In the next step, we further improved on these and described a token-based *Fairness Algorithm for Priority Processes* (FAPP) that addresses both the issues. FAPP justifies properties like correctness of the algorithm, low message complexity, and fairness in token allocation. Appropriate simulation has been done to benchmark FAPP and MRA-Pwith other existing algorithms.

Subsequently, the book describes design of a novel mutual exclusion algorithm that would be compatible to a more flexible underlying topology than the inverted tree topology on which Raymond's original algorithm as well as both MRA-P and FAPP work. It is found that there exists ME algorithms that work on a directed, acyclic graph (DAG). However, the next algorithm discussed in the book named *Link-failure Resilient Token based ME Algorithm on Graphs* (LFRT) works on any directed graph topology, with or without cycles. Like MRA-P, the LFRT algorithm ensures liveness, safety, and fairness in allocating the token on a FCFS basis. The most significant advantage of the LFRT algorithm is its ability to handle the link failures. This is possible due to redundancy in path in the underlying graph topology. LFRT was further improved to *LFRT for Priority Processes* (LFRT-P) that also considers priority of participating processes.

Besides, token-based solution, the book delves in the area of permission-based mutual exclusion algorithms. There exist efficient approaches in the literature that selects one candidate process from many for allowing it to enter its critical section (CS) on the basis of the number of votes received by the processes. In voting-based approaches, a process that gets majority of the total number of votes is only to be allowed for CS. This ensures safety for such an algorithm as no two processes can earn majority of the total number of polls. However, this may lead to a live-lock situation where no single process reaches the magic number of majority votes. In this book, a voting-based algorithm has been discussed to select a process even when no single process has majority of votes.

Two voting-based algorithms are also described in this book. We have described a voting-based mutual exclusion algorithm (BMaV) that finds a candidate for CS when majority consensus is not achieved by any single process. Another algorithm *A New Hybrid Mutual Exclusion Algorithm in Absence of Majority Consensus* (NHME-AMC) has been discussed in the book. In this book, a two-phase, hybrid ME algorithm has been discussed that works even when majority consensus cannot be reached. The second phase of the algorithm, in spite of being symmetric, executes in constant time. These algorithms aim to find an effective solution when no single process achieves majority consensus. We have concluded the book by discussing on the significance of works in this book towards setting future directions of research for process synchronization in distributed systems.

We hope that researchers in the domain of distributed operating systems and faculty members teaching this subject will find the book a useful reference during treatment of distributed control algorithms. The language of the book is lucid and all algorithms are explained with suitable examples. In spite of our best efforts there may be shortcomings that might have escaped our notice. We shall be obliged if suggestions and criticisms are put forward to improve the content of the book. We also like to gratefully acknowledge Mr. Aninda Bose and Kamiya Khatter without whose continuous support the book could not be completed.

Howrah, India — Sukhendu Kanrar
Kolkata, India — Nabendu Chaki
Kolkata, India — Samiran Chattopadhyay

Contents

About the Authors

Sukhendu Kanrar is a faculty member in the Department of Computer Science, Narasinha Dutt College, India. He has done Bachelors in Mathematics from University of Calcutta in 1999. He received MCA in 2004 and M.Tech. in Computer Science in 2010, both from the West Bengal University of Technology. Dr. Kanrar has completed Ph.D. from the University of Calcutta in early 2016. His primary research interest is in the design of Operating Systems for distributed environment.

Nabendu Chaki is a Professor in the Department Computer Science & Engineering, University of Calcutta, Kolkata, India. Dr. Chaki did his first graduation in Physics from the legendary Presidency College in Kolkata and then in Computer Science & Engineering from the University of Calcutta. He has completed Ph.D. in 2000 from Jadavpur University, India. He is sharing six international patents including four US patents with his students. Prof. Chaki has been quite active in developing international standards for Software Engineering and Cloud Computing as a member of Global Directory (GD) member for ISO-IEC. Besides editing more than 25 book volumes, Nabendu has authored six text and research books and has more than 150 Scopus Indexed research papers in

Journals and International conferences. His areas of research interests include distributed systems, image processing and software engineering.

Dr. Chaki has served as a Research Faculty in the Ph.D. program in Software Engineering in U.S. Naval Postgraduate School, Monterey, CA. He is a visiting faculty member for many Universities in India and abroad. Besides being in the editorial board for several international journals, he has also served in the committees of more than 50 international conferences. Prof. Chaki is the founder Chair of ACM Professional Chapter in Kolkata.

Samiran Chattopadhyay is a Professor in the Department of Information Technology, Jadavpur University. He has served as the Head of the Department for more than 12 years and as the Joint Director of the School of Mobile Computing and Communication since its inception. As a graduate, postgraduate, and gold medalist from Indian Institute of Technology, Kharagpur, he received his Ph.D. from Jadavpur University. He has two decades of experience in serving reputed industry houses such as Computer Associates, Interra Systems India, Agilent, Motorola in the capacity of technical consultant. He led the development of an open-source C++ infrastructure and tool set for reconfigurable computing, released under the GNU GPL 3.0 license. He has visited several Universities in the United Kingdom as a visiting professor. He has been working on Algorithms for Security, Bioinformatics, Distributed and Mobile Computing, and Middleware. He has authored and edited several books and book chapters. Professor Chattopadhyay acted as a program chair, organizing chair, and IPC member of over 20 international conferences. He has published more than 120 papers in reputed journals and international peer-reviewed conferences.

Chapter 1
Introduction

Availability of powerful yet low-cost processors coupled with advances in communication technology has contributed greatly to the development of distributed systems [1–4]. These factors made it possible to design and develop distributed systems comprising many processor nodes, memory nodes, and other computational devices interconnected by communication networks [5–9].

The primary advantages of distributed systems over traditional time-sharing mainframe systems include improved price/performance ratio, easier resource sharing, and better system response through load distribution, higher availability, reliability, and modular expandability.

Multiple areas of distributed operating systems on which some significant amount of work has been done were introduced in recent years. The inherent limitations of distributed systems are often caused by the lack of common memory and a system-wide common clock that can be shared by all the processes. Algorithms for distributed computations are difficult to design, as the events of the computation are occurring at different computers and affect the system state differently depending on the order in which they occur. A distributed operating system uses a set of control algorithms to implement a control function (e.g., mutual exclusion, deadlock handling, process migration, etc.). In this work, we have tried to develop different control algorithms for distributed systems. A control algorithm typically provides a service using a group of resources. It is invoked by the kernel when a node needs its service, or when an event related to its service occurs in the system. A resource allocator, an I/O scheduler are typical examples of control algorithms. Actions for such algorithms are executed in several processes. Often, the data needed for its decision-making is also maintained in multiple processes. Further, the state information available at different processes may be incomplete and/or inconsistent. Use of such information in decision-making or toward action of a control algorithm may lead to inconsistent states. Misinterpretation of incomplete data may even result in contradictory and conflicting actions, initiated at different processes in the system.

Recording global state information is a limitation for any distributed system. However, the concern for consistency cannot be addressed easily in a distributed environment due to lack of this global state information. That is the fundamental difference between control algorithms used in distributed and centralized operation systems.

Resource sharing is an important aspect of the real-time distributed systems. Some resources are inherently non shareable and must be accessed in a mutually exclusive way. The traditional approaches towards implementing mutual exclusion (ME) cannot be applied to distributed systems where nodes are loosely coupled.

In a distributed OS, a distributed control algorithm and its processes may execute concurrently on different processors in a single node or in multiple nodes of a system. A well-known problem in the field of distributed algorithms is the mutual exclusion problem. It consists of a set of processes which communicate via message passing and need to exclusively access a shared resource by executing a segment of code called the critical section (CS). A mutual exclusion algorithm must thus ensure that exactly one process can execute the critical section at any given time (safety property) and that all critical section requests will eventually be satisfied (liveness property). The problem of mutual exclusion in a single-computer system with shared memory is quite different from that in distributed systems where both the shared resources and the users may be distributed [10–13]. Consequently, approaches based on shared variables are not applicable to distributed system and approaches based on message passing must be used. Hence, design of distributed mutual exclusion algorithm involves a familiar concern—how to ensure consistency of data structures and actions in various processes of the system? Some obvious aspects of this concern are mutual exclusion over data and synchronization between actions.

Typically, ME algorithms ensure that at most one process enters its critical section (CS) at a particular instance of time [14–16]. Traditional ME algorithms [3, 17–19] often cannot be applied for distributed systems where nodes are loosely coupled. Over the last decade the problem of mutual exclusion has received considerable attention and several algorithms to achieve mutual exclusion in distributed systems have been suggested. These tend to differ in their communication topology (e.g., tree, ring, and any arbitrary graph) and in the amount of information maintained by each site about other sites.

Solutions to the mutual exclusion problem can be either centralized or distributed. However, a centralized approach with a single coordinator is not suitable for any distributed system. A distributed algorithm where each node participates in the decision-making is more appropriate. Distributed mutual exclusions are either token-based or non-token-based. In token-based mutual exclusion algorithms, a unique token exists in the system and only the holder of the token can access the protected resource.

Existing ME algorithms typically follow either a permission-based [5, 20, 21] or a token-based [17, 20, 22, 23] approach. While the permission-based algorithms tend to increase the network traffic, token-based approach offers solutions at a lower communication cost. There exist different ME algorithms for message passing distributed systems [11, 13, 14, 20, 21, 24, 25]. Some of these algorithms have been fine tuned to suit the needs of real-time systems [22, 26–28].

Raymond has designed an efficient token-based ME algorithm. Raymond's algorithm [28] assumes an inverted tree topology. However, one major limitation of Raymond's algorithm is the lack of fairness in the sense that token requests from two equal priority processes may be processed and granted access to CS out of turn. Here, the primitive idea of fairness has been assumed in terms of ensuring first-come-first-serve (FCFS). However, this definition of fairness is often challenged depending on the context and importance of a process. Considering priority of processes is in direct conflict with the FCFS fairness definition. An efficient ME algorithm for a distributed system needs to strike a balance between such conflicting characteristics. This problem is taken up in the design of fairness algorithm for priority processes (FAPP) [29].

In some of the earlier works, token-based algorithms for ME are developed for the distributed environment assuming inverted tree topology [26, 30]. However, such a stable, hierarchical topology is quite unrealistic for mobile networks where link failures [15, 25, 31–36] are frequent due to node mobility. In this book, two new ME algorithms [31, 37] have been developed on graph topology.

In Ricart–Agrawala [27] permission-based algorithm, each requesting process P_i is allowed to enter the CS only when all of the remaining $N - 1$ competing processes send their concurrences to P_i. On the other hand, the progress condition may suffer due to loss of control messages from any of these $N - 1$ nodes. An alternate approach could be a selection of the process which enters the CS by majority voting. A simple principle that a process that gets majority of the total number of votes is allowed for CS ensures safety as no two processes can earn majority of the total number of polls. However, this may lead to a live-lock situation where no single process reaches the magic number of majority votes. In this book, two new dynamic voting algorithms have been presented to handle such situation [38, 39].

Research contributions presented in this book are depicted in Fig. 1.1. This shows that ME algorithms are divided into two categories: (1) token-based mutual exclusion algorithm and (2) permission-based mutual exclusion algorithm. Token-based algorithms are further divided into two categories: (1) tree-based mutual exclusion algorithm and (2) graph-based mutual exclusion algorithm. We have designed 3 Tree-based Mutual Exclusion algorithms in this work, namely *modified Raymond's algorithm* (MRA) [30], *modified Raymond's algorithm for priority* (MRA-P) [30] and *fairness algorithm for priority processes* (FAPP) [29]. We have also designed two graph-based mutual exclusion algorithms in this work, namely *link failure resilient token-based ME algorithm for directed graph* (LFRT) [37] and *link failure resilient priority-based fair ME algorithm for distributed systems* (LFRT-P) [31]. We also designed two voting-based ME algorithms, namely below-majority voting for ME in distributed systems consensus (BMaV) [38] and *a new hybrid mutual exclusion algorithm in absence of majority consensus* (NHME-AMC) [39].

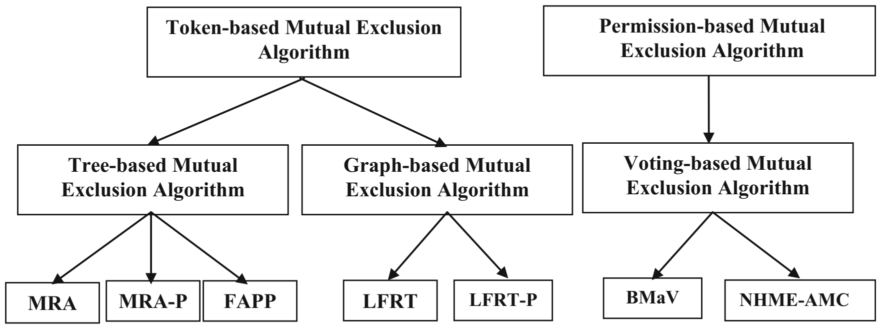

Token-based Mutual Exclusion Algorithm
Permission-based Mutual Exclusion Algorithm
Tree-based Mutual Exclusion Algorithm
Graph-based Mutual Exclusion Algorithm
Voting-based Mutual Exclusion Algorithm
MRA
MRA-P
FAPP
LFRT
LFRT-P
BMaV
NHME-AMC

Fig. 1.1 Novel ME algorithms in the book

1.1 Organization of Book

In Chap. 2, an overview of different types of distributed control algorithms have been discussed specifically on distributed mutual exclusion (DME) algorithms, token-based ME algorithms, tree-based ME algorithms, and graph-based ME algorithms.

Chapter 3 presents our contributions on tree-based mutual exclusion algorithms. The work starts with analysis and limitations of one of the pioneering tree-based ME algorithms like Raymond's algorithm. We have improved on Raymond's algorithm in the form of modified Raymond's algorithm (MRA) to ensure fairness in terms of first-come-first-serve policy. Subsequently, priorities of requesting processes have been taken into consideration in modified Raymond's algorithm for priority (MRA-P) algorithm. The chapter ends with the introduction of another new tree-based ME algorithm called fairness algorithm for priority processes (FAPP) that presents a simpler data structure and hence offers lower complexity.

Designs of graph-based algorithms for ME are introduced in Chap. 4. Solutions of ME problems in graph when one or more link(s) have failed is also worked out in this chapter. Link failure resilient token-based ME algorithm for directed graph (LFRT), a new ME algorithm that works on graph topology and considers priority processes is presented in this chapter. The algorithm is subsequently improved to handle processes with different priorities in link failure resilient priority-based fair ME algorithm for distributed systems (LFRT-P).

Voting-based algorithms are another type of permission-based ME algorithms used in distributed system. A couple of new voting-based algorithms have been presented in Chap. 5. These algorithms find an effective solution when no single process achieves majority consensus. The ME algorithm introduced elects some next process to enter critical section (CS) even in such situations. The concluding comments for each of these works are added at the end of the respective chapters.

References

1. Naimi, M., Trehel, M., Arnold, A.: A log(N) distributed mutual exclusion algorithm based on path reversal. J. Parallel. Distrib. Comput. **34**(1), 1–13 (1996)
2. Sanders, B.: The information structure of distributed mutual exclusion algorithm. ACM Comput. Syst. **5**(3), 284–299 (1987)
3. Agrawal, D., El Abbadi, A.: An efficient and fault tolerant solution for distributed mutual exclusion. ACM Trans. Comput. Syst. **9**(1), 1–20 (1991)
4. Adam, N.R.: New dynamic voting algorithm for distributed database systems. IEEE Trans. Knowl. Data. Eng. Arch. **6**(3), 470–478 (1994)
5. Madria, S.K.: Timestamp based approach for the detection and resolution of mutual conflicts in real-time distributed systems. Computer Science Technical Reports. Paper 1367, pp. 1–16 (1997)
6. Hardekopf, B., Kwiat, K., Upadhyaya, S.: A decentralized voting algorithm for increasing dependability in distributed systems. Join MEETING of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001) and the 7th International Conference on Information System Analysis and Synthesis (ISAS 2001) (2001)
7. Ingols, K.W.: Availability study of dynamic voting algorithms. In: 21st International Conference on Distributed Computing Systems, pp. 247–254, (2001)
8. Maekawa, M.: A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3**(2), 145–159 (1985)
9. Agrawal, D., EL Abbadi, A.: An efficient solution to the distributed mutual exclusion problem. In: Proceeding of the 8th ACM Symposium on Principles of Distributed Computing, pp. 193–200 (1989)
10. Joung, Y.J.: Asynchronous group mutual exclusion. In: Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 51–60 (1998)
11. Suzuki., Kasami, T.: An optimality theory for mutual exclusion algorithms in computer science. In: Proceedings of IEEE International Conference on Distributed Computing and System, pp. 365–370 (1982)
12. Singhal, M.: A heuristically-aided algorithm for mutual exclusion for distributed systems. IEEE Trans. Comput. **38**(5), 70–78 (1989)
13. Bernabeu-Auban, J.M., Ahamad, M.: Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm, vol. 392, pp. 33–44 (1989)
14. Carvalho, O.S.F., Roucairol, G.: On mutual exclusion in computer network. Commun. ACM **26**(2), 146–147 (1983)
15. Sil, S., Das, S.: An energy efficient algorithm for distributed mutual exclusion in mobile Ad-hoc networks. World. Acad. Sci. Eng. Technol. **64**, 517–522 (2010)
16. Barbara, D., Garcia-Molina, H., Spauster, A.: Increasing availbility under mutual exclusion constraints with dynamic vote reassignment. ACM Trans. Comput. Syst. **7**(4), 394–428 (1989)
17. Saxena, P.C., Rai, J.: A survey of permission-based distributed mutual exclusion algorithms. Comput. Stan. Interfaces **25**(2), 159–181 (2003)
18. Toyomura, M., Kamei, S., Kakugawa, H.: A quorum-based distributed algorithm for group mutual exclusion. In: PDCAT'03, pp. 742–746 (2003)
19. Manabe, Y., Park, J.: A quorum based extended group mutual exclusion algorithm without unnecessary blocking. In: Proceedings of 10th International Conference on Parallel and Distributed Systems (ICPADS'04) (2004)
20. Housini, A., Trehel, M.: Distributed mutual exclusion token-permission based by prioritized groups. In: Proceedings of ACS/IEEE International Conference, pp. 253–259 (2001)
21. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
22. Mueller, F.: Prioritized token-based mutual exclusion for distributed systems. In: Proceedings of the 9th Symposium on Parallel and Distributed Processing, pp. 791–795 (1998)

23. Helary, J.M., Mostefaoui, A., Raynal, M.: A general scheme for token and tree based distributed mutual exclusion algorithm. IEEE Trans. Parallel. Distrib. Syst. **5**(11), 1185–1196 (1994)
24. Mittal, N., Mohan, P.K.: A priority-based distributed group mutual exclusion algorithm when group access is non-uniform. J. Parallel. Distrib. Comput. **67**(7), 797–815 (2007)
25. Walter, J.E., Welch, J.L., Vaidya, M.H.: Mutual exclusion algorithm for Ad hoc mobile networks. Wirel. Network **7**(6), 585–600 (2001)
26. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. **7**, 61–77 (1989)
27. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Commun. ACM **24**(1), 9–17 (1981)
28. Lodha, S., Kshemkalyani, A.: A fair distributed mutual exclusion algorithm. IEEE Trans. Parallel. Distrib. Syst. **11**(6), 537–549 (2000)
29. Kanrar, S., Chaki, N.: FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems. Special issue on recent advances in network and parallel computing. Int. J. Networks **5**(1), 11–18 (2010). ISSN: 1796-2056
30. Karnar, S., Chaki, N.: Modified Raymond's algorithm for priority (MRA-P) based mutual exclusion in distributed systems. In: Proceedings of ICDCIT 2006. LNCS 4317, pp. 325–332 (2006)
31. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new link failure resilient priority based fair mutual exclusion algorithm for distributed systems. J. Network. Syst. Manage. (JONS) **21**(1), 1–24 (2013). ISSN 1064-7570
32. Panghal, K., Rana, M.K., Kumar, P.: Minimum-process synchronous check pointing in mobile distributed systems. Int. J. Comput. Appl. **17**(4), 1–4 (2011)
33. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
34. Chen, W., Lin, S., Lian, Q., Zhang, Z.: Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. Microsoft Research Technical Report MSR-TR-2005-58 (2005)
35. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Technical Report in Computer and Communication Sciences, id: 200227 (2002)
36. Walter, J., Cao, G., Mohanty, M.: A k-mutual exclusion algorithm for ad hoc wireless networks. In: Proceedings of the First Annual Work Shop on Principles of Mobile Computing (POMC 2001) (2001)
37. Kanrar, S., Choudhury, S., Chaki, N.: A link-failure resilient token based mutual exclusion algorithm for directed graph topology. In: Proceedings of the 7th International Symposium on Parallel and Distributed Computing—ISPDC 2008 (2008)
38. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new voting-based mutual exclusion algorithm for distributed systems. In: 4th Nirma University International Conference on Engineering (NUiCONE-2013), pp. 1–5 (2013)
39. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new hybrid mutual exclusion algorithm in absence of majority consensus. In: Proceedings of the 2nd International Doctoral Symposium on Applied Computation and security System, ACSS (2015, in press)

Chapter 2
State-of-the-Art Review

A distributed system is a collection of autonomous computers connected via a communication network. There is no common memory and processes to communicate through message passing. In many applications, there are critical operations that should not be performed by different processes concurrently. It may sometimes be required that a typical resource is used by only one process at a time. This gives rise to the problem of mutual exclusion. Mutual exclusion (ME) is crucial for the design of distributed systems.

Due to lack of a common shared memory, the problem of ME becomes much more complex in the case of distributed systems as compared to the centralized systems. It needs special treatment. A number of algorithms have been introduced to solve the ME problem in distributed systems. A good distributed mutual exclusion (DME) algorithm, besides providing ME, should take care that there are no deadlocks and starvation does not occur (Fig. 2.1).

The existing DME algorithms typically follow either a token-based [1–4] or a permission-based [1, 5, 6] approach.

2.1 Definitions of Terminologies

Let us first define the terminologies that we have used in this book. The following definitions clearly mark the scope of each keyword in the context of this book.

Fairness of token-based Algorithms

The definition of fairness that we follow here implies that if the token holder P_{hold} receives a token request from some arbitrary process A ahead of the same from some other process B, then the algorithm ensures that after P_{hold} releases the token, process A gets it ahead of process B.

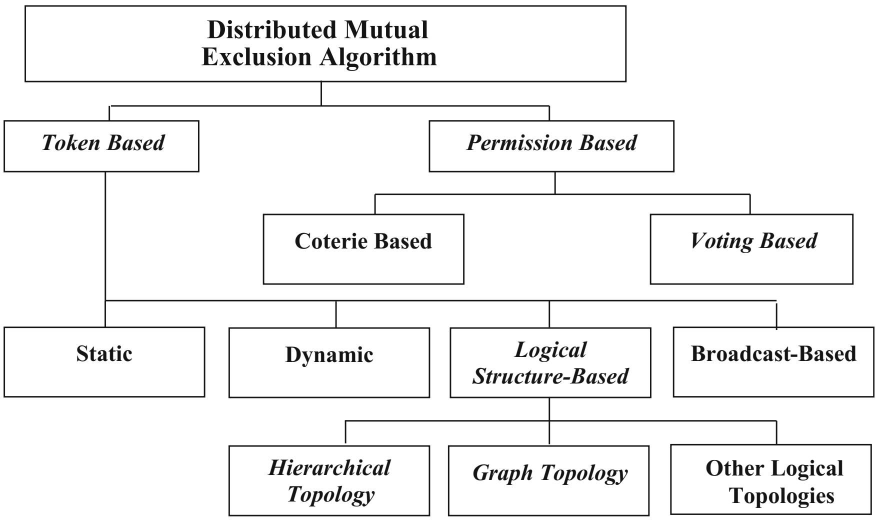


Fig. 2.1 Classification of distributed mutual exclusion algorithm

Liveness
The liveness property implies that any process *A* that requests for the token must get it eventually.

Safety
A mutual exclusion algorithm is safe if it ensures that no two processes would enter the critical section simultaneously.

Correctness
The correctness of a control algorithm is a combination of the liveness along with safety. A control algorithm is correct if it confirms to both liveness and safety aspects. Correctness of an algorithm implies that it should perform correct action and should avoid performing wrong actions. In case of a distributed control algorithm even the former aspect may be difficult to ensure because processes at which control actions are performed lack a global view of the system and its control data.

Conflict Between Priority Processing and Liveness
The definitions of priority-based fairness and liveness are directly in conflict with one another. Let's assume that a process P_A, with a low priority, requests for the token first. After P_A's request, other processes P_B, P_C,…, P_N all with priorities higher than that of P_A, place their requests for the token. In order to ensure the priority-based fairness defined above, all the processes P_B, P_C,…, P_N would receive the token ahead of P_A. In fact, process P_A will never get the token if there is some pending token request from a higher priority process, irrespective of its time of the request. In this situation, the property of liveness is violated. The same had been the major limitation for one of our earlier works, MRA-P [7]. In our attempt to propose a solution that would strike the balance between priority-based fairness and liveness

as defined above, we introduce to dynamically upgrade priorities of token requesting processes.

Priority-based fairness
We assume that the processes have some pre-assigned priorities. It may be desirable to consider the priority of a process while handling the token requests from those. This implies that if the token request from a higher priority process *A* is pending, then the token must not be allotted to processes having priority lower than that of A. Taking into account the priority aspect, we therefore revise the above definition of fairness with a stronger definition below.

The revised definition of priority-based fairness implies that *the token must be allocated to some process A such that among all the processes having priority equal to that of A, the token request from A has reached* P_{hold} *ahead of others and there is no pending token request from any other process B having priority higher than that of A.*

Dynamic Process Priority
In our attempt to propose a solution that would strike the balance between priority-based fairness and liveness as defined above, we introduce to dynamically upgrade priorities of processes waiting in the queue for the token after placing the token request. The initial proposal had been somewhat like the following:

Example
When the token-holding process, say *A*, finds a token request from another process *B* with priority *r* and it is found that *r* is higher than the priority *t* for some previous process *C* whose token request is already stored in the request queue called path list of *A*, i.e., PL_A, then priority of process *C* is increased by 1 to $t + 1$. The list in *A* is re-constructed with this increased process priority for *C*.

The revised value of priority would be updated in the appropriate sorted sequence of pending requests maintained in the P_{hold}, the token-holding process. The motivation behind such a solution was to elevate the priority of a process A every time some higher priority process *B* supersedes *A* by virtue of its higher priority value. This would make sure that even a token request from the otherwise lowest priority process would eventually be satisfied as the priority itself changes dynamically. After a process gets the token, it enters the critical section. As it comes out of the critical section, the priority of a process is reset to its original value.

We explain this solution in Fig. 2.2 with three processes *A*, *B*, and *C* with priority values 4, 4, and 3, respectively. Process *C* of priority 3 places the first request

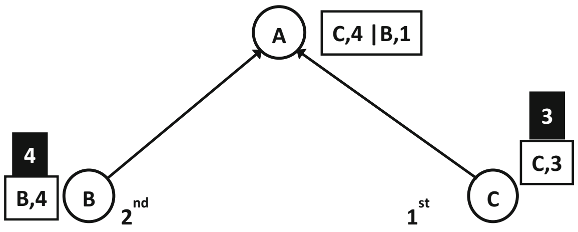


Fig. 2.2 Working principle of dynamic priority

for the token. It stores the 2-tuple in PL_C, i.e., PL_C = <*C*, 3>. This request goes to root process *A* from *C* using 2-tuple <*C*, 3>. Process *A* in root stores the 2-tuple in PL_A as <*C*, 3>. Process *B* now places the second token request. Here PL_B = <*B*, 4>. Process *B* sends the request to *A*. Root process *A* finds that process *B* has a higher priority and hence places <*B*, 4> ahead of <*C*, 3> in PL_A. However, as introduced in rule 1 earlier in this section, the priority of process *C* would increase by 1, i.e., the 2-tuple modifies as <*C*, 4>. According to priority, process *A* arranges the 2-tuples <*C*, 4> and <*B*, 4> into PL_A. The 2-tuple for *C* is placed ahead of that for *B* as the updated priority of *C* and that of last requesting process *B* are same while the request from *C* is older than that of *B*. The PL_A becomes {<*C*, 4>, <*B*, 4>}.

In order to improve our solution to ensure liveness and also to minimize the data structure maintained in the processes, the dynamic update of the process priority is done only on the request list maintained in the token-holding process.

2.2 Token-Based ME Algorithms

A token is an abstract object which represents a privilege to perform some special operation. Only a process possessing a token can perform these operations; other processes cannot. A token can be passed between processes during operation of the system. This way a privilege can be shared among processes. A single token is shared among all processes in the system and the process which holds it has the exclusive right of executing the critical section (CS). The token represents the privilege to enter a CS. Since single token exists in the system safety of the algorithm is obvious. A process wishing to enter a CS must obtain the token and the token is eventually transferred to the requesting process. The token-based approaches overcome this problem and offer solutions at a lower communication cost, faster than the non-token-based algorithms. But they are deadlock prone and their resilience to failures is poor [8]. If the process holding the token fails, complex token regeneration protocols have to be executed, and when this process recovers, it has to undergo a recovery phase during which it is informed that the token it was holding prior to its failure has been invalidated. Fair scheduling of token among competing processes, detecting the loss of token and regenerating a unique token are some of the major design issues of the token-based ME algorithm.

There exist different ME algorithms for message passing distributed systems [1, 5, 9–13]. Some of these algorithms have been fine-tuned to suit the needs of real time systems in [2, 8, 14, 15].One of the earlier algorithms for group mutual exclusion (GME) [9, 16–18] was given by Joung in [17]. Later, he proposed two algorithms RA1 and RA2 based on Ricart–Agrawala algorithm [14] to ensure ME for message passing systems [11]. There are several token-based algorithms for ME in distributed systems [19–23]. Mittal–Mohan algorithm [9] considers the concept of priority. In Mittal–Mohan algorithm a requesting process cannot assign priority to a request. Raymond has designed an efficient token-based ME algorithm. Raymond's algorithm [8] assumes an inverted tree topology. One major limitation of Raymond's algorithm is the lack of fairness in the sense that a token request that is generated at a later instance may be granted ahead of a previous request.

A DME algorithm is static if it does not remember the history of CS execution. In static algorithms, nodes need not keep information about the current state of the system. In Raymond's [12] token-based mutual exclusion algorithm, requests are set over a static spanning tree of the network, toward the token holder. This logical topology is similar to that of Housn and Trehel's approach [5].

In dynamic algorithms, processes keep track of the current state of the system and the algorithm has a built-in mechanism to make a decision on the basis of this knowledge. Using dynamic approach, a number of solutions have been proposed for prioritized mutual exclusion in a distributed system. Some of these approaches impose a higher message passing overhead.

In broadcast-based algorithms, no such structure is assumed and the requesting process sends message to other processes in parallel, i.e., the message is broadcasted. Suzuki and Kasami proposed a broadcast algorithm [11], in which each process S_i, keeps an array of integers $R_{Ni}[1\ldots N]$, where $R_{Ni}[j]$ is the largest sequence number received so far from S_j. In Suzuki–Kazami algorithm, 0 or N messages are sent per critical section executed and synchronization delay is 0 or T. In Naimi and Thiare [24] implemented the causal ordering in Suzuki–Kasami token-based algorithm in a distributed system. Based on Suzuki–Kazami algorithm, Singhal [21] proposed a heuristically aided algorithm that uses state information to more accurately guess the location of the token. The maximum number of messages required by these algorithms is of the order of the total number of nodes in the system.

An interesting algorithm has been proposed in [25] by extending Naimi–Trehel token-based algorithm [24] that reduces the cost of latency and numbers of messages exchanged between far hosts. The work is on hierarchical proxy-based approach for aggregation of request and permission for token by closed hosts. In [26], another hierarchical token-based algorithm is proposed to solve the GME [Group Mutual Exclusion] problem in cellular wireless networks. They claim that the algorithm is the first GME algorithm for cellular networks. In [27], a rooted tree named "open-cube" is introduced that has noteworthy stability and locality properties, allowing the algorithm to achieve good performances and high tolerance to node failures. In [28], a token-based mutual exclusion algorithm is proposed for arbitrary topologies. This algorithm makes use of the network topology and site information held by each node to achieve an optimal performance. It reduces the delay in accessing the CS by allowing the token to service the requesting site which falls en route its destination. In [29], an algorithm, called "Awareness", is proposed that aims at reducing the maximum response time whereas the number of priority violations remains low. The objective is to both postpone priority increments and prevent low priorities from increasing too fast. However, in this case, the response time of low priorities may considerably increase. In [30], the proposed algorithm can reduce the number of the messages exchanged by 40% when the traffic is light compared with Suzuki and Kasami's algorithm [11]. In [31], a general algorithm is proposed which can represent non-token-based algorithms known as information structure distributed mutual exclusion algorithms (ISDME). The work also introduced a new deadlock-free ISDME algorithm (DF-ISDME) which operates on a restricted class of information structures. DF-ISDME allows deadlock-free solutions for a wider class of information structure topologies than Maekawa algorithms (DF-Maekawa). In [32], an algorithm has been designed which uses a distributed queue strategy and maintains alternative paths at each site to provide a high degree of fault tolerance. However, owing to these alternative paths, the algorithm must use reverse messages to avoid the occurrence of directed cycles, which may form when the directions of edges are reversed after the token passes through. In [33], research is done on a distributed mutual exclusion algorithm based on token exchange and is well suited for mobile ad hoc networks is presented along with a simulation study. A new algorithm is developed for a dynamic logical ring topology. The simulation study on a mobile ad hoc network identifies an effective reduction in the number of hops per application message. This can be achieved by using a specific policy to build the logical ring on-the-fly. In [34], they presented token based mutual exclusion algorithms to handle AND synchronization problems where each process can obtain an exclusive access to a set of resources rather than to a single resource. The message complexity of the algorithms is independent of the number of the requested resources. In [35], another hierarchical token-based algorithm is proposed to solve the GME problem in cellular wireless networks. Arguably, this could be the first GME algorithm for cellular networks. In this algorithm, a resource-starved mobile host requires very little data structure and the bulk of the computation is performed at the resource-rich base station level.

In [36], a simple protocol is proposed that ensures sequential consistency when the shared memory abstraction is supported by the local memories of nodes. It can communicate only by exchanging messages through reliable channels. In [37], a new token-based mutual exclusion algorithm is presented that uses quorum agreements. When a good quorum agreement is used, the overall performance of the algorithm compares favorably with the performance of other mutual exclusion algorithms. In [38], a token-based distributed mutual exclusion algorithm for multithreaded systems is proposed. Several per-node requests could be issued by threads running at each node. Their algorithm relies on special-purpose alien threads running at host processors on behalf of threads running at other processors. The algorithm uses a tree to route requests for the token. The work in [39] is another improvement of Suzuki and Kasami's algorithm [11]. This is rather an extension of the work in [30] that presents a dynamic token-based mutual exclusion algorithm for distributed systems. In this algorithm, a site invoking mutual exclusion sends token request messages to a set of sites possibly holding the token as opposed to all the sites as in the algorithm proposed by Suzuki and Kasami.

2.3 Hierarchical Topology Based ME Algorithms

The processes in the system are assumed to be arranged in a logical configuration like tree, ring, etc., and messages are passed from one process to another along the edges of the logical structure imposed. Token-based algorithms use abstract system models whose control edges form convenient graph topologies. Abstract ring and tree topologies are mostly used. We focus on only tree topology. Raymond's algorithm, MRA-P and FAPP use an abstract inverted tree as the system model. It is called an inverted tree because the control edges point toward the root, rather than away from it. P_{hold} designates the process in possession of the privilege token. Raymond's algorithm uses a spanning tree and the number of message exchanged per CS depends on the topology of the tree. An algorithm, based on a dynamic tree structure, was proposed by Trehal and Naimi [24]. The Naimi–Trehel algorithm takes into consideration a second pointer that is used when a node requests to enter CS before the previous requester leaves it. A major limitation of these two tree-based algorithms is in maintaining this strictly hierarchical logical topology. The algorithm proposed by Bernabeu-Auban and Ahamad in [12] uses path reversal. A path reversal at a node x is performed as the request by node x travels along the path from node x to the root node. Token based mutual exclusion algorithms provide access to the CS through the maintenance of a single token that cannot simultaneously be present at more than one process in the system. Requests for critical section entry are typically directed to whichever process is the current token holder. The token-based solutions for the mutual exclusion problems have an inherent safeness property, as the requesting process must get hold of the token before it enters the critical section. The MRA-P [7] is an improvement over Raymond's algorithm that not only overcomes the fairness problem but also handles the priority of the requesting processes [1]. The work on MRA-P has further motivated others to propose group mutual exclusion algorithms for real time systems [9]. The work, however, used too many control messages across the network. In [40], a service level agreement (SLA) aims to support quality of service (QoS) for services by a cloud provider to its client. Since applications in a cloud share resources, they build on two tree-based distributed mutual exclusion algorithms (Kanrar–Chaki algorithm [41] and Raymond's algorithm [8]) that support the SLA concept. In [42], a good survey work is reported. This analyzes several DME algorithms according to their relative characteristics like Maekawa-type approach [43], hybrid approach etc. In [44], a logical grouping of the sites is done to form a hierarchical structure. This structure is not rigid and can be modified to achieve different performance criteria. The hybrid algorithm needs only a maximum of $2\sqrt{N} + 1$ messages where N is the number of sites in the distributed system.

In [45], the work on GME deals with sharing a set of mutually exclusive resources among all n processes of a network. Three new group mutual exclusion solutions are designed for tree networks. In [46], the work on DME algorithm is for Grids. Two new DME algorithms are proposed based on Naimi–Trehel token-based algorithm [24]. These take into account latency gaps, especially those between local and remote clusters of machines.

2.4 Graph Topology Based ME Algorithms

Solutions for strictly hierarchical topology, e.g., tree, may not be usable for many applications. Besides, if the communication media is wireless and/or nodes are mobile as in MANET, frequent link failures may cause performance problem. Graph-based solution could be more fault-tolerant for such applications and communication media. Raymond's algorithm [8] or MRA-P [7] assumes an inverted tree topology while Walter et al. [47] assume a dynamically re-configurable DAG structure. Raymond's work, MRA-P or its extension in [7] are, however, not quite suitable in a wireless environment, where the topology keeps on changing due to link instability. The solution proposed in [48] works on a dynamically changing DAG structure. Their algorithm handles link failures and new link formation. However, in a constantly changing network environment, absence of cycle in the topology may not be guaranteed. Hence, a DAG based solution may not be ideal for the highly dynamic environment. Moreover, Walter's work does not ensure fairness. In [49], a link failure resilient algorithm has been introduced for mutual exclusion on directed graph topology.

Token based algorithms for ME are proposed for the distributed environment assuming inverted tree topology [7, 8]. However, such a stable, hierarchical topology is quite unrealistic for mobile networks where link failures [10, 47, 49–54] are frequent due to node mobility. In this book, new ME algorithms have been introduced on graph topology. In [55], a new fault-tolerant distributed mutual exclusion algorithm is proposed which can tolerate an arbitrary number of node failure and the message complexity of the algorithm is relatively very low. They claimed that the algorithm functions properly when any of the cooperating nodes in the system fails. In [56], another algorithm is proposed to achieve mutual exclusion in distributed systems. This tolerates up to $t - 1$ arbitrary node failures without executing any recovery procedure and requires $O(\sqrt{tn})$ messages in a network of n nodes. In [57], a dynamic triangular mesh protocol is proposed for mutual exclusion. Here the protocol is fault-tolerant up to $(k - 1)$ site failures and communication failures in the worst case, even when such failures lead to network partitioning. In [58], a mutual exclusion mechanism is developed to ensure that isolated groups do not concurrently perform conflicting operations, when a network is partitioned. The work formalizes these mechanisms in three basic scenarios: where there is a single conflicting type of action; where there are two conflicting types, but operations of the same type do not conflict; and where there are two conflicting types, but operations of one type do not conflict among themselves.

2.5 Permission-Based ME Algorithms

In this type of ME algorithms, a process P_k can enter in its critical section only after it obtains permission(s) from other processes in the system that share the same common resource in the respective critical sections. It requires rounds of message among the processes to obtain the permission to execute CS. In permission-based algorithm or symmetric algorithm, a process wishing to enter a critical section (CS) needs to consult other processes. Processes enter CS in FIFO order. Generally, priority decisions are made using time stamps. A process wishing to enter CS sends time-stamped request messages to all other processes and waits till it received a "go ahead" reply from each of them. A process receiving a request message immediately sends a "go ahead" reply if it is not interested in using the CS at present or if the received request precedes its own request in time. Otherwise, it delays its relays its reply until it has itself used the CS. A number of solutions in both the approaches have been studied. Permission-based algorithms tend to increase the network traffic significantly.

Every process has a logical clock and all request messages are time-stamped with the current value of the sender's logical clock. Before sending a message, a process increases its logical clock and, upon reception of a message, a process updates it with the maximum between the message time-stamp value and its current value. A total ordering of request messages can thus be established based on the value of their time stamps and, if necessary, the identity of the sender process in order to break ties. In other words, a request message whose time stamp is lower than a second one has priority over it. If the time stamps are equal, the request message sent by the process with the lowest identifier has priority. In both cases, the request message with higher priority has precedence over the other one. When a process receives a request, it answers either if it is not interested in using the resource (i.e., if the process is in the idle state (I)) or if it is interested, but the received request message has precedence over its own; otherwise it answers to all requests it had received upon executing the exit protocol (exiting the critical section). At that moment, the process also changes its state to idle.

Permission-based mutual exclusion algorithms exchange messages to determine which process can access the CS next. As for example, Lamports' algorithm [5] is a symmetric, permission-based algorithm that requires $3 * (N - 1)$ messages to provide mutual exclusion. The permission-based algorithm proposed by Ricart and Agrawala, reduced the number of required messages to $2 * (N - 1)$ messages per critical section entry [14]. In another work, Carvalho and Roucairol [13] were able to reduce the number of messages to be between 0 and $2 * (N - 1)$ and Sanders [22] gave a theory of information structure to design mutual exclusion algorithms. An information structure describes which processes maintain information about what other processes, and from which processes a process must request information before entering the CS. Singhals' heuristic algorithm [23] guarantees some degree of fairness but is not completely fair, as a lower priority request can execute CS before a higher priority request if the higher priority request is delayed. The algorithm has defined different criteria for fairness. Mueller [2] introduced a prioritized token-based mutual exclusion algorithm. In many of the solutions, the fairness criterion is considered to be equivalent to the priority, i.e., a job that arrives early is defined as a higher priority job [24]. In [50], Sil and Das introduced a new energy efficient mutual exclusion algorithm for a mobile ad hoc network. The whole network is hierarchically clustered to get a logical tree structure of the network. They addressed the issue of mobility of nodes extensively. In [3], Saxena and Rai present a survey of various permission-based distributed mutual exclusion algorithms and their comparative performance. Permission-based algorithms can be further subdivided into coterie-based algorithms and voting-based algorithms.

Acoterie is a nonempty set with a collection of elements satisfying two conditions: the minimal condition and intersection property. The elements of a coterie form overlapping quorum groups or quorum sets or just quorums. If a process wants to perform an operation in the CS, it must obtain permission from each and every process of the quorum to which the process belongs. The candidate process may belong to two or more quorums. It has to get permission from each member of the overlapping quorums. Since a process grants permission to only one process at a time and since any two quorums in a coterie have at least one process in common, mutual exclusion, i.e., safety property is guaranteed [59]. In [60], the primary aim is to investigate the use of a gossip protocol to an ME algorithm to cope with scalability and fault-tolerant problems in cloud computing systems. They present a gossip-based mutual exclusion algorithm for cloud computing systems with dynamic membership behavior. In [61], an algorithm called prioritizable adaptive distributed mutual exclusion (PADME) is proposed to meet the need for differentiated services between applications for file systems and other shared resources in a public cloud. In [62], a hybrid distributed mutual exclusion algorithm is designed that uses Singhal's dynamic information structure algorithm [21] as the local algorithm to minimize time delay and Maekawa's algorithm [43] as the global algorithm to minimize message traffic. In [63], the author presents an N-process local-spin mutual exclusion algorithm, based on non-atomic reads and writes. Each process performs $\Theta(\log N)$ remote memory references to enter and exit its critical section. In [64], authors present a distributed algorithm for solving the GME problem based on the notion of surrogate-quorum. Intuitively, they use the quorum that has been successfully locked by a request as a surrogate to service other compatible requests for the same type of critical section. In [65], an interesting algorithm based on GME is for the philosophers to ensure that a philosopher attempting to attend a forum will eventually succeed, while at the same time encourage philosophers interested in the same forum to be in the meeting room simultaneously. Another approach, based on the work in [64] offers a high degree of concurrency of n, where n is the number of processes in the system [66]. The algorithm can adapt without performance penalties to dynamic changes in the number of groups. In [67], an FCFS group mutual exclusion algorithm is introduced that uses only $\Theta(N)$ bounded shared registers. They also present a reduction that transforms any "abortable" FCFS mutual exclusion algorithm M into a GME algorithm G. Thus, different group mutual exclusion algorithms can be obtained by instantiating M with different abortable FCFS mutual exclusion algorithms. In [68], Vidyasankar introduced a situation where philosophers with the same interest can attend a forum in a meeting room, and up to k meeting rooms are available. In [69], a class of Maekawa-type mutual exclusion algorithms [43] is presented that is free from deadlocks and do not exchange additional messages to resolve deadlocks.

In [70], a new quorum generation algorithm has been presented. A symmetric quorum can be generated based on this algorithm and the size of the quorum is about $2\sqrt{N}$. This distributed mutual exclusion algorithm has reduced the message complexity of Maekawa-type distributed mutual exclusion algorithm m messages, m is the quorum size. In [71], an analysis is done on the shared memory requirements for implementing ME of N asynchronous parallel processes in a model where the only primitive communication mechanism is a general test-and-set operation on a single shared variable. In [72], the concept of coterie is extended to k-coterie for k entries to a critical section. A structure named Cohorts is proposed to construct quorums in a k-coterie. The solution is resilient to node failures and/or network partitioning and has a low communication cost. In [73], a surficial quorum system for GME is presented. The surficial quorum system is geometrically evident and so it is easy to construct. It also has a nice structure based on which a truly distributed algorithm for GME can be obtained, and loads of processes can be minimized.

2.6 Voting-Based ME Algorithms

In voting-based algorithm, a decision is taken based on voting. Unlike coterie-based algorithms, each and every process in a system of n processes need not wait for permission in terms of votes from each of the remaining $n - 1$ processes in the system. Hence, the message complexity of voting-based ME algorithms is lower than the symmetric ME algorithms [74].In voting-based algorithms, each candidate process seeking entry into CS sends request messages for votes to other processes. The votes are counted on receipt of the reply messages (votes) from various nodes. If the number of votes obtained is more than or equal to majority then the corresponding candidate process is granted access to enter its critical section. The voting schemes [75, 76] are often far from being correct as it may suffer from lack of liveness. Say, a network is partitioned and only 85% of the $n - 1$ processes are connected. If there are two requesting processes, it may lead to a scenario where both the requesting processes get more than 40% of the votes. However, none of these would achieve majority consensus. A similar situation may occur when there are more than two requesting processes even if the network is not partitioned and all 100% of the votes are cast.

Many simple majority voting schema [59, 76–83] are far from being correct as they may suffer from lack of liveness. Jajodia and Mutchler [84] proposed two generalizations of voting schemes: dynamic voting and dynamic-linear voting (i.e., dynamic voting with linear ordered copies). In dynamic voting, the number of sites necessary for carrying out an update is a function of the number of up-to-date copies at the time of the update. In case of static algorithms, this depends on the total number of copies. Moreover, dynamic-linear voting accepts an update whenever dynamic voting does.

Rabinovich and Lazowska [78] proposed a mechanism that allows dynamic adjustment of quorum sets when quorums are defined based on a logical network structure. This improves the availability of the replica control protocols. The basic principle is to devise a rule that unambiguously imposes a desired logical structure on a given ordered set of processes. In this protocol, each node is assigned a name and all names are linearly ordered. Among all the nodes replicating a data item, a set of nodes is identified as the current epoch. At any instance, the data item may have only one current epoch associated with it.

Adam [79] also proposed a dynamic voting consistency algorithm, which is effective in environments where the majority of user requests are "read" types of requests. This algorithm also works on the majority partition would be available even if changes in the network topology take place at a higher rate than the update rate, as long as only simple partitioning takes place.

In [81], works on general voting protocol are discussed that reduces the vulnerability of the voting process to both attacks and faults. This algorithm is contrasted with the traditional two-phase commit protocols. The algorithm is applicable to exact and inexact voting in networks where atomic broadcast and predetermined message delays are present.

In [82], two dynamic voting algorithms are proposed. The "optimistic dynamic voting", operates on possibly out-of-date information. On the other hand, "topological dynamic voting" works on the topology of the network on which the copies reside to increase the availability of the replicated data.

Paris and Long [83] proposed a dynamic voting protocol that does not need the instantaneous state information required by other dynamic voting algorithms. This algorithm also served low cost in traffic similar to that of static majority consensus voting.

In the existing literature as reviewed, no work is found that handles the situation when none of the candidate processes achieve majority consensus. The progress condition demands that at least one of the competing processes must be selected for CS execution even when majority consensus is not achieved by any process. On the other hand, the liveness property demands that all competing processes must eventually be allowed to enter the CS. Our work aims to address this gap in the existing literature and proposes a new voting-based algorithm that ensures both progress condition and safety even when no single process wins majority vote. An innovative solution introduced later in this text is also compatible with any voting mechanism that ensures liveness.

In static voting algorithm, the processes do not keep information about the current state of the system and the votes, once assigned, are not changed as the algorithm evolves [75]. Some of the earliest voting-based distributed mutual exclusion algorithms were given by Thomas [75] and Gifford [56]. This was static in nature in which the votes were fixed a priori and the distributed system is assumed to be fully connected by message passing. A process requesting to enter the CS must obtain permission from majority of processes in distributed system; otherwise, it must not enter the CS and wait until it is allowed to enter the critical section. Xu and Bruck [80] proposed a deterministic majority voting algorithm for NMR (*N* Modular Redundant) systems where *N* computational modules execute identical tasks and they need to periodically vote on their current states. Agrawal and Abbadi [85] impose a logical structure on the set of copies of an object and develop a protocol that uses the information available in the logical structure to reduce communication requirements for read and write operations and call it "Tree Quorum Protocol". This algorithm may be considered as a generalization of the static voting-based algorithms. However, when a distributed system is partitioned as a result of node or link failure, a static voting algorithm cannot adapt to the changing topology toward maintaining system availability. Thus, in order to avoid a hang-parliament status, it is necessary to change the paradigm to dynamic voting.

In dynamic voting algorithm, the processes keep track of the current state of the system and in the case of network partitioning, due to link or process failures, new votes may be assigned so as to make at least one partitioned group of processes active and to keep the system working [86, 87]. In this category, a distinguished partition may still have the majority of votes after next partitioning. In order to avoid data inconsistency, dynamic vote reassignment is only possible inside the distinguished partition and the votes of other partitions remain unchanged. It is due to the fact that partitions are unaware of each other. Group consensus and autonomous reassignment are two dynamic vote assignment techniques presented in [77]. In the first approach a process that firstly discovered partitioning, initiates itself as a candidate and establishes an election among the processes in its region. Then, if the majority of processes agreed on its coordination, it would announce itself as new coordinator and install a new vote reassignment such that only processes in majority partition can commit a request. Henceforth, all votes of processes will modify to new votes and mutual exclusion will be achieved in at most one partition. In the second approach, there is no centralized coordinator and each process is responsible for assigning its vote. In order to avoid each process to arbitrarily change its vote and probably compromise mutual exclusion, an approval of a certain number of processes must be obtained formerly. Autonomous reassignment is similar to group consensus except that each node is responsible for deriving its own vote. In symmetric algorithm a process wishing to enter a CS sends time-stamped request messages to all other processes and waits till it received a grant messages from each of them. The process executes CS whose time stamp is lowest like Ricart–Agrawala [14]. Singh and Bandyopadhyay [42] presents a work on mutual exclusion that ensures liveness, and correctness properties. Timed-buffer distributed voting algorithm (TB-DVA) [74] is a secure distributed voting protocol. It is unique for fault tolerance and security compared to several other distributed voting schemes. TB-DVA is a radical approach to distributed voting because it reversed the two-phase commit protocol: a commit phase (to a timed buffer) is followed by a voting phase. This conceptually simple change greatly enhances security by forcing an attacker to compromise a majority of the voters in order to corrupt the system. Recall that in the two-phase commit protocol only one voter must be compromised to corrupt a system. In [88], a vote assignment algorithm is introduced toward maximizing the reliability. In this approach, the voting weight assigned to each node is readily determined if the link failure rate is negligible. Majority voting is commonly used in distributed computing systems to control mutual exclusion, and in fault-tolerant computing to achieve reliability.

2.7 Conclusions

Ensuring mutual exclusion among the processes is an important aspect. The existing algorithms follow either a permission-based or a token-based approach. Studies on a number of solutions in both the approaches reveal that the permission-based algorithms tend to increase the network traffic significantly. The token-based approaches perform better from this perspective and offer solutions at a lower communication cost.

References

1. Housini, A., Trehel, M.: Distributed mutual exclusion token-permission based by prioritized groups. In: Proceedings of ACS/IEEE International Conference, pp. 253–259 (2001)
2. Mueller, F.: Prioritized token-based mutual exclusion for distributed systems. In: Proceedings of the 9th Symposium on Parallel and Distributed Processing, pp. 791–795 (1998)
3. Saxena, P.C., Rai, J.: A survey of permission-based distributed mutual exclusion algorithms. Comput. Stan. Interfaces **25**(2), 159–181 (2003)
4. Helary, J.M., Mostefaoui, A., Raynal, M.: A general scheme for token and tree based distributed mutual exclusion algorithm. IEEE Trans. Parallel Distrib. Syst. **5**(11), 1185–1196 (1994)
5. Lamport, L.: Time, clocks, and the ordering of events in a distributed system. Commun. ACM **21**(7), 558–565 (1978)
6. Madria, S.K.: Timestamp based approach for the detection and resolution of mutual conflicts in real-time distributed systems. Computer Science Technical Reports. Paper 1367, pp. 1–16 (1997)
7. Karnar, S., Chaki, N.: Modified Raymond's algorithm for priority (MRA-P) based mutual exclusion in distributed systems. In: Proceedings of ICDCIT 2006. LNCS 4317, pp. 325–332 (2006)
8. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. **7**, 61–77 (1989)
9. Mittal, N., Mohan, P.K.: A priority-based distributed group mutual exclusion algorithm when group access is non-uniform. J. Parallel Distrib. Comput. **67**(7), 797–815 (2007)
10. Walter, J.E., Welch, J.L., Vaidya, M.H.: Mutual exclusion algorithm for Ad hoc mobile networks. Wirel. Network **7**(6), 585–600 (2001)
11. Suzuki., Kasami, T.: An optimality theory for mutual exclusion algorithms in computer science. In: Proceedings of IEEE International Conference on Distributed Computing and System, pp. 365–370 (1982)
12. Bernabeu-Auban, J.M., Ahamad, M.: Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm, vol. 392, pp. 33–44 (1989)

13. Carvalho, O.S.F., Roucairol, G.: On mutual exclusion in computer network. Commun. ACM **26**(2), 146–147 (1983)
14. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Commun. ACM **24**(1), 9–17 (1981)
15. Lodha, S., Kshemkalyani, A.: A fair distributed mutual exclusion algorithm. IEEE Trans. Parallel Distrib. Syst. **11**(6), 537–549 (2000)
16. Swaroop, A., Singh, A.K.: A Distributed group mutual exclusion algorithm for soft real-time systems. In: Proceedings of World Academy of Science, Engineering and Technology, vol. 26, pp. 138–143 (2007)
17. Joung, Y.J.: Asynchronous group mutual exclusion. In: Proceedings of the 17th annual ACM Symposium on Principles of Distributed Computing (PODC), pp. 51–60 (1998)
18. Manabe, Y., Park, J.: A quorum based extended group mutual exclusion algorithm without unnecessary blocking. In: Proceedings of 10th International Conference on Parallel and Distributed Systems (ICPADS'04) (2004)
19. Attreya, R., Mittal, N.: A dynamic group mutual exclusion algorithm using surrogate quorums. In: Proceedings of the 25th IEEE Conference on Distributed Computing Systems (ICDCS'05) (2005)
20. Lin, D., Moh, T.S., Moh, M.: Brief announcement: Improved asynchronous group mutual exclusion in token passing networks. In: Proceedings of PODC'05, pp. 275–275 (2005)
21. Singhal, M.: A heuristically-aided algorithm for mutual exclusion for distributed systems. IEEE Trans. Comput. **38**(5), 70–78 (1989)
22. Sanders, B.: The information structure of distributed mutual exclusion algorithm. ACM Comput. Syst. **5**(3), 284–299 (1987)
23. Singhal, M.: A dynamic information structure mutual exclusion in distributed system. IEEE Trans. Parallel. Distrib. Syst. **3**(1), 121–125 (1992)
24. Naimi, M., Trehel, M., Arnold, A.: A log(N) distributed mutual exclusion algorithm based on path reversal. J. Parallel. Distrib. Comput. **34**(1), 1–13 (1996)
25. Bertier, M., Arantes, L., Sens, P.: Hierarchical token based mutual exclusion algorithms. In: IEEE International Symposium on Cluster Computing and the Grid, pp. 539–546 (2004)
26. Swaroop, A., Singh, A.K.: A token-based group mutual exclusion algorithm for cellular wireless networks, In: India Conference (INDICON), pp. 1–4 (2009)
27. Helary J.M., Mostefaoui A.: A O($\log_2$ n) fault-tolerant distributed mutual exclusion algorithm based on open-cube structure. In: 14th International Conference in Distributed System, pp. 89–96 (1994)
28. Saxena, P.C., Guptal, S.: A token-based delay optimal algorithm for mutual exclusion in distributed systems. Comput. Stan. Interfaces **21**(1), 33–50 (1999)
29. Lejeune, J., Arantes, L., Sopena, J.: A O($\log_2$ n) fault-tolerant distributed mutual exclusion algorithm based on open-cube structure. In: 42nd International Conference of Parallel Processing (ICPP), pp. 290–299 (2013)
30. Chang, YI.: A dynamic request set based algorithm for mutual exclusion in distributed systems. Operating Syst. Rev. **30**(2), 52–62 (1996)
31. Bonollo, U., Sonenberg, E.A.: Deadlock-free information structure distributed mutual exclusion algorithms. Aust. Comput. Sci. Commun. **18**, 234–241 (1996)
32. Chang, Y.I., Singhal, M., Liu.,M.: A fault tolerant algorithm for distributed mutual exclusion. In: Ninth IEEE Symposium on Reliable Distributed Systems, pp. 146–154 (1990)
33. Baldoni, R., Virgillito, A., Petrassi, R.: A distributed mutual exclusion algorithm for mobile Ad-hoc networks. In: Seventh International Symposium on Computers and Communications (ISCC 2002), pp. 539–544 (2002)
34. Maddi, A.: Token based solutions to M resources allocation problem. In: ACM Symposium on Applied Computing, pp. 340–344 (1997)
35. Swaroop, A., Singh, A.K.: A token-based group mutual exclusion algorithm for cellular wireless networks. In: India Conference (INDICON-2009), pp. 1–4 (2009)

36. Raynal, M.: Token-based sequential consistency in asynchronous distributed systems. In: 17th International Conference on Advanced Information Networking and Applications (AINA 2003), pp. 421–426 (2003)
37. Mizuno, M., Neilsen, M.L., Rao, R.: A token based distributed mutual exclusion algorithm based on quorum agreements. In: ICDCS, pp. 361-368 (1991)
38. Meza, F., Perez, J., Eterovic, Y.: Implementing distributed mutual exclusion on multithreaded environments: The alien-threads approach. In: Advanced Distributed Systems, pp. 51–62 (2005)
39. Chang, Y.I., Singhal, M., Liu, M.T.: A dynamic token-based distributed mutual exclusion algorithm. In: Tenth Annual International Phoenix Conference on Computers and Communications, pp. 240–246 (1991)
40. Lejeune, J., Arantes, L., Sopena, J., Sens, P.: Service level agreement for distributed mutual exclusion in cloud computing. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012), pp. 180–187 (2012)
41. Kanrar, S., Chaki, N.: FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems. Special issue on recent advances in network and parallel computing. Int. J. Networks 5(1), 11–18 (2010). ISSN: 1796-2056
42. Chang, Y.I.: A simulation study on distributed mutual exclusion. J. Parallel Distrib. Comput. **33**(2), 107–121 (1996)
43. Maekawa, M.: A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3**(2), 145–159 (1985)
44. Madhuram, S., Kumar, A.: A hybrid approach for mutual exclusion in distributed computing systems. In: Sixth IEEE Symposium on Parallel and Distributed Processing, pp. 18–25 (1994)
45. Beauquier, J., Cantarell, S., Datta, A.K., Petit, F.: Group mutual exclusion in tree networks. In: Ninth International Conference on Parallel and Distributed Systems, pp. 111–116 (2002)
46. Bertier, M., Arantes, L., Sens, P.: Distributed mutual exclusion algorithms for grid applications: A hierarchical approach. J. Parallel. Distrib. Comput. **66**(1), 128–144 (2006)
47. Walter, J., Cao, G., Mohanty, M.: A k-mutual exclusion algorithm for ad hoc wireless networks. In: Proceedings of the First Annual Work Shop on Principles of Mobile Computing (POMC 2001) (2001)
48. Kanrar, S., Choudhury, S., Chaki, N.: A link-failure resilient token based mutual exclusion algorithm for directed graph topology. In: Proceedings of the 7th International Symposium on Parallel and Distributed Computing—ISPDC 2008 (2008)
49. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new link failure resilient priority based fair mutual exclusion algorithm for distributed systems. J. Network. Syst. Manage. (JONS) **21**(1), 1–24 (2013). ISSN 1064-7570
50. Sil, S., Das, S.: An energy efficient algorithm for distributed mutual exclusion in mobile Ad-hoc networks. World. Acad. Sci. Eng. Technol. **64**, 517–522 (2010)
51. Panghal, K., Rana, M.K., Kumar, P.: Minimum-process synchronous check pointing in mobile distributed systems. Int. J. Comput. Appl. **17**(4), 1–4 (2011)
52. Chandra, T.D., Hadzilacos, V., Toueg, S.: The weakest failure detector for solving consensus. J. ACM **43**(4), 685–722 (1996)
53. Chen, W., Lin, S., Lian, Q., Zhang, Z.: Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. Microsoft Research Technical Report MSR-TR-2005-58 (2005)
54. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Technical Report in Computer and Communication Sciences, id: 200227 (2002)
55. Reddy, R.L.N., Gupta, B., Srimani, P.K.: A new fault tolerant distributed mutual exclusion algorithm. In: ACM/SIGAPP Symposium on Applied computing: Technological Challenges of the 1990's, pp. 831–839 (1992)
56. Bouabdallah, A., Koenig, J.C.: An improvement of Maekawa's mutual exclusion algorithm to make it fault-tolerant. Parallel Process. Let. **02n03**(2), 283–290 (1992)

57. Chang, Y.I., Chang, Y.J.: A fault-tolerant dynamic triangular mesh protocol for distributed mutual exclusion. ACM SIGOPS Operating Syst. Rev. **31**(3), 29–44 (1997)
58. Barbara, D., Garcia-Molina, H.: Mutual exclusion in partitioned distributed systems. Distrib. Comput. **1**(2), 119–132 (1986)
59. Lotem, E.Y., Keidar, I., Dolev, D.: Dynamic voting for consistent primary components. In: 16th ACM Symposium on Principles of Distributed Computing (PODC), pp. 63–71 (1997)
60. Lim, J.B., Chung, K.S., Chin, S.H., Yu, H.C.: A gossip-based mutual exclusion algorithm for cloud environments. In: Advances in Grid and Pervasive Computing. Lecture Notes in Computer Science, vol. 7296, pp. 31–45 (2012)
61. Edmondson, J., Schmidt, D., Gokhale, A.: QoS-enabled distributed mutual exclusion in public clouds. In: On the Move to Meaningful Internet Systems: OTM 2011. Lecture Notes in Computer Science, vol. 7045, pp. 542–559 (2011)
62. Chang, Y.I.: A hybrid distributed mutual exclusion algorithm. Microproces. Microprogram. **41**(10), 715–731 (1996)
63. Vidyasankar, K.: A highly concurrent group mutual L-execution algorithm. Parallel Process. Let. **16**(04), 467–483 (2006)
64. Atreya, R., Mittal, N., Peri, S.: A quorum-based group mutual exclusion algorithm for a distributed system with dynamic group set. IEEE Trans. Parallel Distrib. Syst. **18**(10), 1345–1360 (2007)
65. Vidyasankar, K.: A simple group mutual l-exclusion algorithm. Inf. Process. Let. **85**(2), 79–85 (2003)
66. Atreya, R., Mittal, N.: A distributed group mutual exclusion algorithm using surrogate-quorums. In: 25th IEEE International Conference on Distributed Computing Systems, pp. 251–260 (2005)
67. Jayanti, P., Petrovic, S., Tan, K.: Fair group mutual exclusion. In: Twenty-Second Annual Symposium on Principles of Distributed Computing, pp. 275–284 (2003)
68. Takamura, M., Altman, T., Igarashi, Y.: Speedup of Vidyasankar's algorithm for the group k-exclusion problem. Inf. Process. Let. **91**(2), 85–91 (2004)
69. Singhal, M.: A class of deadlock-free Maekawa-type algorithms for mutual exclusion in distributed systems. Distrib. Comput. **4**(3), 131–138 (1991)
70. Li, M.A., Liu, X.S., Wang, Z.: A high performance distributed mutual exclusion algorithm based on relaxed cyclic difference set. Dianzi Xuebao (Acta Electron. Sinica) **35**(1), 58–63 (2007)
71. Burns, J.E., Jackson, P., Lynch, N.A., Fischer, M.J., Peterson, G.L.: Data requirements for implementation of n-process mutual exclusion using a single shared variable. J. ACM (JACM) **29**(1), 183–205 (1982)
72. Huang, S.T., Jiang, J.R., Kuo, Y.C.: K-coteries for fault-tolerant k entries to a critical section. In: ICDCS, pp. 74–81 (1993)
73. Joung, Y.J.: Quorum-based algorithms for group mutual exclusion. In: Distributed Computing, pp. 16–32 (2001)
74. Hardekopf, B., Kwiat, K., Upadhyaya, S.: A decentralized voting algorithm for increasing dependability in distributed systems. Join MEETING of the 5th World Multiconference on Systemics, Cybernetics and Informatics (SCI 2001) and the 7th International Conference on Information System Analysis and Synthesis (ISAS 2001) (2001)
75. Thomas, T.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans.Database Syst. **4**(2), 180–209 (1979)
76. Ahmad, M., Ammar, M.H., Cheung, S.Y.: Multi-dimensional voting: A general method for implementing synchronisation in distributed systems. In: Proceedings 10th International Conference on Distributed Computer Systems, pp. 362–369 (1990)
77. Barbara, D., Garcia-Molina, H., Spauster, A.: Increasing availability under mutual exclusion constraints with dynamic vote reassignment. ACM Trans. Comput. Syst. **7**(4), 394–428 (1989)

78. Rabinowich, M., Lazowska, E.D.: Improving fault-tolerance and supporting partial writes in structured coterie protocols for replicated objects. In: Proceedings ACM SIGMOD, pp. 226–235 (1992)
79. Adam, N.R.: New dynamic voting algorithm for distributed database systems. IEEE Trans. Knowl. Data Eng. Arch. **6**(3), 470–478 (1994)
80. Xu, L., Bruck, J.: Deterministic voting in distributed systems using error-correcting codes. IEEE Trans. Parallel Distrib. Syst. **9**(8), 813–824 (1998)
81. Hardekopf, B., Kwiat, K., Upadhyaya, S.: Secure and fault-tolerant voting in distributed systems. In: IEEE Aerospace Conference (2001)
82. Paris, J.F., Long, D.D.E.: Efficient dynamic voting algorithms. In: Proceedings of the Fourth International Conference on Data Engineering, pp. 268–275 (1998)
83. Paris, J.F., Long, D.D.E.: A realistic evaluation of optimistic dynamic voting. In: Proceeding of Seventh Symposium on Reliable Distributed Systems, pp. 129–137 (1988)
84. Jajodia, S., Mutchler, D.: Dynamic voting algorithms for maintaining the consistency of a replicated data. ACM Trans. Database Syst. **15**(2), 230–280 (1990)
85. Agrawal, D., El Abbadi, A.: An efficient and fault tolerant solution for distributed mutual exclusion. ACM Trans. Comput. Syst. **9**(1), 1–20 (1991)
86. Zarafshan, F., Karimi, A., Al-Haddad, S.A.R., Saripan, M.I., Subramaniam, S.: A preliminary study on ancestral voting algorithm for availability improvement of mutual exclusion in partitioned distributed systems. In: Proceedings of International Conference on Computers and Computing (ICCC'11), pp. 61–69 (2011)
87. Garcia-Molina, H., Barbara, D.: How to assign votes in a distributed system. J. Assoc. Comput. Mach. **32**(4), 841–860 (1985)
88. Tong, Z., Kain, R.Y.: Vote assignments in weighted voting mechanisms. In: Seventh Symposium on Reliable Distributed Systems, pp. 138–143 (1988)

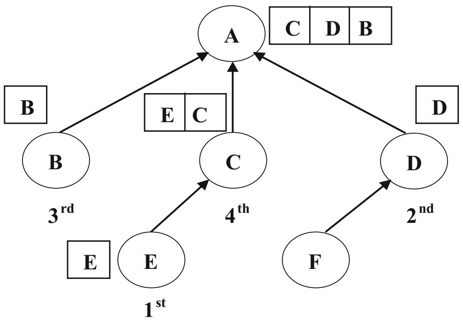
Chapter 3
Tree-Based Mutual Exclusions

In token-based approach of algorithms to solve distributed ME problems [1–4], a tree-based logical topology has been assumed by many researchers [5–8]. Raymond's algorithm [9] is one such widely used distributed token-based ME algorithm [10–12]. In Raymond's algorithm, there is only one token and only a process receiving the token has the right to enter CS. Token requests are sent over a static spanning tree toward the root node holding the token.

One of the concerns with ME algorithms is fairness. This has been illustrated with an example in Fig. 3.1 for Raymond's algorithm. Initially, process A is the holder of the token, i.e., root process. We assume that each process name is considered as process id. The processes *E*, *D*, *B*, and *C* place the first, second, third, and fourth requests to enter the respective critical sections. Figure 3.1 illustrates contents of the local queues after the four requests are placed. Let us assume that process *A* exits from its CS now. According to Raymond's algorithm, process *C* is now the root of the tree as *A* passes the token to process *C* along with a request to return the control since its local queue is not empty. Process *C*, further, sends a request to process *E*. The local queue at process *E* will add the process id of process *C*. Process *C* passes the token to process *E* and also sends a token request to process *E*. Process *E* removes the id *E* from its local queue and enters the CS leaving only *C* in the queue.

After process *E* completes its CS, it removes *C* from its local queue, passes the token back to process *C*, and reverses the edge (*E*, *C*). Once again, process *C* becomes the root. Now, process *C* removes *C* from its local queue and process *C* completes its CS. Thus, process *C* enters the CS ahead of process *D* even though process *D*'s request for the token happened earlier then process *C*. Continuing with the example, we find that process A too is allowed to enter CS after violating fairness as explained above. The Modified Raymond's Algorithm (MRA) [13], as described below aims to solve this problem.

Fig. 3.1 Fairness in a Tree-based topology



3.1 Modified Raymond's Algorithm (MRA)

The proposed algorithm, like Raymond's algorithm [9] assumes that nodes are connected using a logical topology of inverted tree, with the node holding the token P_{hold} at the root of the tree. The proposed algorithm, to be referred as Modified Raymond's Algorithm (MRA), keeps the fundamentals of the Raymond's algorithm as little altered as possible. The fairness problem, however, has been taken care of by introducing a simple queue structure, request queue, maintained at the root of the tree. This additional storage requirement is invariant of the size of the input, i.e., to the number of requests for the token. Thus, the proposed MRA solution solves the fairness issue at no additional cost Fig. 3.2.

3.1.1 Data Structure and Algorithm for MRA Algorithm

Description for MRA Algorithm

In MRA, a Process P_x that wants to enter CS adds its id x in the LDQ$_x$. Next, P_x sends a request for the token TKN on its out-edge along with LDQ$_y$. When a process $P_y \neq P_{\text{hold}}$ receives a token request from another process P_k then P_y adds the ids from LDQ$_k$ into LDQ$_y$ followed by its own id y. LDQ$_y$ is sent out with TKN, where the request is originally from P_x. If a request from P_y is pending, then only token request TKN is sent out. When a process $P_y = P_{\text{hold}}$ receives a token request

Local double queue (LDQ): Every process P$_x$ maintains a local double ended queue (LDQ$_x$). A process P$_x$ wishing to enter the CS, enter its own id, *x*, in the LDQ$_x$.

Request queue (RQ): Process P$_{\text{hold}}$ maintains a request queue that stores the process ids for the token requesting processes in an FIFO order. The RQ is transferred to the new root of the tree along with the token.

Fig. 3.2 Data structure of MRA algorithm

TKN from process P_k, then P_y adds the ids from LDQ$_k$ into LDQ$_y$ followed by its own id, i.e., y. The processor id x is added to the RQ.
On completing the execution of a CS, P_y (current P_{hold}), performs the following. P_y scans the first id k from RQ. It extracts entries starting from k to the first occurrence of y from LDQ$_y$. The extracted sequence of ids is reversed from y to k. Edge P_m to P_y is reversed, where m is the id that follows j in the sequence. P_m is the new P_{hold} and root—the token and RQ are passed to P_m from P_y. If LDQ$_y$ is not empty and the last id left in it is y, then P_y places a token request to P_m along with the reduced LDQ$_y$. The ids from LDQ$_y$ are added to LDQ$_m$ followed by m and LDQ$_y$ is emptied. The newly designated process $P_m = P_{\text{hold}}$ performs the following. If the first id of RQ is m, then it is removed from the RQ. The entry m is also removed from the top of LPQ$_m$. Process P_m enters its CS otherwise token goes to another process. A pseudocode representation for the MRA algorithm is given below.

Begin

```
1. Step 1: A Process $P_i$ wishing to enter CS.
2. insert <i> in $LDQ_i$[ ]; /* $P_i$ enters its id i in the $LDQ_i$ */
3. send token_request <i>;
   /* $P_i$ sends a request for the token {i} on its out-edge along with $LDQ_j$ */
4. Step 2: When a process $P_j \neq P_{hold}$ receives a token request from another
   process $P_{k.}$
5. insert <k, j> in $LDQ_j$[ ];
   /* $P_j$ adds the ids from $LDQ_k$ into $LDQ_j$ followed by its own idj */
6. send token_request <j>;
   /* $LDQ_j$, is sent out with {i}, where the request is originally from $P_i$. If a
   request from $P_j$ is already pending, then only token request {i} is sent out */
7. Step 3: When a process $P_j = P_{hold}$ receives a token request {i} from process $P_k$.
8. insert <i, j> in $LDQ_j$[ ];
   /* $P_j$ adds the ids from $LDQ_k$ into $LDQ_j$ followed by its own id, i.e., j */
9. insert <i> in RQ[ ]; /* The processor id. i is added to the RQ */
10. Step 4: On completing the execution of a CS, $P_j = P_{hold}$, performs the
    following:
11. k $\in$ RQ[ ]; /* $P_j$ scans the first id k from RQ */
12. extract <k,., j>;
    /* It extracts entries starting from k to the first occurrence of j from $LDQ_j$ */
13. reverse <j,., k>; /* The extracted sequence of ids is reversed from j to k */
    /* Edge $P_m$ to $P_j$ is reversed, where m is the id that follows j in the sequence */
14. $P_m = P_{hold}$; /* $P_m$ is the new $P_{hold}$ and root—the token and RQ is passed to $P_m$
    from $P_j$ */
15. if ($LDQ_j$[ ] $\neq$ null)
    /* If $LDQ_j$ is not empty and the last id left in it is j, then $P_j$ places a token
    request to $P_m$ along with the reduced $LDQ_j$ */
16. send token_request <j>; /* The ids from $LDQ_j$ are added to $LDQ_m$ followed by
    m */
```

```
17. else
18. LDQj[ ] = null; /* $LDQ_j$ is emptied */
19. Endif.
20. Step 5: The newly designated process $P_m = P_{hold}$ performs the following:
21. remove the first process tuple <m> from RQ;
22. If (the first id of RQ is m)
    /* The first id of RQ, i.e., m, in this case, is removed from the RQ */
23. remove the first process tuple <m> from $LDQ_m$;
    /* The entry m is removed from the top of $LDQ_m$ */
24. $P_m$ enters its CS;
25. else
26. Repeat from Step 4;
27. Endif.
```

End

Illustrating the MRA Algorithm

The inverted tree structure is drawn for the MRA. The processes P_8, P_{11}, and P_6 have placed the requests to enter the critical section, in that order. As per Step 1 of the proposed algorithm, LDQ_8 stores 8 and the first token request is propagated to P_3 along with a copy of LDQ_8 Fig. 3.3.

P_3 on receipt of the request from P_8, puts the content of LDQ_8 followed by its own id., i.e., 3 into LDQ_3. Thus, $LDQ_3 = \{8, 3\}$. P_3, now forwards the token request along with a copy of $LDQ_3 = \{8, 3\}$ on its out-edge. $P_1 = P_{hold}$ now receives the request and puts {8, 3, 1} in LDQ_1. It also puts the first id of the LDQ_1, thus formed, into RQ, i.e., RQ = {8}, in this case.

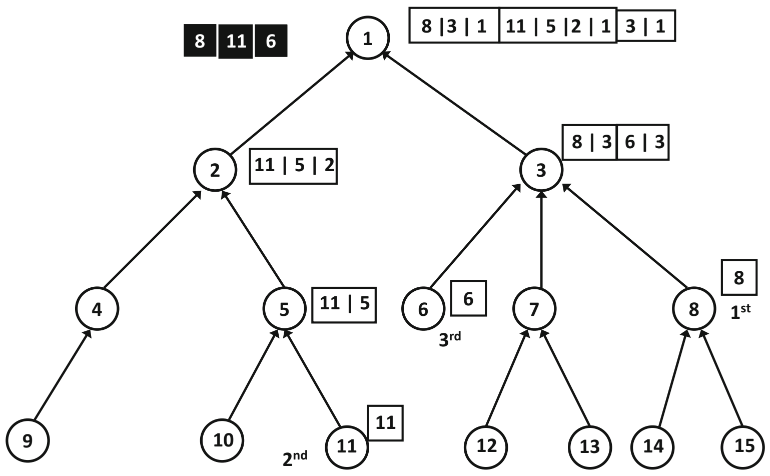


Fig. 3.3 An example for the MRA algorithm

Let us now consider that, at this point of time, P_{11} issues the second token request to enter critical section. So it puts its own process id, i.e., 11 in LDQ_{11} and sends the token request along with a copy of $\text{LDQ}_{11} = \{11\}$ on its outgoing edge. P_1 receives the request from P_2 through P_5 and following Step 3 of the algorithm, updates LDQ_1 as $\{\{8, 3, 1\}, \{11, 5, 2, 1\}\}$. The RQ is further updated as par Step 3 to RQ = $\{8, 11\}$. Process P_6 now places the third token request which updates LDQ_6 to $\{6\}$. Also, LDQ_3 is set as $\{\{8, 3\}, \{6, 3\}\}$. As a request from P_3 is already pending, the revised LDQ_3 is not carried to P_5. Only the token request from P_6, i.e., $\{6\}$ is sent to P_1. The RQ with P_1 is revised to $\{8, 11, 6\}$.

Let us assume that now, P_1 comes out of its CS and following Step 4, the token is handed over to P_3. Therefore, $\text{LDQ}_3 = \{\{8, 3\}, \{6, 3\}\}$. The RQ = $\{8, 11, 6\}$ is now with P_3. Process P_3 further passes the token to P_8 following Step 5. At this stage, RQ with P_8 is $\{8, 11, 6\}$. The condition for Step 5 matches with and therefore P_8 enters the critical section. RQ is updated to $\{11, 6\}$ and LDQ_8 to $\{3, 8\}$. Therefore, as P_8 comes out of the critical section, the entries from LDQ_8 $\{3, 8\}$ are extracted. The LDQ_8 is now reduced to as emptied. The token and RQ = $\{11, 6\}$ is handed over to P_3. The token along with the RQ is further sent to P_1 revising LDQ_3 to $\{6, 3\}$ and LDQ_1 to $\{\{11, 5, 2, 1\}, \{3, 1\}\}$.

Once again, P_1 becomes the root of the inverted tree. Following Step 5, the token along with RQ = $\{11, 6\}$ is now passed to P_5 through $P_{2,}$ while LDQ_1 changes to $\{3, 1\}$ and LDQ_5 to $\{\{11, 5\}, \{2, 5\}\}$. Then token goes to P_{11} through P_5. Now P_{11} enters the critical section after removing its id = 11 from the head of RQ and LDQ_{11}. After, P_{11} leaves the critical section, the remaining ids are extracted from LDQ_{11} leaving it empty and once again the token along with RQ = $\{6\}$ comes back to P_1. All the ids are extracted from P_1 now and the token and RQ is passed to P_3. At this point, $\text{LDQ}_3 = \{6, 3\}$. Therefore the token and RQ = $\{6\}$ is passed to P_6 leaving LDQ_3 empty. P_6 enters the critical section after removing its id = 6 from the head of both RQ and LDQ_6, both of which are emptied now.

The token stays with $P_6 = P_{\text{hold}}$ even after the process comes out of the critical section and till any other request is generated in the system. Table 3.1 is a tabular representation that illustrates how the content of the LDQs maintained at different nodes and that of RQ with P_{hold} change in successive iterations as explained above. The token requests from P_8, P_{11} and P_6 have been placed in Step 1, Step 2, and Step 3, respectively. The process P_8 enters critical section in Step 4. Process P_{11} enters critical section in Step 5 and P_6 enters its critical section in Step 6. These are marked by placing a * before the RQ entries in the appropriate cell of the table. The token is finally left with P_6 as no other request for the token has been issued.

In an identical situation, when the same three processes P_6, P_{11}, and P_6 requested for token in the same order, the Raymond's algorithm failed to ensure fairness. The algorithm MRA, however, supports the desired objective to maintain fairness as far as servicing the token request in the FCFS order is concerned.

Table 3.1 A tabular illustration of the MRA procedure

P_i	Queue	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
P_1	RQ	8	8, 11	8, 11, 6	–	–	–
	LDQ$_1$	8, 3, 1	8, 3, 1; 11, 5, 2, 1	8, 3, 1; 11, 5, 2, 1; 3, 1	11, 5, 2, 1; 3, 1	–	–
P_8	RQ	–	–	–	*8, 11, 6	–	–
	LDQ$_8$	8	8	8	1, 3, 8	–	–
P_3	RQ	–	–	–	–	6, 3	–
	LDQ$_3$	8, 3	8, 3	8, 3; 6, 3	1, 3; 6, 3	–	–
P_6	RQ	–	–	–	–		*6
	LDQ$_6$	–	–	6	6		–
P_{11}	RQ	–	–	–	–	*11, 6	
	LDQ$_{11}$	–	11	11	11	1, 2, 5, 11	

* point of execution

3.1.2 *Limitation of Algorithm MRA*

The MRA algorithm does not consider priorities of requesting processes. It works only in equal-priority processes. In distributed systems, many processes are of different priorities which cannot be solved by MRA. Thus, another important aspect of ME algorithms is to handle the priorities [14–16] of the processes.

3.2 Modified Raymond's Algorithm for Priority (MRA-P)

We present a new token-based algorithm named Modified Raymond's Algorithm for Priority (MRA-P) [13]. It ensures fairness besides serving higher priority jobs ahead of others. The MRA algorithm is modified to consider processes with different priorities such that a higher priority process gets the token first even if a lower priority process has placed its request earlier. However, the fairness property is maintained for two equal-priority processes Fig. 3.4.

Priority queue(PQ$_x$): Every process P$_x$ maintains a priority queue PQ$_x$. A process P$_x$ wishing to enter the critical section, sends a request for the token along with LDQ$_x$ and its priority status on its outgoing edge.

Local double-ended queue (LDQ$_x$): A process P$_x$ wishing to enter the critical section, sends a request for the token along with LDQ$_x$ and its priority status on its outgoing edge.

Request queue (RQ): Process P$_{hold}$ maintains a request queue that stores the process ids for the token requesting processes in an FIFO order. The RQ is transferred to the new root of the tree along with the token.

Fig. 3.4 Data structure of MRA-P algorithm

3.2.1 Data Structure and Algorithm for MRA-P

Description for MRA-P Algorithm

Suppose that process P_x wishes to enter the CS. P_x will enter its id x in its local double-ended queue LDQ$_x$ in an appropriate position depending on the priority of P_x. The priority of P_x is entered in the local priority queue PQ$_x$. Process P_x sends a token request TKN along with the priority of P_x by using LDQ$_x$ and P_x. When a process $P_y \neq P_{\text{hold}}$ receives a token request from process P_k, then id for process P_y is added in PQ$_y$ in ascending order. According to priority, P_y adds id '*k*' in LDQ$_y$ followed by its own id y. The revised LDQ$_y$ and PQ$_y$ are sent out with token request TKN. If a request from P_y is pending, then only token request TKN is sent out. When a process $P_y = P_{\text{hold}}$ receives a token request TKN from process P_k, then P_y adds priority in PQ$_y$ in descending order. Similarly, P_y adds the id '*k*' into LDQ$_y$ according to the priority of P_k followed by its own id y. Token requesting process id x is added to the RQ.

On completing the execution of CS, $P_y = P_{\text{hold}}$ performs the following. Process P_y scans the first id '*k*' from RQ. Process P_y extracts the entries in LDQ$_y$ from '*k*' to the first occurrence of '*y*' in LDQ$_y$ and removes first element of PQ$_y$. The extracted sequence of ids is reversed from '*y*' to '*k*'. Directed edge from P_m to P_y is reversed, where m is the id that follows y. Therefore, $P_{\text{hold}} = P_m$. The RQ is passed to P_m from P_y along with the token. If LDQ$_y$ is not empty then P_y places a token request to P_m along with the reduced LDQ$_y$. The ids in LDQ$_y$ are added to the head of LDQ$_m$ followed by m. The following steps are performed with the newly designated root $P_m = P_{\text{hold}}$. If first id of RQ is '*m*', then remove this first id from RQ and from the top of LDQ$_m$. Process P_m is allowed to enter its CS. On the contrary, if the first id of RQ is not '*m*', and say '*k*', then '*k*' will also be the first entry of LDQ$_m$. This is because entries in the LDQs are according to process priorities. Extract entries from LDQ$_m$ from '*k*' to '*m*'. Reverse edge P_m to P_k, if extracted sequence is of length 2 or more. Pass the token and RQ to this new root P_y. If the length of the extracted sequence is greater than 2, then send a TKN request from P_m.

A pseudo code representation for the MRA-P algorithm is given below.

Begin

```
1. Step 1: A Process $P_i$ wishing to enter the CS.
2. insert <i> in $LDQ_i$[ ];
3. insert <$x_i$> in $PQ_i$[ ]; /* $P_i$ enters its id i in the $LDQ_i$ and priority in $PQ_i$ */
4. send token_request <i, $x_i$>;
   /* $P_i$ sends a token request {i} on its out-edge along with $LDQ_j$ and the priority
   of $P_i$ */
5. Step 2: When a process $P_m \neq P_{hold}$ receives a token request from process $P_k$.
6. insert <$x_k$, $x_m$> in $PQ_m$;
   /* $P_m$ it adds priority in $PQ_m$ in descending order, $x_k > x_m$ */
7. insert <k, m> in $LDQ_m$;
```

```
    /* According to priority, $P_m$ adds the ids from $LDQ_k$ into $LDQ_m$ followed by its own id, i.e., m in this case */
    /* The revised local queue $LDQ_m$ and $PQ_m$, thus formed, is sent along with the token request {i} to the out-edge */
8.  Step 3: When a process $P_m = P_{hold}$ receives a token request {i} from process $P_k$.
9.  insert <$x_k$> in $PQ_m$; /* $P_m$ adds priority in $PQ_m$ in descending order */
10. insert <k, m> in $LDQ_m$;
    /* According to priority, $P_m$ adds the ids from $LDQ_k$ into $LDQ_m$ followed by its own id, i.e., m in this case */
11. insert <i> in RQ; /* The token requesting process id i is added to the RQ */
12. Step 4: On completing the execution of CS, $P_m = P_{hold}$ performs the following:
13. k $\in$ RQ[ ]; /* $P_m$ scans the first id k from RQ */
14. extract <k,..,m>;
    /* $P_m$ extracts entries starting from k to the first occurrence of m from $LDQ_m$ and removes the first element of $PQ_m$ */
15. reverse <m,…, k>;
    /* The extracted sequence of ids is reversed from m to k */
    /* The directed edge from $P_n$ to $P_m$ is reversed, where n is the id that immediately follows m in the reversed sequence */
16. $P_{hold} = P_n$;
17. send_tokent <n>; /* Pass RQ to $P_n$ from $P_m$ along with the token */
18. if ($LDQ_j$[ ] $\neq$ null)
    /* If $LDQ_m$ is not empty then $P_m$ places a token request to $P_n$ along with the reduced $LDQ_m$ */
19. send token_request <m, $x_m$>;
    /* The id of $LDQ_m$ is added to the head of $LDQ_n$ followed by n; */
20. endif.
21. Step 5: The newly designated root $P_n = P_{hold}$ performs the following:
22. remove the first process tuple <n> from RQ;
23. if the first id of RQ is n, then remove the first id of RQ;
    /* i.e., n, in this case, from the RQ */
24. remove the first process tuple <n>from PQ; /* Remove first element of PQ */
25. remove the first process tuple <n> from $LDQ_n$;
    /* The entry n is removed from the top of $LDQ_n$ */
26. $P_n$ enters its CS.;
27. else
28. repeat through the Step 4;
29. endif.
```

End

Illustrating the MRA-P Algorithm

In Fig. 3.5, three processes have requested for the token that is currently held by $P_{\text{hold}} = P_1$. The processes have different priorities. The higher the priority of a

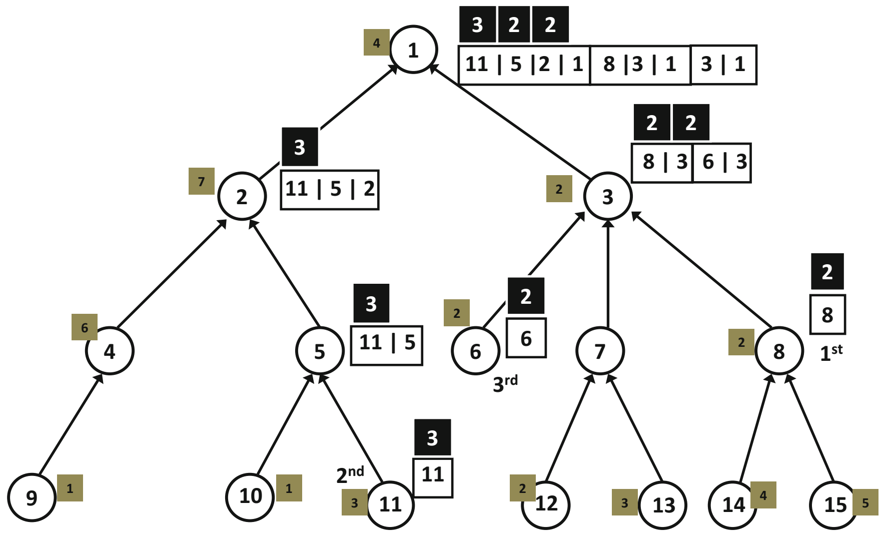


Fig. 3.5 An example for the MRA-P algorithm

process, higher is the integer value representing its priority level. A process with the highest priority is of priority value 6. In Fig. 3.5, we have shown the priority values only for the three processes P_8, P_{11}, and P_6 and these are 2, 3, and 2, respectively. Processes P_8, P_{11}, and P_6 have placed requests to enter the respective critical sections, in that order. However, according to their priorities, the order in which the processes are to be allowed to enter the respective critical sections is P_{11}, P_8, and P_6. We further assume that process P_1 remains in its CS until all three requests are made and the local queues at the nodes are updated.

Table 3.2 is a tabular representation that illustrates how the contents of the LDQs and RQs maintained at different nodes and that of RQ with P_{hold} change in successive iterations as explained above. The token requests from P_8, P_{11}, and P_6 have been placed in Step 1, Step 2, and Step 3, respectively. The RQ in Step 3 is $\{11, 8, 6\}$ and is maintained at $P_1 = P_{\text{hold}}$. In Step 4, process P_{11} has come out of its CS and the RQ is transferred to P_8 before it enters its CS. Similarly, P_8 enters the CS in Step 5.

Process P_6 enters its critical section next in Step 6. The steps in which the processes enter the respective critical sections are marked by placing a * in the RQ entries in the appropriate cell. The token is finally left with P_6 and all the data structures are left empty [Step 6]. In an identical situation, when the same three processes P_8, P_{11} , and P_6 requested for token in the same order, the Raymond's algorithm fails to ensure fairness. The MRA algorithm handles the fairness issue successfully while it does not consider priorities of requesting processes. The MRA-P algorithm, however, considers both the issues effectively.

Table 3.2 A tabular illustration of the MRA-P

P_i	Queue	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6
P_8	RQ	–	–	–	–	*8, 6	–
	PQ_8	2	2	2	2	2	–
	LDQ_8	8	8	8	8	3, 8	–
P_1	*RQ	8	11, 8	11, 8, 3	–	–	–
	PQ_1	2	3, 2	3, 2, 2	2, 2	2	–
	LDQ_1	8, 3, 1	11, 5, 2, 1; 8, 3, 1	11, 5, 2, 1; 8, 3, 1; 3, 1	8, 3, 1; 3, 1	3, 1	–
P_3	RQ	–	–	–	–	–	–
	PQ_1	2	2	2, 2	2, 2	2	–
	LDQ_1	8, 3	8, 3	8, 3; 6, 3	8, 3; 6, 3	1, 3, 6, 3	–
P_6	RQ	–		–	–	–	*6
	PQ_6	–		2	2	2	2
	LDQ_6	–		6	6	6	–
P_{11}	RQ	–	–	–	*11, 8, 6	–	–
	PQ_{11}	–	3	3	3	–	–
	LDQ_{11}	–	11	11	11	–	–

*point of execution

3.2.2 *Performance Analysis for MRA and MRA-P*

The MRA requires 2 * $(H - 1)$ control message exchanges per access to the critical section, where H is the height of the tree. In case of a balanced binary tree $H = \log_2 N$ for total N number of processes. Thus the number of control messages exchanged per access to the critical section would be $O(\log_2 N)$. This is same as the Raymond's algorithm. Thus the proposed MRA achieves fairness without introducing any additional overhead in terms of message complexity.

Simulation Result

The simulation is done in MATLAB and the simulation setup has been documented in Table 3.3. The chart in Fig. 3.6 provides a graphical representation of the comparative performance of the MRA versus the classical Raymond's algorithm. It establishes that the fairness aspect is taken care of in the proposed MRA solution only. Response time in MRA gradually increases but that of Raymond's algorithm decreases because it serves one sub-tree then and the other.

3.3 Fairness Algorithm for Priority Processes (FAPP)

Both MRA [13] and MRA-P [13] suffer from high message complexity and relatively complex routing mechanism. Besides, the MRA-P algorithm lacks in ensuring liveness of token requests, particularly from the low-priority processes.

Table 3.3 Simulation parameters of MRA

Parameters	Value
Connection topology	Logical topology of inverted tree
No of process in the tree	20
Edge length	Static
Direction of edge	Bidirectional
Maximum out-degree	3
Maximum degree	4
Minimum degree	1
Priority of process in the tree	Same
Maximum number of token requests	Gradually increased (4, 6, 8, 10, 16) 95

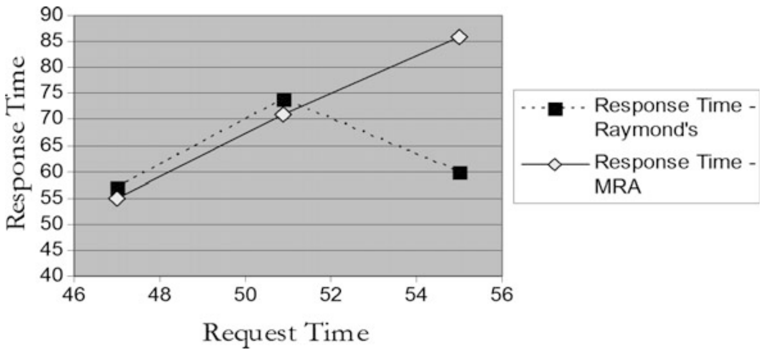


Fig. 3.6 Comparative performance of the MRA and Raymond's algorithm

Path List (PL): An arbitrary process A maintains a path list (PL_A) which is a process request queue as detailed below:

PL_A: {[<R, R_P>], [<S, S_P>], [<T, T_P>], …), where R, S, T… are the direct descendants of A and R_P, S_P, T_P… are the priorities of the respective descendent processes in a non-increasing sequence, i.e., . $R_P \geq S_P \geq T_P$…

Fig. 3.7 Date structure of FAPP algorithm

The FAPP (Fairness Algorithm for Priority Processes) algorithm [17] is a completely new algorithm that follows a simpler routing, maintains lesser information in each process, and has significantly lower message complexity compared to MRA-P. The FAPP algorithm too assumes that nodes are connected in inverted tree topology [18, 19]. The root process holds the token and is designated as P_{hold}. The fairness as well as the priority issues is taken care of by introducing one list referred as request queue. Additional space complexity is proportional to the number of token requesting processes Fig. 3.7.

3.3.1 Data Structure and Algorithm for FAPP

Description for FAPP Algorithm

A process M with priority $\{m\}$ wants to enter the critical section (CS). According to priority m, it adds the tuple $<M, m>$into PL_M. M sends token request using its outer edge. When a process $S \neq P_{hold}$ receives a token request $<K, r>$ and priority $\{r\}$ from another process K. Priority of K, i.e., r, is higher than the priority t for some process C whose token request is sent through S. Priority value t is increased by 1 i.e., $t = t + 1$. According to priority, it adds the tuple $<K, r>$ into PL_S and update PL_S in descending order of entries. S sends token request only when it receives request with higher priority or after update priority.

When a process $J = P_{hold}$ receives a token request $<K, r>$ and priority $\{r\}$ and there is a pending request from K then replace old priority of K with r. The new request from K must have a higher priority. Revise PL_J in accordance with updated priority. All intermediate processes from P_{hold} to the requesting process follow the same logic for processing.

After execution of CS, $E = P_{hold}$ performs the following. Remove the first process tuple $<M, m>$ from PL_E where m is priority of process M. M would be the new P_{hold}; the token is passed to M from E. If $PL_E[\ ] \neq$ null then send dummy token_request $<E, x>$. E places a token request to M along with the highest priority which E receives. The new root $M = P_{hold}$ receives token $<G, r>$ and removes the first tuple $<G, r>$ from PL_M. Thus, r is updated priority of process G. If G is same as M, then M enters CS otherwise token $<G, r>$ is sent. G would be the new P_{hold} and the token is passed to G from M. If $PL_M[\ ] \neq$ null then send dummy token_request $<M, x>$. G places a token request to M along with the highest priority which E receives. A pseudo code representation for the MRA-P algorithm is given below.

Begin

```
1. Step 1: When a process M with priority {r} wants to enter the critical section
   (CS).
2. insert <M, r> in $PL_M$[ ];
   /* According to priority r, it adds the tuple <M, r> into $PL_M$ */
3. send token_request <M, r>; /* M sends token request using its outer edge */
4. Step 2: When a process S $\neq$ $P_{hold}$ receives a token request <K, r> and priority
   {r} from another process K.
5. if (r > t)
6. t = t + 1;
   /* Priority of K, i.e., r, is higher than the priority t for some process C whose
   token request is sent through S. Priority value t is increased by 1 i.e., t = t + 1
   */
7. endif
8. insert <K, r> in $PL_S$[ ] in the sorted order of descending priority;
   /* According to priority, it add the tuple <K, r> into $PL_S$ */
   /* Update $PL_S$ in descending order of entries */
```

```
 9. send token_request (S, r);
    /* S sends token request only when it receives request with higher priority or
    after update priority */
10. Step 3: When a process J = $P_{hold}$ receives a token request <K, r> and priority
    {r} from another K,
11. if(k $\in$ $PL_J$[ ]) /* There is a pending request from $P_k$ */
12. replace old priority of K with r;
    /* The new request from $P_k$ must have a higher priority */
13. update $PL_J$[ ]; /* Revise $PL_j$ in accordance with decreased priority */
14. else
15. repeat Step 2;
16. endif
17. Step 4: On completing the execution of a CS, $E_j$ = $P_{hold}$ performs the
    following:
18. remove the first process tuple <M, m> from $PL_E$; /* m is priority of process M
    */
19. send token (M, m); /* M would be the new $P_{hold}$—the token is passed to M
    from E */
20. if ($PL_E$[ ] $\neq$ null)
21. send dummy token_request(E, x);
    /* E places a token request to M along with the highest priority which E
    receives */
    /* According to priority, it add the tuple <K, r> into $PL_S$ and no priority
    updating is
    required */
22. endif
23. Step 5: The newly designated root process M = $P_{hold}$ that receives token (G, r)
    performs the following:
24. remove the first tuple <G, r> from $PL_M$;
    /* r is priority or updated priority of process G */
25. if (G = = M)
    /* After executing CS, process does not update its priority */
26. enter CS;
27. else
28. send token (G, r); /* G would be the new $P_{hold}$—the token is passed to G from
    M */
29. endif
30. if ($PL_M$[ ] $\neq$ null)
31. send dummy token_request <M, x>;
    /* G places a token request to M along with the highest priority that E receives.
    According to priority, it adds the tuple <K, r> into $PL_S$. No priority updating is
    done. */
32. Endif
```

End

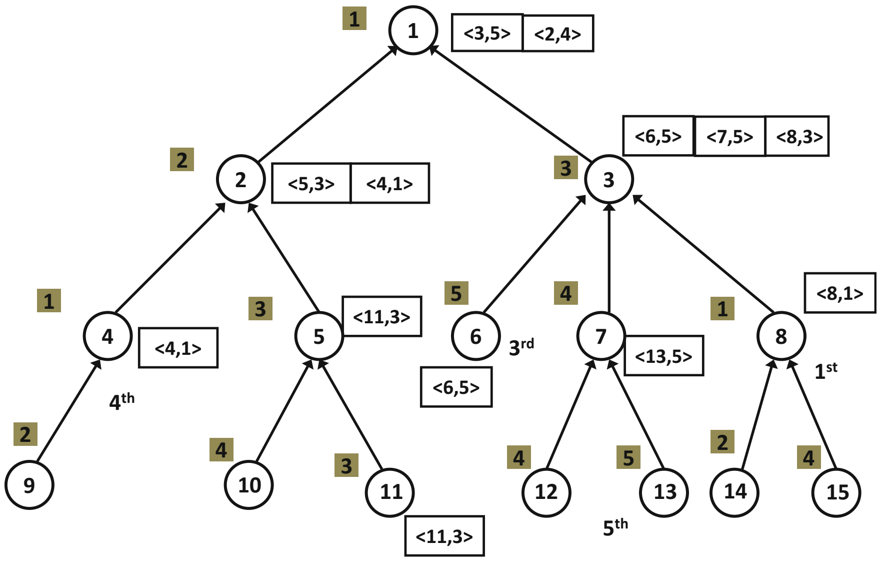


Fig. 3.8 Propagation of token requests to the root

Illustrating the FAPP Algorithm

The example under consideration deals with 15 (fifteen) processes with process ids1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, and 15 with the priority values 1, 2, 3, 1, 3, 5, 4, 1, 2, 4, 3, 4, 5, 2, and 4, respectively. We further assume that the priority 5 is highest priority of the process. Process P_8 places the first request for the token with the root node $P_1 = P_{\text{hold}}$. Next P_{11}, P_6, P_4, and P_{13} processes have placed the requests for token in that order.

According to Step 1 of the algorithm, PL_8 stores <8, 1>. The first token request is then propagated to 3 along with id of the requesting node, i.e., 8 in this case, and priority of the requesting process P_1. When process P_3 receives the request from process P_8, it puts 8 in PL_3, thus PL_3 = <8, 3>. Process P_3, now places a token request along with its own id 3 and priority of process P_8, i.e., 1, again, on its outs edge. Process $P_1 = P_{\text{hold}}$ now receives the request and puts <3, 1> in PL_1 Fig. 3.8.

Let's now consider that, at this point of time, P_{11} issues the second token request to enter CS. Process P_{11} has a priority of 3. So it puts its own process id and priority i.e., <11, 3> in PL_{11} and sends the token request to P_5 along with its own id and priority on its outgoing edge. Then Process P_5 updates as <11, 3> in PL_5 and sends the token request to P_2 and Process P_2 is updated as <5, 3>. P_1 receives the request and following Step 3 of the algorithm, it modifies PL_1 to <11, 3>, <3, 2>. Process P_1, first increases the priority of P_3 which originated at P_8, by one, then arranges the requests.

Similarly, process P_6 issues the third request to enter CS. P_6 has a priority of 5. So, it puts its own process id, i.e., <6, 5> in PL_6 and sends the token request along with its own id and priority on its outgoing edge. Then process P_3 receives the request from process P_6, the process P_3 puts 6 in PL_3, thus PL_3 = <6, 5>, <8, 2>. P_3, now places a token request along with its own id 3 and priority of process P_6, i.e., 5, again, on its outs edge. Process $P_1 = P_{hold}$ now receives the request and puts <3, 5> in $PL_{1.}$ Then PL_1 is modified to <3, 5>, <2, 4>. Then, process P_4 issues the fourth token request. It enters <4, 1> in PL_4. Now process P_2 gets the token and adds <4,1> in PL_2. Since priority of P_4 is 1, which is lower than the priority of $P_{11.}$ As process P_2 places a token request to P_1 along with the highest priority which P_2 receives so P_2 is just appended of request of P_4 into PL_2. Then PL_2 is updated as <5, 3>, <4,1>. Now, Process P_{13} sent fifth request. It enters <13, 5> in PL_{13}. Now process P_7 gets the token and adds <13, 5> in PL_7. Then process P_3 receives the request from process P_7, the process P_3 puts 7 in PL_3, thus PL_3 = <6, 5>, <7, 5>, <8, 3>. Since P_3 already sends a request on behalf of P_6 with same priority as for P_{13}. So P_2 is just appended in PL_3. No further request is send to P_1 for P_{13} Fig. 3.9.

After receiving all the token requests, P_1 comes out of its CS and following Step 4, the first tuple from PL_1 is extracted. The token is handed over to P_3. Now, PL_1 = <2, 4>. As PL_1 is not empty, a dummy request from P_1 is sent to P_3 along with the highest priority, which process P_1 receives, i.e., token request as <2, 4>. The token reaches P_3 and at this point, PL_3 = <6, 5>, <7, 5>, <1, 4>, <8, 3>. Process P_3 extracted <6, 5>. The condition for Step 5 matches and therefore P_6 enters the CS after updating PL_6 = <3, 5>, which is shown in Fig. 3.9. Therefore, as P_6 comes out of the CS, the entries from PL_6 is extracted once again and the

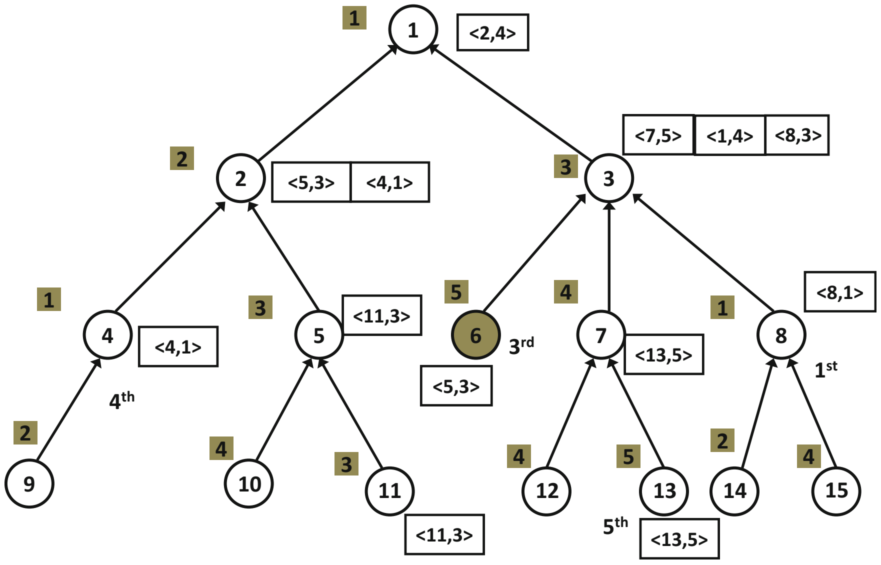


Fig. 3.9 Sequence of the requesting processes to enter the CS

token is handed over to P_3 and leaving $PL_6 =$ (i.e., empty) and once again, P_3 becomes the root of the inverted tree.

Similarly, using Step 4 and 5, process P_{13}, P_{11}, P_8 and P_4 enters CS, respectively. When enters the CS after removing its tuple for PL_4, all the lists are emptied. The token stays with $P_4 = P_{\text{hold}}$ even after the process comes out of the CS and till any other request is generated in the system.

Table 3.4 is a tabular representation of how the process P_6 enters CS in Step 6, P_{13} enters CS in Step 7 and process P_4 enters CS in Step 10. These are marked by placing a * before the PL entries for the process in the CS, in the appropriate cell of the table. The # symbol indicates that a new token request is issued. The '-' entries in the cells of the tables are used to indicate null lists. In the case study, the token is finally left with P_4, as no other request for the token has been issued.

3.3.2 Performance Analysis for FAPP

We analyze the performance of our algorithms using the following metrics: message complexity per CS request, fairness, correctness of the FAPP algorithm, time complexity of the FAPP algorithm, synchronization delay, and maximum concurrency for FAPP algorithm.

Message Complexity
The control messages for the FAPP algorithm are of two types. A request control message initiates from a process that requests the token and moves toward the root P_{hold} of the inverted tree. The token transfer control messages flow from the root process toward one of the token requesting processes, at a time, and every time after current P_{hold} completes its own critical section. The complexity of control messages per CS access are estimated by separately computing the same for both types of control messages and making a sum thereafter.

When a process P_m requests for a token, its id $\{m\}$ and the priority value, say $\{m_p\}$ both moves up to its parent process P_n. The process P_n inserts $\{m, m_p\}$ in PL_n and sends a fresh request $\{n\}$ along with the priority value $\{n_p\}$ to its own parent. The relaying of token request from P_m continues right up to the root process P_{hold}. This may stop at any intermediate process P_k, if there is already a pending request for P_n whose priority is higher than that of P_m. The maximum number of request control messages for each token request from P_m is same as the level of P_m in the tree.

Lemma 3.1 *The number of request control messages along with the request paths required for a new token request lies in the interval* $[0, (H - 1)]$, *where H is the height of the tree.*

Proof The total number of nodes on the unique path from the requesting node P_i to root node P_{hold} is $L + 1$, where L is the level of node P_i in the inverted tree. In the worst case, the request control message from P_i to its parent P_j along with the

Table 3.4 A tabular illustration of FAPP

P_{id}	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8	Step 9	Step 10
	First req	Second req	Third req	Fourth req	Fifth req	P_6 exe	P_{13} exe	P_{11} exe	P_8 exe	P_4 exe
P_1	<8, 1>	<2, 3> <3, 2>	<3, 5> <2, 4>	<3, 5> <2, 4>	<3, 5> <2, 4>	<2, 4>	<2, 4>	<3, 3>	<2, 1>	–
P_{11}		#<11, 3>	<11, 3>	<11, 3>	<11, 3>	<11, 3>	<11, 3>	*<11, 3> <5, 3>	–	–
P_6			#<6, 5>	<6, 5>	<6, 5>	*<6, 5> <3, 5>	–	–	–	–
P_3	<8, 1>	<8, 1>	<6, 5> <8, 2>	<6, 5> <8, 2>	<6, 5> <7, 5> <8, 3>	<7, 5> <2, 4> <8, 3>	<2, 4> <8, 3>	<8, 3>	<1, 1>	–
P_2		<5, 3>	<5, 3>	<5, 3> <4, 1>	<5, 3> <4, 1>	<5, 3> <4, 1>	<5, 3> <4, 1>	<3, 3> <4, 1>	<4, 1>	–
P_5		<11, 3>	<11, 3>	<11, 3>	<11, 3>	<11, 3>	<11, 3>	<3, 3>	–	–
P_8	#<8, 1>	<8, 1>	<8, 1>	<8, 1>	<8, 1>	<8, 1>	<8, 1>	<8, 1>	*<8, 1> <3, 1>	–
P_7					<13, 5>	<13, 5>	<3, 4>	–	–	–
P_4				#<4, 1>	<4, 1>	<4, 1>	<4, 1>	<4, 1>	<4, 1>	*<4, 1>
P_{13}					#<13, 5>	<13, 5>	*<13, 5> <7, 4>	–	–	–

# point of request, *point of execution

priority value of P_i is relayed right up to the root node using the intermediate nodes. Thus the maximum number of such messages will be L. If P_i happens to be a leaf node on one of the longest paths of the tree, then $L = H - 1$ following the definition of height H of tree. The maximum number of request control message would thus be $L = (H - 1)$.

It is trivial to prove that in an extreme case, if there is already a pending request of higher priority from the requesting node P_i to its parent then no request control message is sent at all. Thus, the total number of request control messages for a new token request from any node P_i, lies within the closed interval of $[0, (H - 1)]$, where H is the height of the tree.

Lemma 3.2 *The maximum number of token transfer control messages for each CS access is* $(H - 1)$, *where H is the height of the tree.*

Proof The FAPP algorithm stores the request paths of the tokens using local queues in each node. When the root node, say P_m, comes out of its critical section, it will pass the token to some node, say P_i, which is on top of the PL_m. This transfer is performed using the intermediate nodes from P_m to P_i, where a transfer control message is issued to mobilize the token through each intermediate node.

If the level of node P_i is L, then the total number of nodes on the unique path from P_m to P_i would be $L + 1$, inclusive of both the source and the target nodes. The number of transfer control messages along the path would therefore be L. The value of L, is again $(H - 1)$ for the extreme case when target node P_i happens to be leaf node on one of the longest paths of the tree of height H. Thus the statement of Lemma 3.2 is proved.

Theorem 3.1 *The maximum number of control messages per CS access for FAPP using a N-node balanced binary tree is* $O(\log N)$.

Proof Lemma 3.1 establishes that the total number of request control messages for a new token request from any node P_i, lies within the closed interval of $[0, (H - 1)]$, where H is the height of the tree.

Similarly, from Lemma 3.2, the maximum number of token transfer control messages for each CS access is again $(H - 1)$, where H is the height of the tree.

Thus the total number of control messages using a N-node balanced binary tree is $2 * (H - 1) = 2 * (\log_2 N - 1) \approx O(\log N)$.

The actual number of messages could even be smaller for m-way trees, for $m > 2$. The message complexity for such a topology however, remains the same, i.e., $O(\log N)$.

Lemma 3.3 *Algorithm FAPP ensures fairness.*

Proof Let us prove this by contradiction. In other words, let us assume that, even if the request from a process P_r reaches the root node first, another equal-priority process P_j that requested for token after P_r gets it ahead of P_r. Without any loss of generalization, let us further assume that P_j and P_r are direct descendants of P_{hold}. This assumption simplifies the scenario as we can do away with the routing through the intermediate nodes. We know that P_j is to get the token before P_r, and priority of both processes is the same. Further, the allocation of token initiates by extracting the first element, say P_k, from $\mathrm{PL}_{\mathrm{hold}}$ and then by passing the token toward P_k using the intermediate nodes.

$$\text{Therefore, the occurrence of } P_j \text{ must precede that of } P_r \text{ in } \mathrm{PL}_{\mathrm{hold}}. \tag{3.1}$$

On the other hand, for token requests from two equal-priority processes, the entries in $\mathrm{PL}_{\mathrm{hold}}$ are to be made following the order of arrival of request control messages. P_j and P_r are of equal priority and the token request from P_r arrives before that from P_j.

$$\text{Therefore, the occurrence of } P_r \text{ must precede that of } P_j \text{ in } \mathrm{PL}_{\mathrm{hold}}. \tag{3.2}$$

The conditions 1 and 2 are in conflict. This proves that the converse of our initial hypothesis is correct. Therefore, the Fairness Algorithm for Priority Processes (FAPP) indeed, meets the fairness criteria.

Correctness of the FAPP Algorithm

The correctness of a distributed control algorithm is often defined as a collection of two separate characteristics, safety, and liveness. A process synchronization algorithm satisfies the safety property if it ensures that no two processes are allowed to access the respective critical sections simultaneously. In other words, the algorithm is safe if the competing processes access the critical sections in a mutual exclusion. Liveness demands that any process that requests for the token must get it eventually.

Safety

Any token-based mutual exclusion algorithm is inherently safe as there is only one token and the process that enters CS must get the token first. FAPP is a token-based algorithm where all the competing processes share the same token. The process P_{hold} that holds the token can only access its CS. Thus FAPP maintains the safety property.

Liveness

Apparently liveness and priority-based fairness are just two conflicting aspects to deal with. If an algorithm prefers a higher priority process, then it is almost obvious that a lower priority job may suffer from starvation. This will be denial of likeness. However, this paper aims to strike a balance on this. The dynamics in revising process priorities described in Step 2 of FAPP ensures that even a token request from a lowest priority process would eventually be granted as the priority itself changes dynamically. Thus liveness as defined in Chap. 2 would be maintained for FAPP.

Concurrent Occupancy

In the FAPP algorithm, before a process $P_m = P_{\mathrm{hold}}$ starts execution in CS, it extracts the first id from PL_m. If the first id of PL_m i.e., $\{m, P_m\}$, is equal to the own priority of P_m, replace the positions of PL_m by the id of the sender and the priority sent by it. This is done only if the PL_m is not empty. Then P_m enters its CS. Thus the requesting process enters in its CS only after it receives the token. Hence, it is proved that FAPP algorithm satisfies the concurrent occupancy property.

Time Complexity

The FAPP algorithm iterates in two stages for each access to CS. As explained in the proof of theorem 3.1, a maximum number of $(H - 1)$ iterations, H being the height of the tree, is required for carrying a token request form any node to the root node. Similarly, in order to pass the token to a requesting leaf node, the number of iterations will not be more than $H - 1$. Thus the overall time complexity will be $2 * (H - 1) \approx O(\log N)$, N being the number of nodes in a balanced binary tree.

Theorem 3.2 *The maximum concurrency of the proposed FAPP algorithm is L, where L is the maximum level of any node in the tree.*

Proof In FAPP algorithm, when a node P_i requests for token, its id and the priority value, say $\{p\}$ both moves up to its immediate parent node P_j. The node P_j inserts $\{\text{i}, p\}$ in PL_j before sending a fresh request {j} along with the priority value $\{p\}$ to its own parent. This relaying of token request from originating node P_i continues right up to the root node P_{hold}. The process, may however, stop at any intermediate node P_k, if there is already a pending request for P_j whose priority is higher than that of P_i. Thus, the maximum number of request control messages for each token request from P_i could be L, where L is the level of node P_i in the tree. Therefore, maximum concurrency of our algorithm is L.

Simulation Result

The experimental setup has been documented in Table 3.5. In Fig. 3.10, it is found that the FAPP algorithm uses more control messages in comparison to the Raymond's algorithm.

However, this is justified as FAPP ensures the fairness among the priority processes competing for the token. This requires a slightly higher number of control messages per token request.

Table 3.5 Simulation parameters of FAPP

Parameters	**Value**
Platform	MATLAB 7.1
Connection topology	Inverted tree
Nodes in the tree	15
Edge length	Static
Maximum degree	3
Minimum degree	1
Priority of process	Per-assign and fixed
Maximum number of request	Gradually increased (5,6,7)
Candidate select	User choice
Links between the processes	Bidirectional.
Measure	Execution time and control message

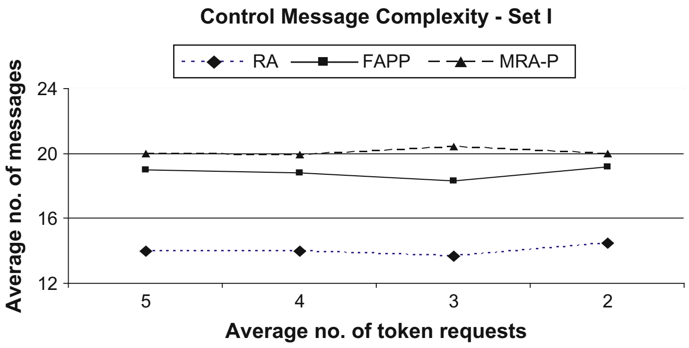


Fig. 3.10 Comparative performance for message complexity

3.4 Concluding Remarks on Tree-Based ME Algorithms

The fairness aspect in terms of responding to the token requests from equal-priority processes in a FIFO order is not always ensured in existing token-based ME algorithms. MRA solves the fairness problem of Raymond's algorithm [9]. In MRA-P [13], the competing processes get token first considering their priorities and then in the order in which the requests have been entered. This suggests that MRA-P should be used only when strict priority ordering is required. However, MRA-P uses many control messages. FAPP [17] algorithm overcomes this and ensures fairness amongst equal-priority processes too.

References

1. Lin, T., Moh, S., Moh, M.: Brief announcement: improved asynchronous group mutual exclusion in token passing networks. In: Proceedings of PODC'05, pp. 275–275 (2005)
2. Zarafshan, F., Karimi, A., Al-Haddad, S.A.R., Saripan, M.I., Subramaniam, S.: A preliminary study on ancestral voting algorithm for availability improvement of mutual exclusion in partitioned distributed systems. In: Proceedings of International Conference on Computers and Computing (ICCC'11), pp. 61–69 (2011)
3. Saxena, P.C., Rai, J.: A survey of permission-based distributed mutual exclusion algorithms. Comput. Stan. Interfaces **25**(2), 159–181 (2003)
4. Helary, J.M., Mostefaoui, A., Raynal, M.: A general scheme for token and tree based distributed mutual exclusion algorithm. IEEE Trans. Parallel. Distrib. Syst. **5**(11), 1185–1196 (1994)
5. Naimi, M., Trehel, M., Arnold, A.: A log(N) distributed mutual exclusion algorithm based on path reversal. J. Parallel. Distrib. Comput. **34**(1), 1–13 (1996)
6. Sanders, B.: The information structure of distributed mutual exclusion algorithm. ACM Comput. Syst. **5**(3), 284–299 (1987)
7. Singhal, M.: A dynamic information structure mutual exclusion in distributed system. IEEE Trans. Parallel. Distrib. Syst. **3**(1), 121–125 (1992)

8. Kakugawa, H., Yamashita, M.: Local coteries and a distributed resource allocation algorithm. Trans. Inf. Process. Soc. Japan **37**(8), 1487–1496 (1996)
9. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM. Trans. Comput. Syst. **7**, 61–77 (1989)
10. Housini, A., Trehel, M.: Distributed mutual exclusion token-permission based by prioritized groups. In: Proceedings of ACS/IEEE International Conference, pp. 253–259 (2001)
11. Mueller, F.: Prioritized token-based mutual exclusion for distributed systems. In: Proceedings of the 9th Symposium on Parallel and Distributed Processing, pp. 791–795 (1998)
12. Kanrar, S., Choudhury, S., Chaki, N.: A link-failure resilient token based mutual exclusion algorithm for directed graph topology. In: Proceedings of the 7th International Symposium on Parallel and Distributed Computing—ISPDC 2008 (2008)
13. Karnar, S., Chaki, N.: Modified Raymond's algorithm for priority (MRA-P) based mutual exclusion in distributed systems. In: Proceedings of ICDCIT 2006. LNCS 4317, pp. 325–332 (2006)
14. Mittal, N., Mohan, P.K.: A priority-based distributed group mutual exclusion algorithm when group access is non-uniform. J. Parallel. Distrib. Comput. **67**(7), 797–815 (2007)
15. Barbara, D., Garcia-Molina, H., Spauster, A.: Increasing availability under mutual exclusion constraints with dynamic vote reassignment. ACM Trans. Comput. Syst. **7**(4), 394–428 (1989)
16. Zarafshan, F., Karimi, A., Al-Haddad, S.A.R., Saripan, M.I., Subramaniam, S.: A preliminary study on ancestral voting algorithm for availability improvement of mutual exclusion in partitioned distributed systems. In: Proceedings of International Conference on Computers and Computing (ICCC'11), pp. 61–69 (2011)
17. Kanrar, S., Chaki, N.: FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems. Special issue on recent advances in network and parallel computing. Int. J. Networks 5(1), 11–18 (2010). ISSN: 1796-2056
18. Naimi, M., Thiare, O.: Distributed mutual exclusion based on causal ordering. J. Comput. Sci. 398-404 (2009). ISSN 1549-3636
19. Chaki, N., Chaki, R., Saha, B., Chattopadhyay, T.: A new logical topology based on barrel shifter network over an all optical network. In: Proceedings of 28th IEEE International Conference on Local Computer Networks (LCN '03), pp. 283–284 (2003)

Chapter 4
A Graph-Based Mutual Exclusion Algorithm Using Tokens

In some of the earlier works, token-based algorithms for ME are presented for the distributed environment [1–6]. These are for inverted tree topology. However, such a stable, hierarchical topology is quite unrealistic for many types of networks, e.g., mobile ad hoc networks (MANET), due to frequent link failures. There are existing ME algorithms that work on a directed, acyclic graph (DAG). Again a DAG-based [6] solution may not be ideal for the highly dynamic mobile environment [7–10] too. Moreover, these do not ensure fairness. The solution described in [11] works on a dynamically changing DAG structure. Their algorithm handles link failures [11–14] and formation.

4.1 Link Failure Resilient Token-Based ME Algorithm for Directed Graph (LFRT)

We introduced a new token-based ME algorithm for directed graph topology, with or without cycles (LFRT) [11]. Besides maintaining the correctness in terms of liveness and safety, the algorithm ensures fairness in allocating the token on a FCFS basis. It offers solution paths even in presence of link failures. The most significant advantages of LFRT [11] algorithm are its ability to handle the link failures. LFRT algorithm is able to find an alternate path, in case of a link failure. The token request messages do not form cycles. This keeps message complexity low.

N_p= Neighborhood set of a node P is an ordered set of 2-tuples N_P = {(T, P) v (P, T)}, where T is a node in the set of neighboring nodes of P. The 2-tuple entry is (T, P) or (P, T), depending on the direction of the link to or away from P respectively.

N_i = {(i, j)} where there exists an edge from P_i to P_j. It's called Neighborhood set.

3-Tuple = <i, j, k> where, i: Id. of the original token requesting node,

j:Id.ofsome intermediate node and k: Id. of the next hop destination node.

LRQ_i = {<j, k>}: Local Request Queue in P_i. This is maintained locally in every node P_i. The entry <j, k> in LRQ_i, is for the token request initiated by P_j that reaches P_i through P_k.

ORQ = {i}: Original Request Queue. This is a global list maintained in P_{hold}. Id for each token request initiating node P_i is stored in ORQ.

Fig. 4.1 Data structure of LFRT algorithm

4.1.1 Data Structure and Algorithm for LFRT

(See Fig. 4.1).

Description for LFRT Algorithm

Each process P_x maintains an ordered set of 2-tuple $N_x = \{(x, y) \vee (y, x)\}$, where P_y is included in the set of neighboring processes of P_x. The 2-tuple entry is (y, x) or (x, y), depending on the direction of the link to or away from P_x, respectively. The set N_x, thus represents the topological neighborhood of process P_x. We present to refer set N_x as the neighborhood set for P_x. Besides, each process P_x maintains a double-ended local request queue LRQ_x to store the identifiers of the nodes from that requested for token.

A process P_x must get the token before it enters the CS. P_x appends its own id, i.e., $<x, x>$ in its local request queue LRQ_x. Process P_x initiates a token request message $<x, x, y>$ and sends it to its neighbor P_y, for each 2-tuple (x, y) stored in P_x. The token request message format is <original requesting process id, intermediate process id, next hop process id>. Process P_x need not store the 3-tuple that it sends. When some process P_y receives this request $<x, x, y>$, it checks LRQ_y for x. If it already exists in LRQ_y, no further request is generated by P_y on behalf of P_x. On the contrary, if this is the first request from P_x through P_y, then P_y sends a request message $<x, y, k>$ to process P_k, for each 2-tuple $(y, k) \in N_y$. No such request, however, is generated on a path if $x = k$. This ensures that the messages do not face a count to infinity problem. Process P_y also appends id of the process P_x in LRQ_y. This continues till the request reaches $P_t = P_{\text{hold}}$, the token holder process. The token holder P_t maintains an additional queue called original request queue (ORQ), to store the id of the process that generates the token request. This is stored as the first entry of the 3-tuple. In this case, id 'x' is to be appended to ORQ. On return path, if the local request queue (LRQ) of the token sender process, say P_s, is not empty then it sends a dummy request $<0, s, k>$ to P_k along with the token. The dummy request starts with a special id “0”. On receipt of the token and the dummy request from P_s, process P_k extracts an id from the LRQ$_k$ and places <0, *s*> at the front of LRQ$_k$. The dummy token requests are there to bring the token back to the predecessor when requests are pending. This ensures fairness in the token allocation process. A pseudocode representation for the LFRT algorithm is given below.

Begin

```
1.  Step 1: When a process $P_i$ wants to enter the CS,
2.  insert {<i, i>} in LRQ$_i$; /* $P_i$ adds {<i, i>} into its local request queue LRQ$_i$ */
3.  send token_request <i, i, j>;
    /* $P_i$ sends token request messages <i, i, j>, $\forall$ (i, j) $\in$ $N_j$, the neighborhood set
    of $P_i$ */
4.  Step 2: Node $P_k \neq P_{hold}$ on receipt of a 3-tuple <i, j, k> from node $P_j$.
5.  if (i $\in$ LRQ$_k$ .OR. out-degree of $P_k$ == 0)
6.  send <bounce, i, j> to $P_j$;
    /* Node $P_k$ detects that it has earlier received the token request originated by $P_i$
    and sends a bounce message back to $P_j$ */
    /* This ensures that control messages do not suffer from the count to infinity
    problem even in the presence of cycles in the underling directed graph topology
    */
    /* Assumed that the edge between $P_j$ and $P_k$ is temporarily reversed during the
    bounce message transmission */
7.  else
8.  insert <i, j> in LRQ$_k$;
    /* Appends id of the requesting node $P_i$ and sender $P_j$ as a doublet <i, j> in
    LRQ$_k$ */
9.  send token_request <i, k, x>;
    /* Node $P_k$, sends a 3-tuple <i, k, x>, $\forall$ (k, x) $\in$ $N_k$, the neighborhood set of $P_k$.
    */
10. end if
11. Step 3: Node $P_{hold}$ = $P_t$ on receipt of a 3-tuple <i, s, t> from some node $P_{s.}$
12. if (i $\in$ LRQ$_t$)
13. send <bounce, i, s> to $P_s$;
14. else
15. insert <i, s> in LRQ$_t$;
    /* $P_{hold}$ = $P_t$ appends ids of node $P_i$ and $P_s$ as a doublet <i, s> in LRQ$_t$ */
16. insert {i} in ORQ$_t$;
    /* The token holder $P_t$ appends the id of the original token requesting node $P_i$
    into the ORQ maintained with $P_{hold}$ */
17. end if
18. Step 4: Node $P_k$, on receipt of the message <bounce, i, k>
19. If (count_bounce[i] > 0) then
20. increase count_bounce[i] by 1;
```

```
21. else
22. create count_bounce[i] = 1;
    /* If this is the first bounce message for token request from $P_i$, then a new local
    data structure count_bounce[i] in $P_k$ is created and initialized to 1. If $P_k$ has
    earlier received <bounce, i, k>, then count_bounce[i] is increased by 1 */
23. end if
24. if (count_bounce[i] == out-degree of $P_k$)
    /* Condition will be true when all the neighbors of $P_k$, bounced the request
    from $P_i$ */
25. delete <i, j> from $LRQ_k$;
    /* Deletion of <i, j> in $P_k$ indicates that there is no new route through $P_k$ for the
    request */
26. send <bounce, i, j> to $P_j$;
    /* Assumed that the edge between $P_j$ and $P_k$ is temporarily reversed during the
    bounce message transmission */
27. end if
28. Step 5: Node $P_t = P_{hold}$ performs the following after it comes out of CS
29. if (ORQ $\neq$ NULL)
30. remove the first entry <m, k> from $LRQ_t$.
31. read the first process id {m} from ORQ;
    /* scans 'm' from ORQ. Therefore, node $P_m$ is the original token requesting
    node and node $P_k$ happens to be the first node from $P_{hold}$ to $P_m$ */
32. send <token, k>;
33. send <ORQ, k>;
    /* The directed edge from $P_t$ to $P_k$ is reversed, and token is sent along with
    ORQ */
34. end if
35. if ($LRQ_t$ $\neq$ NULL)
36. send token_request <0, t, k>;
    /* If reduced $LRQ_t$ is not empty then $P_t$ sends a dummy token request to new
    $P_{hold}$ */
37. end if
38. Step 6: The newly designated $P_k = P_{hold}$ node performs the following as it
    receives <token, k>, <ORQ, k> and the dummy request <0, t, k> [optional]
    from $P_t$:
39. remove the first entry <m, j> from $LRQ_k$;
40. if ($P_k$ receives dummy request <0, t, k> from $P_t$)
41. insert <0, t> as the first entry of $LRQ_k$;
    /* $P_{hold} = P_k$ replaces the first entry of the local queue with <0, t> */
42. end if
43. read the first process id {m} from ORQ;
44. if ((m == j) && (k == m))
45. remove the id {m} from ORQ;
```

```
46. enter CS;
47. else
48. send <token, j>;
49. send <ORQ, j>;
    /* The directed edge from $P_k$ to $P_j$ is reversed, and token is sent along with ORQ */
50. if ($LRQ_m \neq$ NULL)
51. send token_request <0, k, j>;
    /* If the reduced $LRQ_k$ is not empty then node $P_k$, sends a dummy token request to the new $P_{hold}$ */
52. end if
53. end if
54. Step 7: In case of a link failure $\{k \rightarrow i\}$ during token passing, an intermediate node K holding the token $P_k = P_{hold}$ performs the following:
55. replace {<i, m} by {<i, λ>} in $LRQ_k$;
    /* $P_{hold} = P_k$ replaces all occurrences of the second entry m by a dummy λ in $LRQ_k$ */
    /* $P_m$ sends a request on behalf of $P_i$ using the 3-tuple <i, m, k> to its neighbors $P_k$ */
56. delete the 2-tuple $((k, i) \vee (i, k)) \in N_k$ for node $P_k$;
    /* $P_k$ deletes the 2-tuple (k, i) or (i, k), from the neighborhood set of $N_k$ for node $P_k$ */
57. $TN_k = N_k$;
    /* $TN_k$ stores the link information starting from $P_k$ till some link to $P_m$ is stored in it */
    /* $TN_k$ needs to be created only in case of a link failure on the shortest path from $P_k$ */
58. repeat
59. add $\{(s, h) \vee (h, s)\}$ to $TN_k$ $\forall$ s $\in TN_k$ $\forall P_h \in N_s$;
    /* $P_k$ copies all possible neighbor information of its existing neighbors into $TN_k$ after the first iteration and adds next level of neighbors in every new iteration */
60. until $(\{(n, s) \vee (s, n)\} \in N_k)$
61. if $(\{(n, x) \vee (x, n)\} \in TN_k)$
    /* If literal 'n' occurs in the $TN_k$, then an alternate path from $P_{hold} = P_k$, to the token requesting node $P_n$ can eventually be traced starting from $P_k$ through $P_x$ */
62. replace {<i, λ>} by {<i, x>} in $LRQ_k$
    /* replace all occurrences of the dummy entry 'λ' in $LRQ_k$ by 'x' */
63. end if
```

End

Illustration of the Algorithm

In Fig. 4.2, we find a directed graph with six nodes labeled as P_1 through P_6. Each node P_i maintains a neighborhood set N_i that consists of a set of 2-tuple. Each

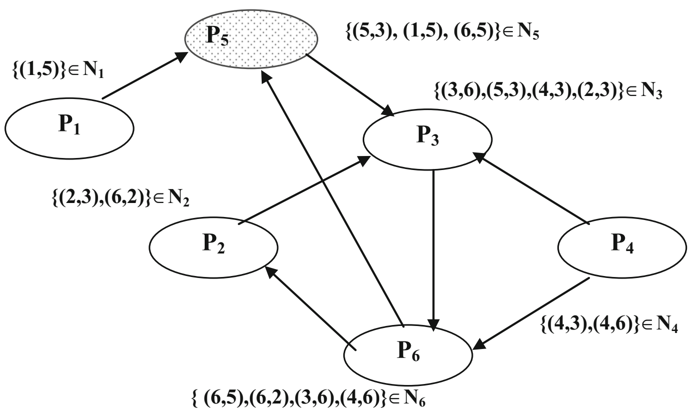


Fig. 4.2 Initial topology—P_5 holds the token

2-tuple $\{(i, j)\}$ represent a directed link from P_i to P_j. As for example, P_3 = {(3, 6), (5, 3), (4, 3), (2, 3)}.

This represents the links that originate from or incident up on P_3. The neighborhood sets for the six nodes are shown in the Fig. 4.2.

As shown in Fig. 4.3, the node P_4 has placed the first token request. The entries in LRQ's and in the ORQ are also shown. As per Step 1 of the LFRT algorithm, {<4, 4>} is stored in LRQ$_4$. Node P_4 sends its request using the 3-tuple <4, 4, 6> and <4, 4, 3> to the neighbors P_6 and P_3, respectively. On receipt of the request from P_4, nodes P_3 and P_6 insert the first two literals of the 3-tuple, i.e., {<4, 4>} into LRQ$_3$ and LRQ$_6$, respectively. Now P_3 sends a request on behalf of P_4 using the 3-tuple <4, 3, 6> to its neighbors P_6. Node P_6, however, already received the

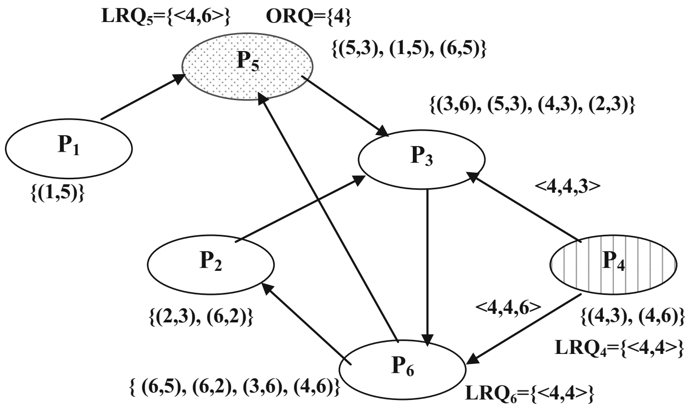


Fig. 4.3 Token request from node P_4 reached node P_5

same request form P_4 directly. Thus, the 3-tuple from P_3 is to be ignored following Step 2 of the LFRT algorithm.

Now, P_6 sends the request of P_4 to its neighbors P_2 and P_5 using the 3-tuple <4, 6, 2> and <4, 6, 5> respectively following Step 2. Similarly, P_3 rejects the request form P_2 following Step 2 because 4 is in LRQ_3. $P_5 = P_{hold}$, receives the 3-tuple <4, 6, 5> and put <4, 6> in LRQ_5. It also puts the first element of 3-tuple in ORQ, i.e., ORQ = {4}. Let us now consider that, P_1 issues a second token request. So it puts {<1, 1>} in LRQ_1 and sends the 3-tuple <1, 1, 5> on its outgoing edge to P_5. Node P_5 receives the 3-tuple and following Step 3 of the LFRT algorithm, updates LRQ_5 as {<4, 6>, <1, 1>}. The ORQ is updated to {4, 1}.

The node P_6 now places the third request which updates LRQ_6 to {<4, 4>, <6, 6>} and sends the 3-tuple <6, 6, 5> and <6, 6, 2> to the neighbors P_5 and P_2, respectively. Node P_2 sends 3-tuple <6, 2, 3> to P_3 and node P_3 has no other tuple, but <6, 3, 6> to be sent to P_6 only. However, according to Step 2, node P_3 detects that it has earlier received the token request originated by P_6 and sends a *bounce* message back to P_2. The Step 4 of the algorithm describes how the local request queues of P_2 and P_3 are restored as there is no path to P_{hold} from P_6 through P_2. Node P_5 updates the LRQ_5 and ORQ to {<4, 6>, <1, 1>, <6, 6>} and {4, 1, 6} respectively on receipt of the 3-tuple <6, 6, 5> directly from P_6.

The situation after all three token requests from P_4, P_1 and P_6 are registered with the token holding node P_5 is illustrated in Fig. 4.4. Say, P_5 now comes out of its critical section. P_5 would pass the token and ORQ to the node from which it received the request first, as described in line 29, Step 5. The first entry of ORQ is now '4'. Then the first entry <4, 6> is removed from LRQ_5. Node P_5 has to pass the token to P_4 via P_6. As in Step 6, the directed edge from P_6 to P_5 is reversed and the token along with the ORQ is passed. Since LRQ_5 is not yet empty, a dummy request <0, 5, 6> is sent to P_6. Node P_6 becomes the token holder (Fig. 4.5).

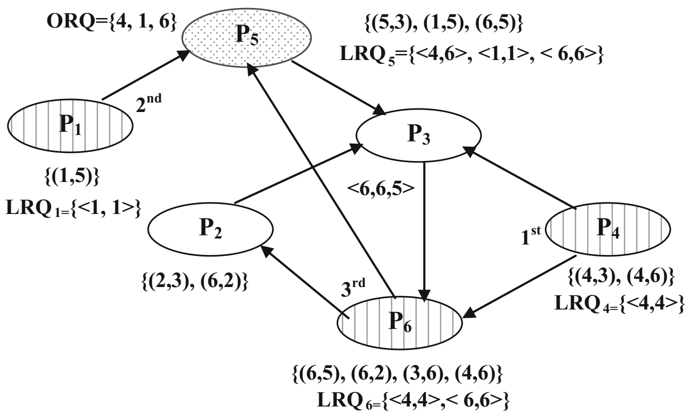


Fig. 4.4 Token requests from node P_4, P_1 and P_6 reached node P_5 in that order

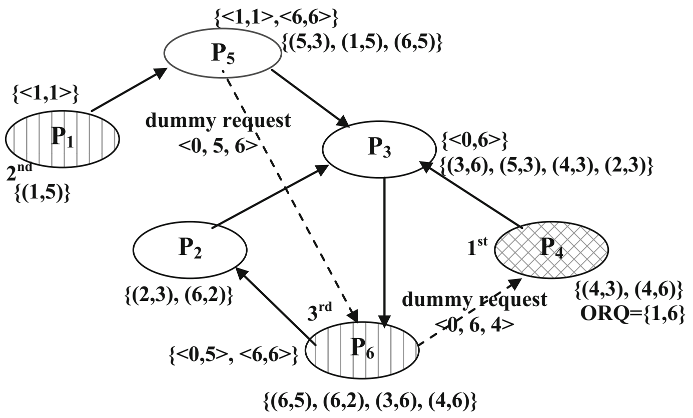


Fig. 4.5 Node P_4 holds the token and ORQ—the dummy requests are displayed

It removes the first entry <4, 4> from LRQ_6 and adds <0, 5> at the beginning of LRQ_6, thus updating it to {<0, 5>, <6, 6>}. Process P_6 further passes the token to P_4. According to Step 6, P_4 now enters the CS after updating the ORQ to {1, 6}. P_4 also receives a dummy request <0, 6, 4> from P_6 and LRQ_4 is now {<0, 6>}.

Thereafter, P_4 comes out of the CS. The LRQ_4 is now empty and the token and ORQ = {1, 6} is further sent to P_6. Node P_6 extracts <0, 5> from the LRQ_6 and sends the token to P_5 along with dummy request <0, 6, 5>. Once again, P_5 becomes the token holder. Following Step 5, the token along with ORQ = {1, 6} is now passed to P_1 while LRQ_5 changes to {<6, 6>}. P_1 enters the CS after removing its node number 1 from the head of ORQ. LRQ_1, at this point is {<0, 5>}. After P_1 leaves the CS, the remaining entry <0, 5> is extracted from LRQ_1 leaving it empty. Once again the token along with ORQ = {6} comes back to P_5. Now, P_5 passes the token along with ORQ = {6} to the node P_6 leaving LRQ_5 empty. Node P_6 enters its CS after removing its own node id from both LRQ_6 and ORQ. The token stays with $P_6 = P_{hold}$ even after the process comes out of the CS and till any other request is generated in the system. The situation has been illustrated in Fig. 4.6. Once again, as it was initially, the neighborhood lists at the nodes are only populated.

Link Failure Resilience

The LFRT algorithm is self-adaptive and resilient against a link failure. Consider the network shown in Fig. 4.7. Let us assume that the link between P_6 and P_5 has failed and node P_5 has just come out of its critical section. The top most nodes in LRQ_5 and ORQ are <4, 6> and ‘4’, respectively. Thus, the path to P_4 from P_5 should begin with a hop to P_6. However, the link between P_6 and P_5, is assumed as failed.

LRQ_5 is revised to {<4, λ>, <1, 1>, <6, λ>} following Step 6 of the LFRT algorithm. Now P_5 would extract the neighborhood information of its own neighbors P_1 and P_3 from the respective neighborhood sets as per Step 7. The 2-tuple in P_5 and the newly extracted data from N_3 to N_1 constitute TN_5. The final

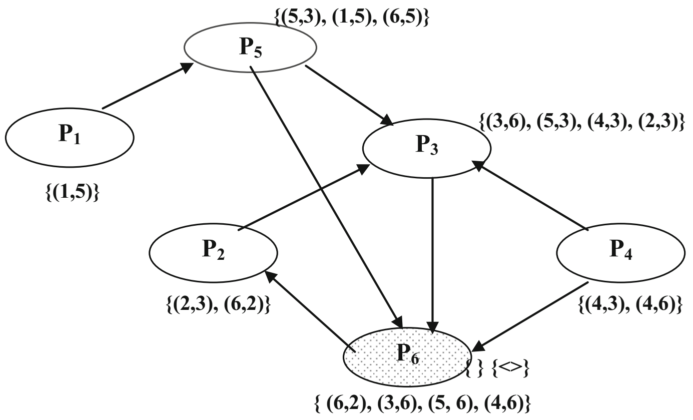


Fig. 4.6 P_4, P_1, and P_6 completed their critical sections

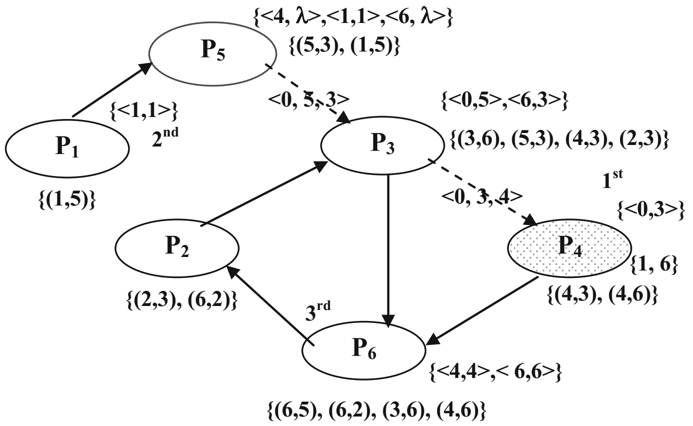


Fig. 4.7 Link between P_6 and P_5 has failed

destination P_4 is one hop away from P_3, and the 2-tuple (4,3) would be a part of TN$_5$. Therefore, following Step 7, the LRQ$_5$ is revised to {<4, 3>, <1, 1>, <6, 3>}. LRQ$_3$ would be {<4, 3>, <6, 3>}. The token along with ORQ is transferred to P_3. It also receives a dummy 3-tuple <0, 5, 3> and updates LRQ$_3$ to {<0, 5>, <6, 3>}. The token along with ORQ and a dummy request <0, 3, 4> is now sent to P_4. Process P_4 updates the ORQ as {1, 6} and LRQ$_4$ as {<0, 3>} before entering the critical section as shown in Fig. 4.6. After P_4 comes out of the critical section, <0, 3> is extracted from LRQ$_4$ leaving it empty. The token is sent back to node P_3, on its way back to P_5. The entire methodology has been explained in Step 6 of the LFRT algorithm (Fig. 4.8).

N_p= Neighborhood set of a node P is an ordered set of 2-tuples N_P = {(T, P) v (P, T)}, where T is a node in the set of neighboring nodes of P. The 2-tuple entry is (T, P) or (P, T), depending on the direction of the link to or away from P respectively.

TN_K =It would store all the link information starting from K. It needs to be created only in case of a link failure on the shortest path from K.

4-Tuple = <A, B, C, A_P> where, A: Id. of the original token requesting node; B: Id. of some intermediate node, C: Id. of the next hop destination node and A_P: Priority of the original token requesting node.

RQ_C = {<A, B, A_P>}. This is maintained locally in every node C. The entry <A, B, A_P> in RQ_C, maintained in C, is for the token request initiated by A with priority A_P that reaches C through B. When a process C receives a token request from another process B, there are three possibilities:

1. If the RQ_C is empty, then the process C enters <A, B, A_P> in RQ_C.

2. If the RQ_C is not empty and priorities of two process are not same, after updates the priorities, the process C checks which priority is higher and enters the higher priority tuple first in RQ_C (i.e. arrange in descending order and assume higher number be the higher priority) and then the other(s).

3. If the RQ_C is not empty and priorities of two processes are same, then process C enters priorities in FCFS.

ORQ ={C}: This is a global list maintained in P_{hold}. The id for each token request initiating node C is stored in ORQ.

count_sent[S]: The count of number of nodes to which a token request from S is forwarded by an intermediate node.

Fig. 4.8 Data structure of LFRT-P algorithm

4.1.2 Limitations of LFRT

LFRT does not consider priority of participating processes. This algorithm redefines fairness and liveness in light of process priorities and proposes new data structure, algorithm, and evaluates properties of another algorithm. The new token-based link failure resilient algorithm (LFRT-P) for ME maintains fairness and considers dynamic process priorities. The underlying topology is directed graph.

4.2 Link Failure Resilient Priority Based Fair ME Algorithm for Distributed Systems (LFRT-P)

The LFRT-P [15] is a token-based algorithm that works for processes with assigned priorities on any directed graph topology with or without cycles. In spite of considering priorities of processes, it ensures liveness in terms of token requests from low priority processes. Moreover, the algorithm keeps control message traffic reasonably low. The present work aims to create a new token-based link failure resilient ME algorithm that maintains fairness. The underlying topology is any directed graph. It also takes into consideration priority of participating processes. This was not in the scope of LFRT [11]. It is assumed without any loss of generalization that only one requesting process exists in every node. The algorithm is implemented for two or more requesting processes in the same node.

4.2.1 Data Structure and Algorithm for LFRT-P

Description for LFRT-P Algorithm

When a process P wants to enter the critical section (CS), it has to get hold of the token first. The algorithm would fetch the request for token to the current P_{hold}, say, R. When a process S wants to enter the critical section (CS) then based on its priority r, a new tuple $<S, S, r>$ is inserted into the local request queue RQ_S. This local request queue is maintained as a sorted sequence in descending order on the priority of requesting processes. A fresh token request $<S, S, X, r>$ is sent to X such that $(S, X) \in N_S$, the neighborhood set of S. When a process $X \neq P_{\text{hold}}$ receives a token request $<S, K, X, r>$, then X identifies position P at which $<S, K, r>$ is to be inserted in RQ_X.
All the priority values for entries RQ_X beyond P would be increased by 1. The tuple $<S, K, r>$ would be inserted as the last entry amongst the tuples with priority value r in the sorted sequence of RQ_X. Subsequently, process X sends token request $<S, X, Y, r>$ to all of its neighbor processes Y, if priority r is higher than that of earlier requests from K to X, if any. The token request would also be sent to the neighbors of X, if this is the first token request from K to X. Total number of requests sent to the neighbors for the source process S would be stored in the intermediate process X. However, the recipient process X does not allow any insertion in RQ_X and bounces back the token request $<S, K, X, r>$ to K in case the out-degree of X is 0 or $S \in \text{RQ}_X$. The format for the bounce message to K would be <bounce, S, K>. This ensures that the process terminates in finite time.

When a process $X = P_{\text{hold}}$ receives a token request $<S, K, X, r>$ it bounces back the request if already another request from the same source S is recorded in RQ_X. Otherwise, insertion of the tuple $<S, K, r>$ is made into RQ_X as described above. Besides, the source id S is entered into the global queue ORQ maintained at P_{hold}. Process K, on receipt of the message <bounce, S, K>, increases count_bounce[S] in K by 1. If this count becomes equal to the total number of requests sent for S from K, then the entry against S is deleted from RQ_K.

On completion of a CS, $T = P_{\text{hold}}$ sends the token along with ORQ <token, K, ORQ> to process K where the first entry of RQ_T is $<S, K, r>$. The tuple $<S, K, r>$ is deleted from RQ_T. If RQ_T is not empty, a dummy token request $<0, T, K, r>$ is sent to K. The newly designated P_{hold} checks the first entry S of ORQ. If $S = K$, then K deletes S from ORQ and also deletes the first entry from RQ_K before entering into the CS. However, if the first entry of the ORQ does not match with the id of the current process, then the token along with ORQ is forwarded to the process

M where the first entry of RQ_K is <*S*, *M*, *r*>. The process continues till the requesting process *S* becomes P_{hold}. The algorithm terminates when both the ORQ and the RQ_s maintained in different processes becomes empty. Token remains with the last process that entered CS.

LFRT-P algorithm is activated only if some link failure is detected during transmission of tokens. When a link failure (*K*, *I*) occurs and process *I* cannot be reached from *K*, The entry *I* in RQ_K is replaced by λ. An alternate path is established from *K* to *I* using the neighboring processes of *K* that still can be reached. Initially, all the neighbor information of process *K* is copied into a temporary data structure TN_K. In next iteration, the neighbor of the processes in TN_K is also included into it. The process terminates when *I* is again found in TN_K. An alternate path may now be established from *K* to *I* using the processes in TN_K. If *C* is the first process on the new path from *K* to *I*, then λ would be replaced in RQ_K by *C*. A pseudocode representation for the LFRT-P algorithm is given below.

Begin

```
1. Step 1: When a process S with priority {r} wants to enter the critical section
   (CS).
2. insert <S, S, r> in $RQ_S$; /* S adds {<S, S, r>} into its local request queue $RQ_S$
   */
3. send token_request <S, S, X, r> to all X such that (S, X) $\in N_S$;
   /* S sends token request message <S, S, X, r> to all X such that (S, X) $\in N_S$,
   the neighbor set of S */
4. Step 2: When a process X $\neq P_{\text{hold}}$ receives a token request <S, K, X, r> from
   another process K,
5. if (out-degree of X == 0) then
6. send <bounce, S, K> to K;
   /* Node X detects that it has earlier received the token request originated by S
   and sends a bounce message back to K */
   /* This prevents that control messages does not suffer from the count to infinity
   problem even in the presence of cycles in the underling directed graph topology
   */
   /* Assume that the edge between K and S is temporarily reversed during the
   bounce message transmission */
7. else
8. increase priority value p for all requests in $RQ_X$ by 1 for all p < r;
9. insert <S, K, r> in $RQ_X$ in the sorted order of descending priority;
   /* While maintaining FCFS order for same priority values of two or more
   entries */
10. if the entry <S, K, r> is inserted at the beginning of the new $RQ_X$ then
11. send token_request <S, X, T, r> to all T;
    /* (X, T) € $N_X$, the neighborhood set of X */
12. count_sent[S] = $|N_X|$ /* $|N_X|$ the cardinality of the neighborhood set of X */
```

```
13. end if
14. end if
15. Step 3: When a process $X = P_{hold}$ receives a token request <S, K, X, r> from
    another process K.
16. increase priority value p for all requests in $RQ_X$ by 1 for all $p < r$;
17. insert <S, K, r> in $RQ_X$ in the sorted order of descending priority;
18. rebuild ORQ by taking the first ids from the triplets of the revised $RQ_X$ for
    $X = P_{hold}$;
19. Step 4: Node K, on receipt of the message <bounce, S, K>
20. count_sent[S] = count_sent [S] − 1;
21. if (count_sent[S] == 0) then
22. delete <S, J, r> from $RQ_S$;
    /* Deletion of <S, J, r> in $RQ_S$ indicates that there is no new route through J for
    the request */
23. end if
24. Step 5: On completing the execution of a CS, $T = P_{hold}$ performs the following:
25. if (ORQ $\neq$ NULL) then
26. remove the first entry <S, K, r> from $RQ_T$;
27. read the first process id {S} from ORQ;
28. send <token, K, ORQ>;
29. end if
30. if ($RQ_T \neq$ NULL) then
31. send token_request <0, T, K, s>, s being priority of process in the head of $RQ_T$;
    /* If reduced $RQ_T$ is not empty then T, sends a dummy token request to new
    $P_{hold}$ */
32. end if
33. Step 6: The newly designated $P_{hold}$ node performs the following as it receives
    <token, K, ORQ> and an optional dummy request <0, T, K, s> from previous
    $P_{hold}$, i.e., T.
34. remove the first entry <S, K, r> from $RQ_K$;
35. if (K receives dummy request <0, T, K, $T_P$> from T) then
36. insert <0, T, $T_P$> in proper place of $RQ_K$;
37. read the first process id {S} from ORQ;
38. if (S == K) then
39. remove the id {S} from ORQ;
40. enter CS;
41. else
42. send <token, ORQ, X>, where X is the second id in the first entry of $RQ_K$;
43. end if
44. end if
45. if ($RQ_K \neq$ NULL) then
46. send token_request <0, K, X, s>, s being the priority of the process in head of
    $RQ_K$;
```

```
47. end if.
48. Step 7: In case of a link failure {K → I} during token passing, an intermediate
    node K holding the token does the following:
49. replace {I} by {λ} in $RQ_K$;
50. delete the 2-tuple ((K, I) ∨ (I, K)) from $N_K$ for node K.
51. $TN_K = N_K$;
52. repeat
53. add {(S, H) ∨ (H, S)} to $TN_K$ ∀ S ∈ $TN_K$ where H is a neighbor of S;
54. until ({(M, S) ∨ (S, M)} ∈ $N_K$)
55. if ({(M, X) ∨ (X, M)} ∈ $TN_K$) then
56. replace {λ} by {X} in $RQ_K$;
57. end if
```

End

Illustrative Example of the Algorithm

In Fig. 4.9, we find a directed graph with six nodes labeled as *A* through *F*. The example under consideration deals with six processes *A*, *B*, *C*, *D*, *E*, and *F* with the priority values 1, 3, 4, 3, 5, and 2, respectively. We further assume that the priority 99 is highest and a higher value indicates that the corresponding process has a greater priority. The neighborhood sets for the six nodes are shown beside each node in the figure.

As shown in Fig. 4.10, the node *D* has placed the first token request with priority 3. The entries in RQ's and the ORQ are also shown. RQ_D stores <*D*, *D*, 3> as par Step 1 of the LFRT-P algorithm. *D* sends its request using the 4-tuple <*D*, *D*, *F*, 3> and <*D*, *D*, *C*, 3> to the neighbor nodes *F* and *C*, respectively. On receipt of a request from *D*, nodes *C* and *F* insert the first two literals and last literal of the 4-tuple, i.e., <*D*, *D*, 3> into RQ_C and RQ_F, respectively. Now *C* sends a request on behalf of *D* using the 4-tuple <*D*, *C*, *F*, 3> to its neighbor *F*.

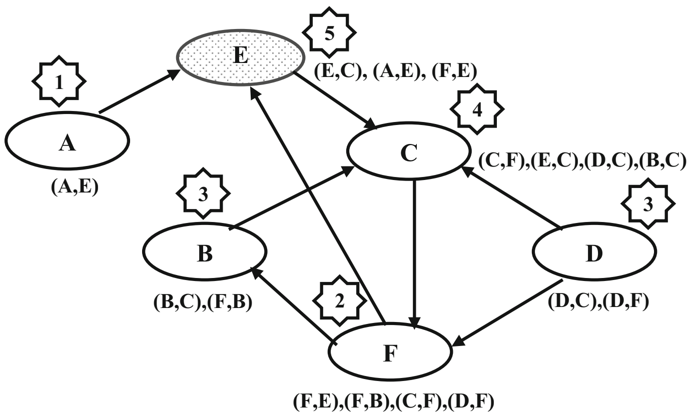


Fig. 4.9 Initial topology—*E* holds the token

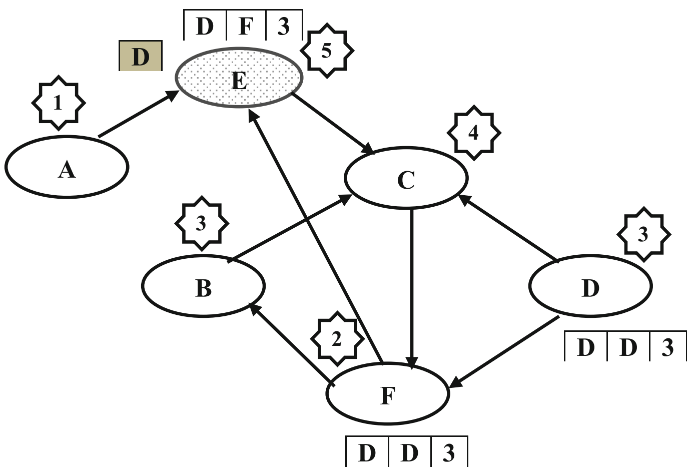


Fig. 4.10 Token request from node *D* reached node *E*

Node *F*, however, already received the same request from *D* directly so the 4-tuple from *C* is to be ignored and following Step 2 of the LFRT-P algorithm, i.e., node *C* detects that it has earlier received the token request originated by *D* and sends a bounce message back to *C*.

Now, *F* sends the request from *D* to its neighbor nodes *B* and *E* using the 4-tuple <*D*, *F*, *B*, 3> and <*D*, *F*, *E*, 3>, respectively following Step 2. Similarly, *C* rejects the request from *B* following Step 2 because *D* € RQ$_C$. *E* = P_{hold}, receives the 4-tuple <*D*, *F*, *E*, 3> and put <*D*, *F*, 3> in RQ$_E$. It also puts the first element of 4-tuple in ORQ, i.e., ORQ = {*D*}. Let us now consider that, *A* issues a second token request with priority 1. So it puts <*A*, *A*, 1> in RQ$_A$ and sends the 4-tuple <*A*, *A*, *E*, 1> on its outgoing edge to *E*. Node *E* receives the 4-tuple and following Step 3 of the LFRT-P algorithm, the new request from *A* must have a lower priority than earlier request. Revise the queue RQ$_E$ in accordance with the decreased priority, updates RQ$_E$ as {<*D*, *F*, 3>, <*A*, *A*, 1>}. The ORQ is updated to ORQ = {*D*, *A*}.

The node *F* now places the third request with priority 2. So RQ$_F$ = <*F*, *F*, 2>. *F* also sends the 4-tuple <*F*, *F*, *E*, 2> and <*F*, *F*, *B*, 2> to the neighbors *E* and *B*, respectively. *E* receives the request and following Step 3 of the algorithm, it modifies RQ$_E$ to <*D*, *F*, 3>, <*A*, *A*, 2>, <*F*, *F*, 2>. Process *E*, first increases the priority of *A* which originated by *A*, by one, then arranges the requests, which updates RQ$_E$ to {<*D*, *F*, 3>, <*A*, *A*, 2>, <*F*, *F*, 2>} and ORQ = {*D*, *A*, *F*}. Node *B* sends 4-tuple <*F*, *B*, *C*, 2> to *C* and node *C* has no other tuple, but <*F*, *C*, *F*, 2> to be sent to *F* only. This is prohibited by Step 2. Node *C* detects that it has earlier received the token request originated by *F* and sends a bounce message back to *B*. The bounce process as mentioned in Step 4 ensures that the local request queues of *B* and *C* are restored as there is no path to P_{hold} from *F* through *B*. Node *E* updates the RQ$_E$ and ORQ to {<*D*, *F*, 3>, <*A*, *A*, 2>, <*F*, *F*, 2>} and {*D*, *A*, *F*} respectively on receipt of the 4-tuple <*F*, *F*, *E*, 2> directly from *F*. The situation is

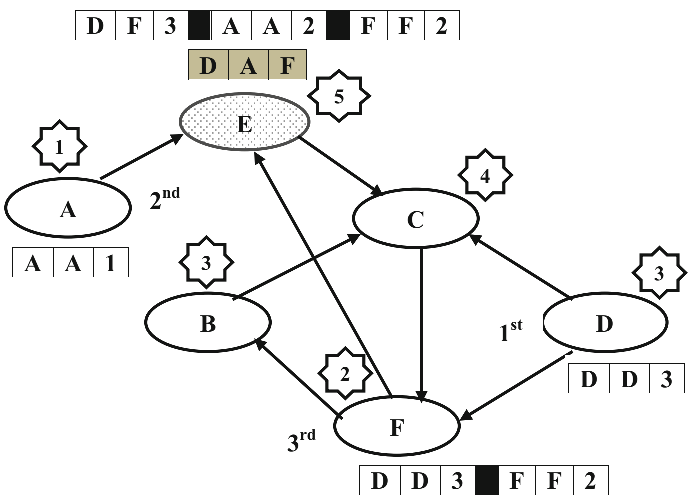


Fig. 4.11 Token requests from node *D*, *A* and *F* reached *E*

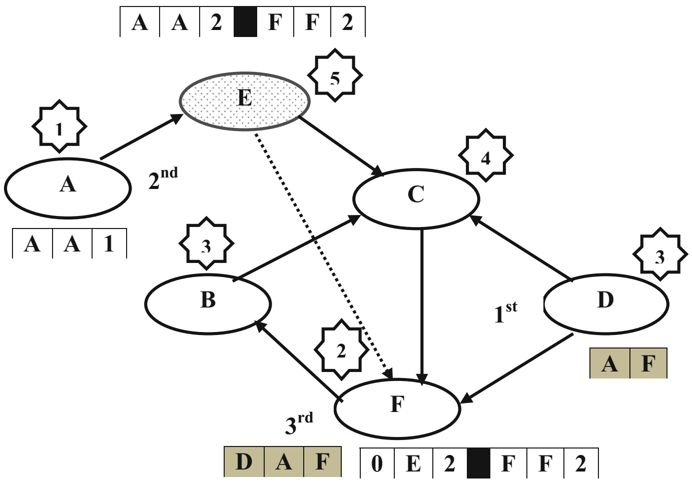


Fig. 4.12 Node *F* holds the token

illustrated in Fig. 4.11 after all three token requests from *D*, *A*, and *F* are registered with the token holding node *E*. Say, *E* comes out of its critical section now. The node will pass the token and ORQ to the node from which it received the request first following procedure of Step 5. It read the first entry '*D*' from ORQ and removes the first entry <*D*, *F*, 3> from RQ_E following Step 5. Node *E* has to pass the token to *F* (Fig. 4.12).

Following Step 6, the directed edge from *E* to *F* is reversed and the token along with the ORQ is passed. Since the RQ_E is not yet empty, a dummy request <0, *E*, *F*, 2> is sent to *F*. It removes the first entry <*D*, *D*, 3> from RQF and adds <0, *E*, 2> at the beginning of RQF, thus updating it to {<0, *E*, 2>, <*F*, *F*, 2>}.

Since priorities of both the processes are same, the dummy request is preferred. The condition as specified in Step 6 matches and therefore *D* enters the CS after updating the ORQ to {*A*, *F*}. Thereafter, *D* comes out of the CS.

The RQD is now empty and following Step 6, the token and ORQ = {*A*, *F*} is further sent to *F*. Node F becomes the token holder. It removes the first entry <0, *E*, 2> from RQF and sends the token to *E* along with dummy request <0, *F*, *E*, 2>. Since the RQF is not yet empty, a dummy request <0, *F*, *E*, 2> is sent to *E*. *E* further passes the token to *A*.

The condition as specified in Step 6 matches and therefore, node *A* enters its CS after removing its own node id from both RQ_A and ORQ. Thereafter, *A* comes out of the CS. The RQ_A is now empty. Thus *A* cannot send any dummy request. The token and ORQ = {*F*} is further sent to *E*. Now, *E* passes the token along with ORQ = {*F*} to the node *F* leaving RQ_E empty. The node F becomes the token holder. It removes the first entry <*F*, *F*, 2> from RQ_F. The token stays with $F = P_{\text{hold}}$ even after the process comes out of the CS and until any other request is generated in the system. The situation has been illustrated in Fig. 4.13. Once again, as was initially, the neighborhood lists at the nodes are only populated.

Illustration Involving Link Failures

The LFRT-P algorithm can also deal with link failures. The steps have been discussed in Step 7. Consider the network shown in Fig. 4.14. Let us assume that the link between nodes *F* and E has failed and node *E* has just come out of its critical section. The second tuple in RQ_E and ORQ are <*F*, *F*, 2> and '*F*', respectively. However, the link between *F* and *E*, as assumed here, has failed. The RQ_F is revised to {<*D*, *C*, 3>, <*A*, *A*, 2>, <*F*, *C*, 2>} following Step 7 of the LFRT-P algorithm. Now *E* would extract the neighborhood information of its own neighbors *A* and *C* from the respective neighborhood sets as per Step 7.

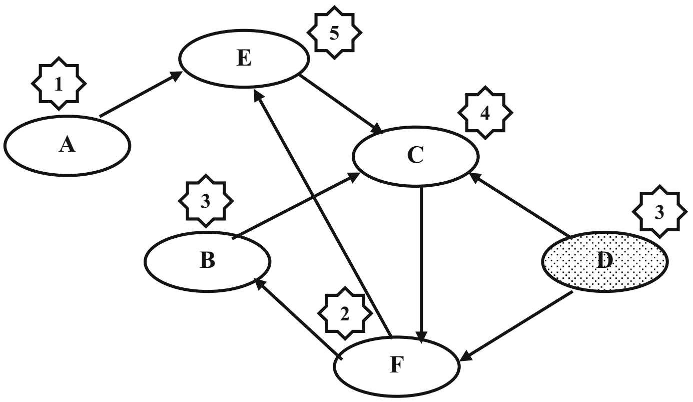


Fig. 4.13 Nodes *D*, *A* and *F* completed their critical section

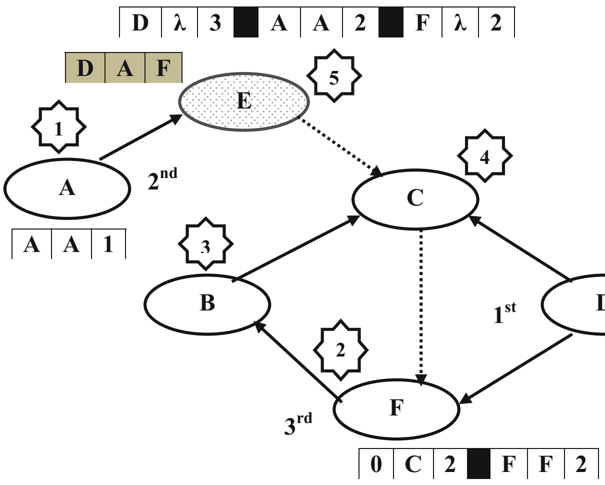


Fig. 4.14 Handling link failure

The 2-tuple in E and this newly extracted information from N_C to N_A constitute TN_E. The final destination to F is one hop away from C, and the 2-tuple (D, C) would be a part of TN_E. Therefore, following Step 7, the RQ_F is revised to {<D, C, 3>, <A, A, 2>, <F, C, 2>}. RQ_C would be {<D, C, 3>, <F, C, 2>}. The token along with ORQ is transferred to F. Node F updates the ORQ as {D, A, F} and RQ_F as {<0, C, 2>, <F, F, 2>} before entering the critical section as shown in Fig. 4.14. After D comes out of the critical section, the token is sent back to node C, on its way back to E. In another example, the link failure may be handled by finding an alternate path to the desired intermediate node F, in this case. The methodology has been explained in Step 7 of the LFRT-P algorithm.

4.2.2 *Performance Analysis for LFRT and LFRT-P*

We evaluate these algorithms from multiple perspectives. We consider correctness, message complexity, fairness, and other important factors of distributed control algorithms and evaluate the solution from these aspects.

Low Storage

Using LFRT and LFRT-P, very little data is stored at the participating nodes. A node need not even know the total number of nodes in the system or the id of the token holding node to place its request for the token. When a token request occurs, then some additional data is stored in some of the nodes. This additional data reduces to NULL after the token is delivered to the requesting node. The neighborhood list, as designated by N_i in Fig. 4.8, is the only permanent data structure stored in any node i.

Safeness and Fairness

A mutual exclusion algorithm satisfies the safeness specification of the mutual exclusion problem if it provides mutually exclusive access to the critical section. One of the concerns with many of the mutual exclusion algorithms is fairness. The commonly accepted definition of fairness in the context of mutual exclusion is that requests for access to the CS are satisfied in the order of their occurrences. The revised definition of priority-based fairness as introduced in Chap. 2, implies that the token must be allocated to some process A such that among all the processes having priority equal or lower than that of A, the token request from A has reached P_{hold} ahead of others and there is no pending token request from any other process B having a priority higher than that of A.

Lemma 4.1 (Safeness) *The LFRT-P token-based algorithm provides safe mutual exclusion as defined in* Sect. 2.6.

Proof The LFRT-P algorithm, being a token-based solution maintains mutual exclusion by controlling the number of tokens. It is assumed that only one token will be there for each instance of a sharable resource. Any participating process in the system needs to get hold of the token before entering the critical section. Obviously, no two processes can hold the token simultaneously. Hence, the property of safety is ensured. Thus the statement of Lemma 4.1 is correct.

Lemma 4.2 (Priority based Fairness) *The LFRT-P token-based algorithm provides fair and safe mutual exclusion as defined in* Sect. 3.3.2.

Proof Without any loss of generalization, let A be the first request and B be the second request among all requests ever made until now with equal priority. With this assumption, the proof follows from Lemma 4.2 in [16].

There could be a second case where A may be the first request with a lower priority and B be the second request with higher priority. In such cases, the request A will be after the request B in the RQ of P_{hold} after executing Step 3. Accordingly the ORQ will also change. Therefore, process A will get the token only after process B completes its execution in CS. Thus, the statement of Lemma 4.2 holds good for both of the possible cases.

Liveness

Liveness of a mutual exclusion algorithm is said to be maintained, if every process that wants to enter its critical section is eventually allowed to do so.

Lemma 4.3 (Liveness) *The LFRT-P algorithm achieves liveness.*

Proof Let A be the first request among all requests ever registered with the P_{hold} and B be the last among all the requests until now. Let us also assume that A and B have the same priority. In [16], it has been shown that A is served before B. The liveness is satisfied in such cases.

Let request A has higher priority than B while request B for token is made ahead of A. In order to prove liveness, we have to show that the request B would eventually be served. In Step 2 of our algorithm, the priority of B is increased by 1 each time, a request from a higher priority process is logged before the position of B in the local RQ of any node on the path from the requesting node to P_{hold}. This ensures that after a finite number of steps, the priority of B will eventually increase to the maximum priority of M. However, for any other request C also with priority M, such that C arrives after B, the latter request will be served after B only. This is by the first part of the proof. Thus, the request B is served eventually.

Correctness
The correctness of control algorithms is typically defined as a collection of safeness and liveness. Both of these properties have been individually proved in Sect. 4.2.2. Hence, the correctness of the LFRT-P algorithm is also proved.

Handling Link Failure and Cycles
One of the most significant advantages of the LFRT-P algorithm is its ability to handle the link failures. In a wireless environment, existing links are dropped frequently. The LFRT-P algorithm is able to find an alternate path as par algorithm, in case of a link failure. There are many existing mutual exclusion algorithms that work on a directed, acyclic graph (DAG). The LFRT-P algorithm works on any directed graph, with or without cycles. Steps of the LFRT-P algorithm ensure that the token request messages do not form cycles. This keeps the message complexity low.

Message Complexity
The number of messages per critical section access can be deterministically expressed as a measure of concurrency of requests. A site X that wants to enter the critical section sends 3-tuple through each outgoing link from it. If k be the length of the shortest directed path from X to $P_{\text{hold},}$ and the number of outgoing links from an intermediate node K be L_k, then the total number of 3-tuple sent would be $\sum L_i$, for all $i \in [1 \ldots k]$. The number of hops that the token on the return path takes is exactly $k - 1$, as the path is predetermined and stored using the local request queues.

Simulation Results
We present the simulation results for the LFRT-P algorithm [15] and the tree-based FAPP [16] solution. We have conducted multiple sets of simulations on different networks by choosing different P_{hold} nodes from a connected graph of 22 nodes. The requesting nodes and the order of requests are also selected at random for different sizes of request queues. The average for each selection is plotted. The simulation is done using MATLAB 7.7.0. The simulation parameters are listed in Table 4.1. A confidence interval analysis has been done to check the randomness of the selected data set.

The results that have been plotted in different figures represent average of multiple executions. The different orders of requests are taken absolutely at random. Besides, these are executed for different sizes of request sets varying from 4 to 12. Thus the analysis based on these randomly selected executions may be considered unbiased and represents the generic behavior of the system for the LFRT-P algorithm. Statistical analysis by computing the confidence intervals, as presented

Table 4.1 Simulation parameters

Parameters	Value
Connection topology	Directed graph with cyclic or without cyclic
Nodes in the graph	22
Edge length	Static
Maximum out-degree	2
Maximum degree	3
Minimum degree	2
Priority of nodes in the graph	1…10 (user choice)
Maximum number of token requests	Gradually increased (4, 6, 8, 10, 12)

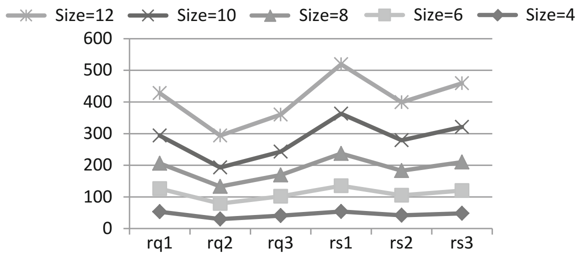


Fig. 4.15 Control message complexity for request set size of 4, 6, 8, 10, and 12

below, confirms the randomness of selection for the sequence of requests. In Fig. 4.15, five different series are plotted for 4, 6, 8, 10, and 12 requesting nodes, respectively. Plots against rq1, rq2, and rq3 show the average numbers of control messages exchanged for three different P_{hold} nodes. The respective response data plots are marked with rs1, rs2, and rs3. The y-axis representing the number of control messages has been scaled up nearly 333% for the sake of improved relative comparison between the nearly parallel series.

The plots for the six different sets in each series in Fig. 4.15 reflect the variation in performance for arbitrarily built networks and requesting nodes. As the number of request for CS is increased, in all of the cases the number of control messages also increased. The small number of messages exchanged is an indirect proof for the stability of the LFRT-P algorithm for different topologies. The logic of experimentations and the interpretation of the observations are given below. A particular node is selected as P_{hold} for each set. This is used with four different set of requests for size = 4 in the format P_{hold} (First request, Second request, Third request, Fourth request) e.g., $P_1(P_3, P_7, P_{11}, P_{21})$, $P_1(P_{12}, P_9, P_4, P_{20})$, $P_1(P_7, P_{22}, P_2, P_{10})$, etc. The average of the number of control messages for rq1 are computed from this for size = 4 and plotted on the graph. Similarly, points have been plotted for other sizes like 6, 8, 10, and 12. The same method of experimentation has been followed for rq2 and rq3, but with different nodes as P_{hold}.

Confidence Interval and Randomness of the Simulation

We consider rq1(X) and rs1(Y) as the performance evaluation parameters. Now we have to test the null hypothesis H_0:$\mu X = \mu Y$ against all H$_1$:$\mu X > \mu Y$ or H$_1$:$\mu X < \mu Y$. If we put $Z_i = X_i - Y_i Z' = \sum X/N$, $S'^2_Z = (1/(N-1)) * \sum_i (Z_i - Z')^2 = (1/(N-1)) * \left[\sum Z_i^2 - N * Z'^2\right]$, $S_Z' = \sqrt{S'^2_Z}$ and observed value of the statistic is $t = (\sqrt{N} * Z')/S'_Z = -2.84$. Here, value of N is 5.

The parameters for computation of hypothesis testing have been taken as (X = rq1, Y = rs1), (X = rq2, Y = rs2) and (X = rq3, Y = rs3), respectively. Table 4.2 depicts the computation of validity of the hypothesis for the first set. Here the value for τ (tabulated) is $\tau_{0.05,4} = 2.132$. In the Table 4.2, computed |τ (observed)| is equal to $|-2.84| = 2.84$. In all three cases, it has been observed from the τ-distribution that τ (observed) > τ (tabulated) which is highly significant at 5% level of significance. Hence, we reject the null hypothesis H_0 and conclude that there is a significant change for control message complexity for response set against the request set.

The average number of control messages for a particular set depends on the degree of P_{hold}. As for example, if the P_{hold} has a high degree, then it is found that the number of control messages involved is relatively low as compared to a P_{hold} of low degree for the same request size. This is why in Fig. 4.15, the control message for request set1 (rq1) is always higher than the request set2 (rq2) and the request set3 (rq3) for a particular request size. It is also observed that the number of control messages for request and response depends on the size of request set. The control message increases with increase in the request set size and vice versa.

Next, we would look at the randomness of data sets that have been selected for the plotted results. The series for rq1 as plotted in Fig. 4.15 shows the number of control messages for different request sets with node P_6 as P_{hold}.

As for example, in the original randomly selected set for requesting nodes for request size = 4, consists of P_6(P_{14}, P_{18}, P_8, P_{20}). The total count of control messages for these requests are 53, as recorded in Table 4.3. In the next three columns for request set size of 4, the number of control messages for three alternate sets of requesting nodes with the same $P_{\text{hold}} = P_6$ is listed.

The instances of requesting nodes are P_6(P_2, P_4, P_8, P_{16}), P_6(P_2, P_7, P_{12}, P_{17}) and P_6(P_5, P_{10}, P_{15}, P_{20}) and the total number of control messages exchanged are 30, 42, and 36, respectively. Similarly, alternate data sets are considered for request size 6, 8, 10, and 12. The observations are recorded in subsequent rows in Table 4.3. The τ (tabulated) value for the tests of significance of 95 and 99% are, $\tau_{0.05,4} = 2.132$ and $\tau_{0.01,4} = 3.747$ respectively. Since |τ (observed)| < τ (tabulated) for all three alternate sets considered are less than the corresponding table values for both 1 and 5% levels, it may be concluded that the null hypothesis is accepted both at 1 and 5% levels and we say that it is not significant. So, we have conducted the remaining part of the simulation using the column for original set of Table 4.3. Figures 4.16 and 4.17 represent the plots for control messages for request and response sets, respectively. It is observed that the number of control messages increases for increased size of request sets (Fig. 4.16) and response sets (Fig. 4.17).

Table 4.2 Computing tests of significance

Request set	rq1 (X)	rs1 (Y)	$Z = X - Y$	$Z^2 = Z * Z$	$Z' = \sum X/N$	$S'^2_Z = (1/(N-1)) \sum_i (Z_i - Z')^2$	$S'_Z = \sqrt{S'^2_Z}$	$t = (\sqrt{N} * Z')/S'_Z$
4	53	54	−1	1	−18.2	205.2	14.33	−2.84
6	73	81	−8	64				
8	80	102	−22	484				
10	88	126	−38	1444				
12	134	156	−22	484				

Table 4.3 Analysis for alternate data set

Request set size	Original set with $P_6 = P_{\text{hold}}$	Alternate set1 with $P_6 = P_{\text{hold}}$	Alternate set2 with $P_6 = P_{\text{hold}}$	Alternate set3 with $P_6 = P_{\text{hold}}$	τ (observed) for original versus set1	τ (observed) for original versus set2	τ (observed) for original versus set3
4	53	30	42	36	−1.176	−1.134	−0.566
6	73	72	63	63			
8	80	100	97	90			
10	88	135	112	119			
12	134	162	161	144			

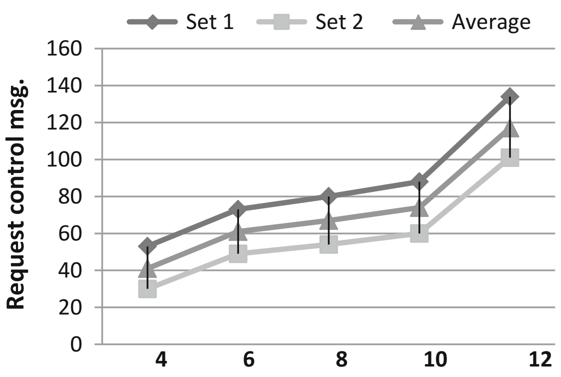


Fig. 4.16 Request control message complexity against the size of request sets

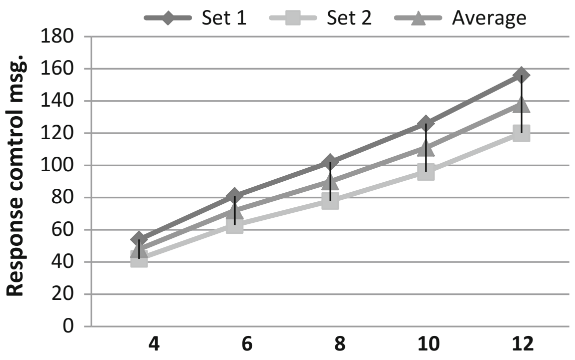


Fig. 4.17 Response control message complexity against the size of request sets

However, as seen in Figs. 4.16 and 4.17 the growth in number of control messages is somewhat linearly proportional to the growth in the size of the network. Thus the LFRT-P algorithm is highly scalable.

Computation of tests of significance is done for set1 and set2 for both request and response message computations. The comparison in Fig. 4.16 is between the two sets of request control messages for two different P_{hold} nodes. On the contrary, in Fig. 4.16, the comparison is between the two corresponding sets of response control messages. As the same 95% confidence interval is selected, τ (tabulated) is same as $\tau_{0.05,N-1} = 2.132$. The τ-distribution results have been recorded Table 4.4. The τ-distribution results have been recorded Table 4.4.

Table 4.4 Tests of significance for alternate data set

Sl.	Description	τ (observed)	τ (tabulated)
1	Request set (Fig. 4.16)	15.12	2.132
2	Response set (Fig. 4.17)	5.66	2.132

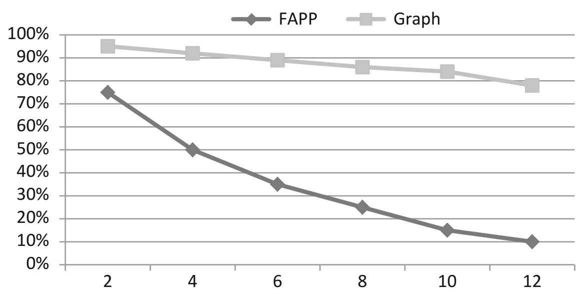


Fig. 4.18 Success rate for the LFRT-P algorithm versus FAPP in presence of link failures

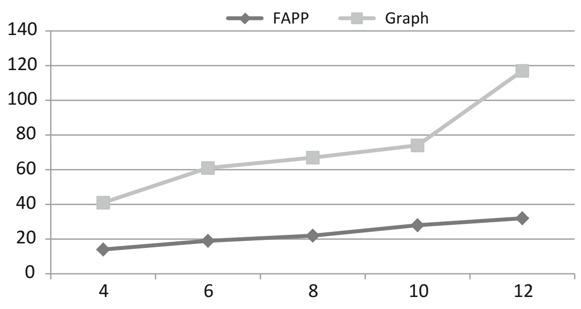


Fig. 4.19 Number of control message of the LFRT-P algorithm versus FAPP

In both cases, it is found that $|\tau \text{ (observed)}| > \tau$ (tabulated). Hence, the null hypothesis, H_0 is rejected at 5% level. Thus, it may be concluded that set2 values are better than set1 values for both request and response diagrams. The control message complexity depends on the degree of P_{hold}. In this case, degree of the root node is lowers for set2 as compared to set1.

In Fig. 4.18, the advantage of the LFRT-P algorithm has been demonstrated over FAPP [16]. In the LFRT-P graph based algorithm, a node has one or more paths toward P_{hold}. Thus if a link is broken, an alternate path may still be used to pass the token request or to receive the token from P_{hold}. FAPP, being a tree-based solution, fails to connect when an edge brakes on the unique path to P_{hold} from any requesting node.

The simulation result as shown in Fig. 4.18 confirms the same. However, the LFRT-P graph based solution cannot guarantee 100% link failure resilience. The LFRT-P algorithm fails in case removal of one or multiple links leaves the requesting node and P_{hold} in two disjoint graph partitions.

In Fig. 4.19, a comparative performance of the LFRT-P algorithm with FAPP is presented in terms of the number of response control messages exchanged for different set of request. The response control message is the control message that grants permission to a process to enter its CS.

FAPP is expected to offer a lower message complexity as it is a tree-based implementation as compared to the directed graph topology for the LFRT-P

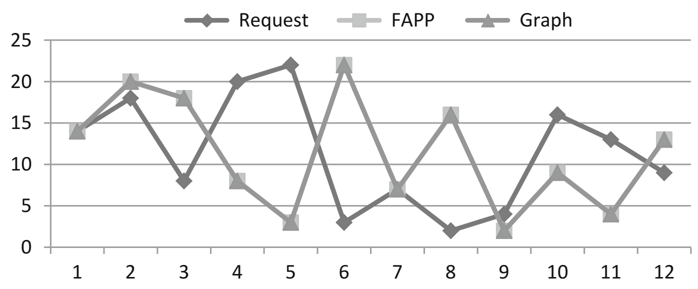
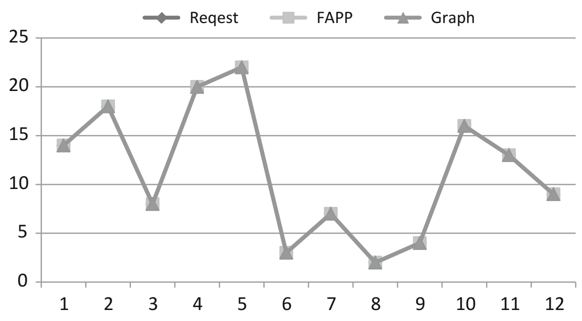


Fig. 4.20 Order of execution for the LFRT-P versus FAPP

Fig. 4.21 Order of execution for the LFRT-P algorithm versus FAPP for equal priority processes



algorithm. However, the LFRT-P algorithm also performs reasonably well here in terms of message complexity.

This may be primarily attributed to the process that a token request is forwarded in the LFRT-P algorithm by an intermediate node, only if no higher priority request is pending.

In Fig. 4.20, a comparative performance of the LFRT-P algorithm is done against FAPP [16] in terms of the number of request ordering and their execution ordering among 12 request set. Since the node request depends on priority, in dynamic priority changes in the FAPP and LFRT-P algorithm and depends on number of hop changes to reach to P_{hold}. The simulation results confirm that the LFRT-P algorithm satisfies the properties of liveness, fairness, safety, and stability.

In Fig. 4.21, a comparative performance of the LFRT-P algorithm versus FAPP for request ordering and their execution ordering among 12 requests set when priorities are same. As expected, the LFRT-P algorithm follows FCFS. In both the cases, it maintained fairness. The simulation results confirm that the LFRT-P algorithm satisfies the properties of liveness, fairness, safety, and stability.

4.3 Concluding Remarks on Graph-Based ME Algorithms

The algorithm considers a network in the form of a connected, directed graph, with or without cycles. The beauty of LFRT [11] algorithm is that it requires very little storage space for each process to start with. The token-based LFRT algorithm handles the node mobility issue by providing a failed link tolerant token request and release mechanism. Besides a very simple and minimal data structure is required for the implementation of the algorithm. LFRT has its own mechanism to avoid formation of cycles when token request is forwarded. This makes possible that the solution works on any directed graph topology with or without cycles. It also helps keeping the turnaround time and the number of control messages low. LFRT algorithm maintains safeness and liveness, the two constituent elements of correctness. LFRT-P [15] algorithm for ME holds fairness while considering a dynamic process priority. One major limitation of the algorithm is that it is not serializable. The algorithm could be improved further by making it serializable so that it could be deployed faster by splitting the parallel components and scheduling those to different processors in the system.

References

1. Swaroop, A., Singh, A.K.: A Distributed group mutual exclusion algorithm for soft real-time systems. In: Proceedings of World Academy of Science, Engineering and Technology, vol. 26, pp. 138–143 (2007)
2. Lodha, S., Kshemkalyani, A.: A fair distributed mutual exclusion algorithm. IEEE Trans. Parallel. Distrib. Syst. **11**(6), 537–549 (2000)
3. Madria, S.K.: Timestamp based approach for the detection and resolution of mutual conflicts in real-time distributed systems. Computer Science Technical Reports. Paper 1367, pp. 1–16 (1997)
4. Paris, J.-F., Darrell, D.E.: Long, efficient dynamic voting algorithms. In: Proceedings of the Fourth International Conference on Data Engineering, pp. 268–275 (1998)
5. Maekawa, M.: A $\sqrt{n}$ algorithm for mutual exclusion in decentralized systems. ACM Trans. Comput. Syst. **3**(2), 145–159 (1985)
6. Chaki, N., Chaki, R., Saha, B., Chattopadhyay, T.: A new logical topology based on barrel shifter network over an all optical network. In: Proceedings of 28th IEEE International Conference on Local Computer Networks (LCN'03), pp. 283–284 (2003)
7. Walter, J.E., Welch, J.L., Vaidya, M.H.: Mutual exclusion algorithm for Ad hoc mobile networks. Wirel. Network **7**(6), 585–600 (2001)
8. Sil, S., Das, S.: An energy efficient algorithm for distributed mutual exclusion in mobile Ad-hoc networks. World Acad. Sci. Eng. Technol. **64**, 517–522 (2010)
9. Zarafshan, F., Karimi, A., Al-Haddad, S.A.R., Saripan, M.I., Subramaniam, S.: A preliminary study on ancestral voting algorithm for availability improvement of mutual exclusion in partitioned distributed systems. In: Proceedings of International Conference on Computers and Computing (ICCC'11), pp. 61–69 (2011)
10. Agrawal, D., EL Abbadi, A.: An efficient solution to the distributed mutual exclusion problem. In: Proceeding of the 8th ACM Symposium on Principles of Distributed Computing, pp. 193–200 (1989)

11. Kanrar, S., Choudhury, S., Chaki, N.: A link-failure resilient token based mutual exclusion algorithm for directed graph topology. In: Proceedings of the 7th International Symposium on Parallel and Distributed Computing—ISPDC 2008 (2008)
12. Chen, W., Lin, S., Lian, Q., Zhang, Z.: Sigma: A fault-tolerant mutual exclusion algorithm in dynamic distributed systems subject to process crashes and memory losses. Microsoft Research Technical Report MSR-TR-2005-58 (2005)
13. Delporte-Gallet, C., Fauconnier, H., Guerraoui, R., Kouznetsov, P.: Mutual exclusion in asynchronous systems with failure detectors. Technical Report in Computer and Communication Sciences, id: 200227 (2002)
14. Lin, S., Lian, Q., Chen M., Zhang, Z.: A practical distributed mutual exclusion protocol in dynamic peer-to-peer systems. In: Proceeding of IPTPS'2004 (2004)
15. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new link failure resilient priority based fair mutual exclusion algorithm for distributed systems. J. Network. Syst. Manage. (JONS) 21(1), 1–24 (2013). ISSN 1064-7570
16. Kanrar, S., Chaki, N.: FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems. Special issue on recent advances in network and parallel computing. Int. J. Networks 5(1), 11–18 (2010). ISSN: 1796-2056

Chapter 5
Voting-Based Mutual Exclusion Algorithms

Concurrency control [1, 2] for a distributed system is always quite challenging and is getting even more complex with the increasing sophistication of such systems. Voting is one of the relatively simpler techniques and does not bear high overhead. A vote to the candidate process, say C, from another process, say A, is to grant permission against CS request of C. The existing literature points that there exist various voting approaches to select one candidate process from many to allow its access to the critical section (CS). A symmetric algorithm proposed by Ricart and Agrawala [3] for this problem, requires $2(N - 1)$ messages per CS entry. In such a classical application of voting, a process can enter into its CS only after receiving votes from all the other processes. Ricart-Agrawala algorithm fails when one or more of the reply messages are not delivered for some reason.

An alternate approach could be selection by majority voting. Let's consider a scenario where there are three (3) different candidates for CS and each of these get approximately 30–35% of the votes. In such a scenario, none of these would gain majority and hence no process is allowed to enter the CS!

5.1 Below-Majority Voting for ME in Distributed Systems (BMaV)

In the traditional sense, voting is used to find a candidate with majority votes in its favour. However, there's no guarantee that such will be the case. We introduced a voting-based mutual exclusion algorithm, called below-majority voting (BMaV) algorithm [4] that finds a candidate for CS when majority consensus is not achieved and even when the network is partitioned. The proposed approach is robust enough to find a solution where a network with N processes is split into N partitions leaving only a single process in each partition. The BMaV algorithm considers the occupancies of resources by different processes toward finding the candidate node in a distributed system. The BMaV algorithm, described later in this chapter, combines voting with resource allocation information. This leads to a quicker completion of processes holding more number of resources.

Assumptions

The following set of assumptions form the foundation for the proposed dynamic mutual exclusion algorithm for distributed environment.

- Initially all processes and links are non-faulty in a singular distinguished partition.
- It is assumed that failures do not occur during voting. However, the processes or links may fail before processing the update request in a given site.
- No new process is added during voting.
- A message that is sent eventually reaches the desired destination.
- A process that receives a request will either accept or reject the request.
- On receiving a request, a process A sends its permission to the requesting process B only if it has not already granted permission to some other process, say X. Only after receiving a RELEASE message from X, process A grants permission to B.
- Each process communicates with others unless it has failed. No response after a predetermined deadline is considered as a process failure.
- All requesting processes store time stamps for their own requests and forward the same to other processes.

5.1.1 Description for BMaV

A process wishing to enter CS votes itself and requests the remaining $N - 1$ processes for their votes and waits for the responses. If any process P_i gets majority voting, then P_i is allowed to enter its CS. Otherwise, a pre-defined non-negative integer threshold, say τ, is assumed for execution of the rest of the algorithm. All the processes that obtain votes greater than τ are taken in a set, say S. In the next phase, any process $A \in S$, that holds the maximal number of resources is allowed to enter its critical section. In the event that there is a tie between two or more processes holding the same maximal number of resources, the process that has requested earlier would be allowed access to CS. Subsequently, all these resources are released and once again made available for other processes (Fig. 5.1).

Procedure BMaV

Begin

Step 1 A process wishing to enter CS votes itself and sends request along with time-stamp to the remaining $N - 1$ processes for vote and waits for the responses;

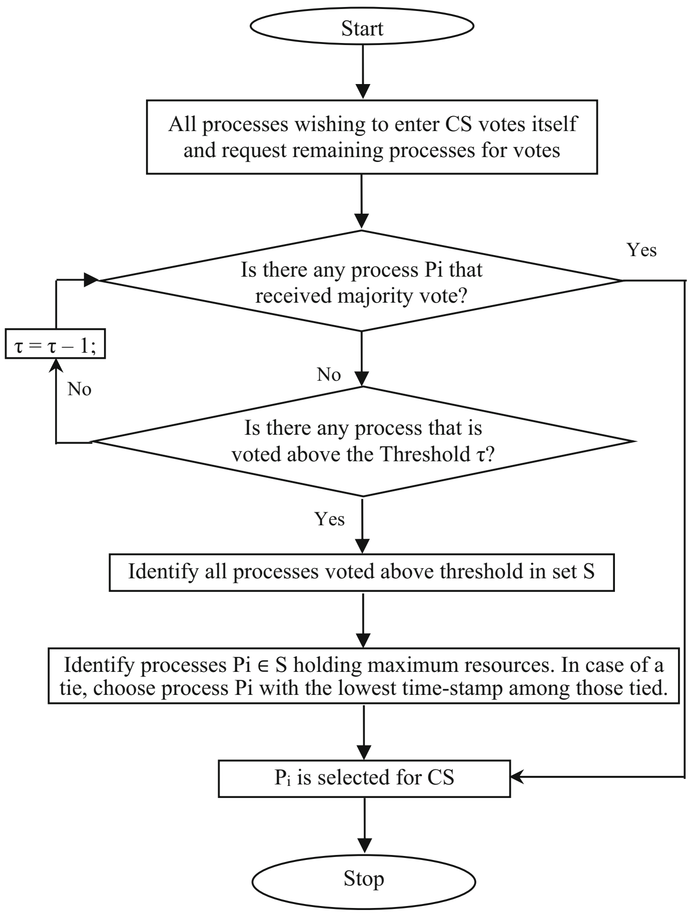
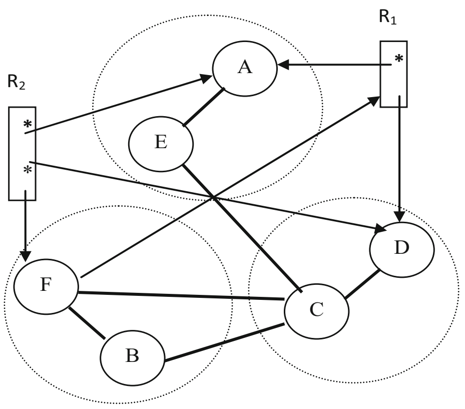

Start
All processes wishing to enter CS votes itself and request remaining processes for votes
Is there any process Pi that received majority vote?
Yes
No
τ = τ – 1;
No
Is there any process that is voted above the Threshold τ?
Yes
Identify all processes voted above threshold in set S
Identify processes Pi ∈ S holding maximum resources. In case of a tie, choose process Pi with the lowest time-stamp among those tied.
Pi is selected for CS
Stop

Fig. 5.1 Flowchart of BMaV algorithm

```
Step 2  If any process $P_i$ gets majority voting, then go to Step 6 else Step 3;
Step 3  If there exists any process voted above a pre-set threshold $\tau$ then Step 4
        else reduce threshold $\tau$ by 1 and return to Step 1 to allow all processes to
        put fresh vote requests;
Step 4  Identify all processes voted above threshold $\tau$ in set $S$;
Step 5  Identify processes $P_i \in S$ that is holding maximum resources. In case of
        a tie, choose the process $P_i$ with the lowest time stamp and tied with
        other process(es) holding the maximum number of resources;
Step 6  Select $P_i$ for entry into its CS;
```

End

Fig. 5.2 Case study for the BMaV algorithm



Illustrating the BMaV Algorithm

In Fig. 5.2, we have an example with which we will study the BMaV algorithm. There are six arbitrary processes *A*, *B*, *C*, *D*, *E*, and *F*. The use case as illustrated here with six possible processes (analogous to nodes) are not based on any simulation result and just to illustrate the functionality and usefulness of the BMaV algorithm.

Let's further consider that three of these processes, say *D*, *A* and *F*, are to enter the respective CS. As in Step 1 of BMaV, each of these three candidates sends time-stamped request messages to other five processes and waits till it receives votes from other nodes. We also assume time-stamp (*D*) < time-stamp (*A*) < time-stamp (*F*). All requesting processes store their own time stamps as well. In Fig. 5.2, there are two resources R_1 and R_2. Resource R_1 consists of two instances and R_2 consists of three instances. We also assume that the two instances for resource R_1 are currently held by process *A* and process *D* while process *F* requested for resource R_1. Resource R_2 is held by processes *A*, *D*, and *F* respectively. We also set threshold value τ as 1. There are three requests for CS and we assume that at least one of the requesting processes gets two votes. There could be four different scenarios for this given condition and let us study how the BMaV algorithm behaves for each of these.

Scenario 1: Assume process *A* gets 4 votes (majority).

A is to be selected for entry into its CS and terminate.

Scenario 2: Assume the network is divided into three partitions as shown in Fig. 5.2. We also assume that two processes *A* and *F* get two votes and process *D* gets only one vote.

Process *A* and *F* are voted above threshold τ as defined in Step 3. Thus, the set *S* as mentioned in Step 4 of the BMaV algorithm would be $S = \{A, F\}$. Using Step 5, process $A \in S$ is found to be holding maximum resources (i.e., 2). The algorithm would elect process *A* for entry into its CS.

Scenario 3: Assume the network is divided into three partitions as shown in Fig. 5.2**. We also assume that 3 processes *A*, *D* and *F* get two votes each.**

Here, all three requesting processes gets vote above threshold. Thus the set S as mentioned in Step 4 of the BMaV algorithm would be $S = \{A, D, F\}$. In the next step, it is found that processes A and D each hold two resources, while F holds only 1. This is a case of a tie between A and D. The algorithm would elect the process with a lower time-stamp (i.e., process D) for entry into its CS.

Scenario 4: Assume the network is divided into three partitions as shown in Fig. 5.2**. We also assume that each processes *A*, *D*, and *F* get only one vote individually.**

Since threshold value is one and none of the processes have got enough votes to cross the threshold, S is NULL. Reduce the threshold τ by 1, i.e., set $\tau = 0$. All processes need to put fresh vote request.

5.1.2 *Comments on the BMaV Algorithm*

The BMaV algorithm identifies a process $P_i \in S$ that has not only obtained votes more than the set threshold, but also holds the maximal number of resources while ensuring a FCFS fairness among multiple such candidates. The algorithm is compatible with a simple clock model like Lamport's logical clock model. However, even then, it is not quite easy to store such information on all the processes in a distributed system with multiple sites.

5.2 A New Hybrid Mutual Exclusion Algorithm in Absence of Majority Consensus (NHME-AMC)

Voting based process synchronization approaches are gaining importance toward meeting these diverse demands. However, voting often fails to identify a candidate with a majority support. This may lead to complex scenario unless handled in a proper manner. NHME-AMC [5] algorithm works for a partitioned network where majority consensus cannot be reached to elect the next process to enter critical section (CS).

All the voting based mutual exclusion (ME) algorithms that work on majority consensus inherently conform to safety criterion. However, such algorithms may violate progress condition when no single process gets majority of votes. In this section, a new two-phase, hybrid ME algorithm has been NHME-AMC that works even when majority consensus cannot be reached. Simulation results establish the superiority of the NHME-AMC algorithm as compared to established as well as recent algorithms in terms of low message and time complexity. The second phase of the algorithm, in spite of being symmetric, executes in constant time.

Reduced Assumptions

The following assumptions form an important foundation for the NHME-AMC algorithm for a distributed system with of n processes labelled as N_i for $i \in [1...n]$:

- Initially all processes and links are non-faulty. There is a singular distinguished partition, i.e., a set of processes which elects one candidate.
- Site cannot arbitrary connect to other processes once it has been repaired. Addition is only permitted to a distinguished partition.
- Processes or links may fail before processing the update request in a given site.
- On receiving a request message in phase 1, a process would vote the requesting process only if it has not already voted in favor of some other process.
- On the contrary, in phase 2, a process may vote in favour of any number of processes depending on the time stamp of a request for the P2V.

5.2.1 A Description for NHME-AMC

The NHME-AMC algorithm works in two phases. In the event that in the first phase any candidate process receives majority of votes, the algorithm terminates there and this is no different from conventional majority voting algorithm. However, if no clear winner is found in phase 1, the second phase is initiated. There is a pre-defined non-negative integer threshold τ assumed for the NHME-AMC algorithm. All the processes that have obtained votes greater than this threshold τ build a set S and elect the winner in themselves in the second phase of voting. In phase 2, processes that have already earned votes over the threshold τ sends request for a phase 2 vote (P2V). This time with a request for P2V vote, the time stamp (TS) of the original request for CS by the respective process is also sent. Any node $P_Y \in S$ that has received a request for P2V from another node P_Z sends P2V to Z iff TS(P_Y) > TS(P_Z).

It is to be noted here that phase 2 follows a symmetric algorithm approach to choose the winner. However, as the maximum cardinality of set S cannot exceed $\lfloor n/\tau \rfloor$ for a total of n competing processes, the overall complexity of the NHME-AMC algorithm would be quite low as long as $\tau \gg 0$. It is important toward ensuring both safety and progress condition of the NHME-AMC algorithm so that neither two different processes can enter the CS simultaneously nor it leads to a situation where the algorithm comes to a halt without being able to elect a winner.

Procedure NHME-AMC

Begin

```
Step 1  A process wishing to enter CS votes itself and requests the remaining
        $N - 1$ processes for vote and waits for the responses;
Step 2  If any process $P_i$ gets majority voting, then go to Step 7 else go to Step 3;
Step 3  If there exists a process that gets vote above a pre-set threshold $\tau$ then
        Step 4 else reduce threshold $\tau$ by 1 and return to Step 1;
```

Step 4 All processes P_X voted above threshold τ are collected in a set S. These processes request for a phase 2 vote (P2V) along with the time stamp (TS) of the original request from P_X for CS. P_X is to wait for certain predefined time for the responses from all the processes $P_Y \in S$. When this pre-defined time expires, the process P_x goes back to Step 1.

Step 5 Any process $P_Y \in S$ that has received a request for P2V from another process P_Z sends P2V to P_Z iff $\mathrm{TS}(P_Y) > \mathrm{TS}(P_Z)$.

Step 6 Identify processes P_i that receives $(|S| - 1 - n)$ P2V from peers, where n is the number of disconnected processes.

Step 7 Select P_i for entry into its CS.

End

(See Fig. 5.3).

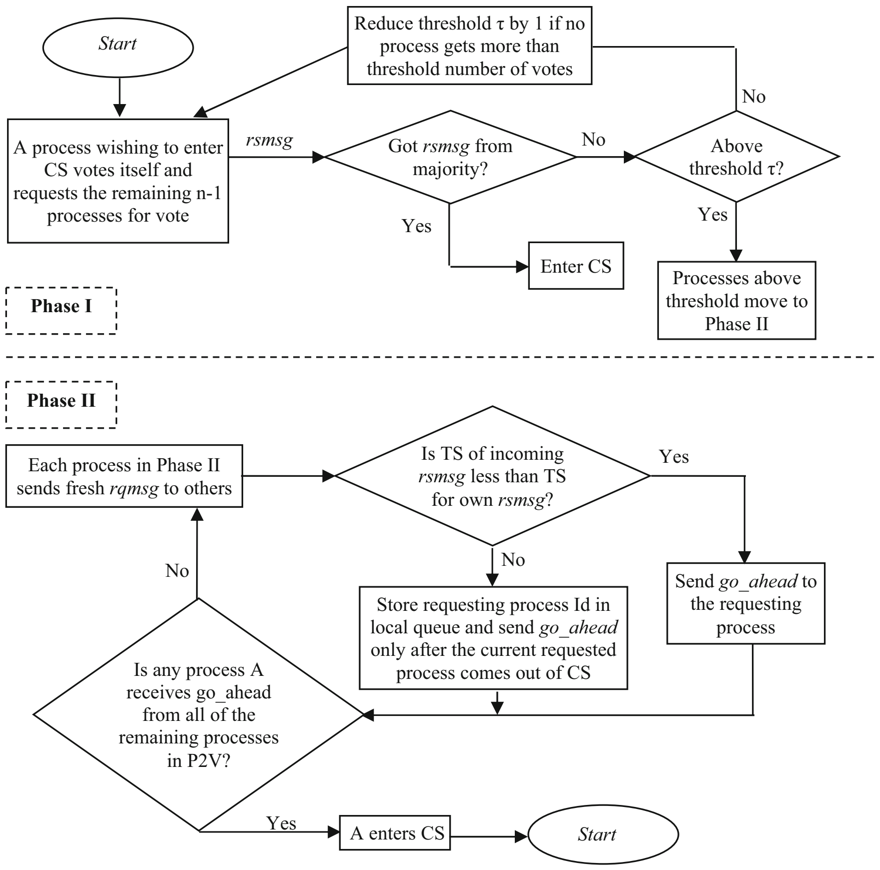

Start
Reduce threshold τ by 1 if no process gets more than threshold number of votes
A process wishing to enter CS votes itself and requests the remaining n-1 processes for vote
rsmsg
Got rsmsg from majority?
No
Yes
Enter CS
Above threshold τ?
No
Yes
Processes above threshold move to Phase II
Phase I
Phase II
Each process in Phase II sends fresh rqmsg to others
Is TS of incoming rsmsg less than TS for own rsmsg?
Yes
No
Store requesting process Id in local queue and send go_ahead only after the current requested process comes out of CS
Send go_ahead to the requesting process
No
Is any process A receives go_ahead from all of the remaining processes in P2V?
Yes
A enters CS
Start

Fig. 5.3 Flowchart of NHME-AMC algorithm

5.3 Performance Analysis for BMaV and NHME-AMC

In this section, the algorithms BMaV and NHME-AMC are evaluated from multiple perspectives. Issues considered for performance evaluation include correctness of the algorithm in terms of both progress condition and safety, message complexity, fairness and other important factors of distributed algorithms.

Safety
A mutual exclusion algorithm satisfies the safety specification of the mutual exclusion problem if it provides mutually exclusive access to the critical section.

Lemma 5.1 *Only one process in phase 2 will get* $(|S| - 1 - n)$ *votes where S is the set of processes shortlisted for phase 2 in the NHME-AMC algorithm, and n is the number of processes that do not respond.*

Proof Without any loss of generalization, one may assume that a standard clock model like Lamport's logical clock model or vector clock would be deployed that puts a unique time-stamp for each and every voting request. In phase 2 of the NHME-AMC algorithm, the process $P_i \in S$ which has the smallest time stamp (TS) of its original request for CS would be the winner.

According to the algorithm, the winner is identified when as it receives P2V from all other $|S| - 1 - n$ processes short listed for phase 2. Any other process $P_k \in S$ for $k \neq i$ cannot get more than $|S - 2|$ phase-2 votes. This is because (i) P_k will not send itself a vote and (ii) P_i will not send P2V to P_k as $\mathrm{TS}(P_i) < \mathrm{TS}(P_k)$.

Lemma 5.2 (Safety) *The NHME-AMC ensures safe mutual exclusion.*

Proof A mutual exclusion algorithm is safe if it ensures that no two processes would enter the respective critical sections simultaneously. In the NHME-AMC algorithm, the winner is selected either from phase 1 or from phase 2. The safety is to be considered separately for the two cases.

Case 1: The winner is selected from phase 1.

This implies that there is a process that gains a majority of votes at the end of phase 1 and hence no other processes can get majority vote. So only one process is allowed to enter the CS satisfying safety criterion.

Case 2: The winner is selected from phase 2.

From Lemma 5.1, only one process would get $(|S| - 1)$ number of votes and will be allowed entry to CS and the property of safety is ensured.

Lemma 5.3 *The value for the threshold* τ *for both the BMaV and NHME-AMC algorithms will never be negative if it is initialized with some positive integer.*

Proof Let τ be initialized with X, for some $X > 0$. In the event that no process could cross this threshold τ for some iteration of these algorithms, the value of τ is reduced by 1 in Step 3 without checking its present value. We shall show that τ cannot be negative by the method of contradiction.

Let us assume that at some point the value of τ becomes -1. This implies (i) in the previous iteration τ was equal to 0 and (ii) no process has obtained any vote greater than $\tau = 0$ in that iteration. However, this is in contradiction with the basic assumption that each process would get at least 1 vote and that from itself. In other words, each process would cross the threshold τ when its value is 0. Therefore, τ would not be reduced further in Step 3. Thus the assumption of $\tau = -1$ is found to be absurd.

Progress Condition
Progress Condition for a mutual exclusion algorithm demands that one of the contending processes for critical section will eventually be allowed to enter the CS, even when no single process gets majority votes.

Lemma 5.4 (Progress Condition) *Progress Condition for the NHME-AMC algorithm is maintained.*

Proof The NHME-AMC algorithm selects a process that has earned majority voting in phase 1, in case such a process exists. The case where no process gets majority voting is discussed next.

From Lemma 5.3, we can conclude that τ cannot be negative and so there are some processes shortlisted for phase 2. The NHME-AMC algorithm always elects at least one process in phase 2 from the shortlisted one as has been shown in Lemma 5.1.

Lemma 5.5 *In phase 2, in spite of being symmetric in nature, the NHME-AMC algorithm runs for constant time.*

Proof For a threshold of τ, the number of processes that can earn τ votes and get into phase 2 cannot exceed $\eta = \lfloor n/\tau \rfloor$. In other words, the cardinality for the set S mentioned in Step 4 of the NHME-AMC algorithm cannot exceed $\lfloor n/\tau \rfloor$. Each of these nodes would send a request for P2V and receives $\eta - 1$ to 0 votes depending up on the time-stamp of the request for CS. If the threshold is set to as low as 15% of the total voting processes, then η cannot exceed following the equation $\eta = n/\tau$. This effectively implies a constant time complexity for phase 2 of the execution.

Correctness
The correctness of control algorithms is typically defined as a collection of safety and liveness. In Lemma 5.2, the safety property has been proved. In Lemma 5.4, progress condition of the algorithm has been proved. Any existing mutual exclusion algorithm [6] that ensures liveness may also be used for this purpose. In this consideration, the NHME-AMC two-phase solution provides a framework that is compatible with many different existing voting algorithms that maintain both safety and liveness. The NHME-AMC two-phase algorithm therefore may be claimed for correctness in tandem with a voting mechanism that is used in phase 1 which ensures liveness in execution.

Storage Requirement

The NHME-AMC solution requires storing very little data at the participating processes. In fact, a process needs to know the total number of processes in the system and time-stamp of its own request for CS.

Message Complexity

The number of messages per critical section access can be deterministically expressed as a measure of concurrency of requests. Let us assume that a total of m out of n processes want to enter respective critical sections. Each of these m processes would request the remaining $n - 1$ processes for vote and a total of $m * (n - 1)$ requests would be sent. In phase 1, one process is allowed to cast only one vote. Therefore, the number of voting messages would be $(n - 1)$. Thus, in phase 1, the average number of messages exchanged per CS request would be

$$K = \frac{m * (n-1) + (n-1)}{m} \approx O(n) \tag{5.1}$$

For a threshold of τ, the number of processes that can earn τ votes and get into phase 2 cannot exceed $\eta = \lfloor n/\tau \rfloor$. In other words, the cardinality for the set S mentioned in Step 4 of the NHME-AMC algorithm cannot exceed $\lfloor n/\tau \rfloor$. The symmetric approach followed in phase 2 involves the exchange of messages between only these η numbers of processes. Each of these nodes would send a request for P2V and receives $\eta - 1$ to 0 votes depending up on the time stamp of the request for CS. Hence, the average number of messages exchanged in phase 2 for each request to CS would be

$$\begin{aligned} \Upsilon &= \frac{\eta + (\eta - 1) + (\eta - 2) + (\eta - 3) + \cdots + 1}{\eta} \\ &= \frac{\eta + 1}{2} \\ \text{i.e.,} \quad \Upsilon &\leq \lfloor n/\tau \rfloor \end{aligned} \tag{5.2}$$

Therefore, adding the cost from Eqs. 5.1 and 5.2, the total number of control messages exchanged is $O(n)$, for a total of n competing processes. Besides, in the NHME-AMC algorithm, even when threshold is close to 30% of the total processes, the number of such processes η entering to phase 2 of the algorithm cannot exceed 3. If threshold is set to 15% of the total processes, then η cannot exceed 6 following the equation $\eta = n/\tau$. This effectively implies a constant time complexity for phase 2 of the execution. Thus, the algorithm terminates faster and with much lower message complexity compared to what may appear to be the average or even worst case performances from Eqs. 5.1 and 5.2.

Table 5.1 Simulation parameters

Parameters	Value
Connection topology	Connected graph topology
Number of processes in the graph	Gradually increased from 6 to 12
Edge length	Static
Request time	Network communication delay
Release time	Twice of network communication delay
Length of CS in terms of execution time	Pre-defined
Maximum degree of a node	4
Minimum degree of a node	2
Priority of process in the tree	Increases with version number
Maximum number of requests	Gradually increased (4, 6, 8, 10, 16) 95

Simulation Results

Similar to NHME-AMC algorithm, the timed-buffer distributed voting algorithm (TB-DVA) [7] also uses two phase commit protocol and Lamport's logical time stamping. Hence, TB-DVA is considered to benchmark the performance of the NHME-AMC algorithm in terms of turnaround time for a batch of concurrent processes.

Simulation Performance of the NHME-AMC Algorithm with TB-DVA and RA

On the other hand, the NHME-AMC algorithm, if it has to enter in its phase 2, uses symmetric approach. It's a well known fact that symmetric algorithms, in spite of being simple to implement, typically involve very high message complexity. Hence, in order to assess message complexity, the NHME-AMCalgorithm is compared with well-established Ricart–Agrawal symmetric algorithm (RA) [8]. The results of simulation for the NHME-AMC method vis-a-vis RA and TB-DVA are shown in Sect. 5.1 (Table 5.1).

A connected network topology is considered. The size of the network is gradually increased from 6 to 12 with different connections between the nodes. In order to make a comparative analysis with TB-DVA [7], the value of request time and release time are selected on the same basis as followed for TB-DVA [7]. The difference between the release and the request time stamps is taken as the turn-around time for a particular job. In every case, we compute the average of all results. We also consider the number of requests as follows:

Network size	6	8	10	12
Number of requests	4	5	6	8

We consider the length of CS for different jobs as follows:

Site	S1	S2	S3	S4	S5	S6	S7	S8	S9	S10	S11	S12
CS length in time (ms)	3	7	10	2	12	8	11	3	4	7	11	5

Time Complexity of the NHME-AMC Algorithm as Compared to TB-DVA
In the NHME-AMC algorithm, only the request message is sent to each of the remaining $n - 1$ nodes in the partition. However, for phase 1, the grant messages are considered only from majority of the nodes. Thus, the total number of messages exchanged per token request granted for phase 1 will be $n + n/2$, where n is the total number of nodes in the network.

As in [7], the request time is selected according to the network communication delay and release time is considered as twice of the network communication delay. In order to plot the results in graphs, we have assumed that network communication delay is equal to the total number of nodes in millisecond (Fig. 5.4).

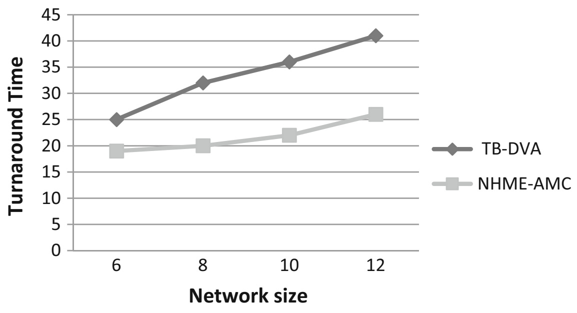
In the NHME-AMC algorithm, we assumed that 50% of candidates are granted permission in phase 1 and calculated the average turnaround time on that basis. Thus, the total turnaround time equals to request time for phase 1 added with that for phase 2 and length of CS. The NHME-AMC algorithm works better than the TB-DVA algorithm for turnaround time.

In Fig. 5.5, we consider a fixed size network with 32 nodes. The number of nodes requesting to enter the CS is chosen to be 8(=25%), 16(=50%) and 24 (=75%). In some cases, TB-DVA does not elect any candidate when no single process gets majority of votes. The NHME-AMC algorithm, however, gives a solution in all these cases.

Figure 5.5 also establishes that the turnaround time of the NHME-AMC algorithm is less than TB-DVA in all cases.

Another set of simulation results is generated for networks by gradually increasing its size from 4 to 16. In Fig. 5.6, we see that NHME-AMC algorithm selects a process irrespective of whether majority votes are obtained or not for

Fig. 5.4 Turnaround time for TB-DVA and the NHME-AMC algorithm



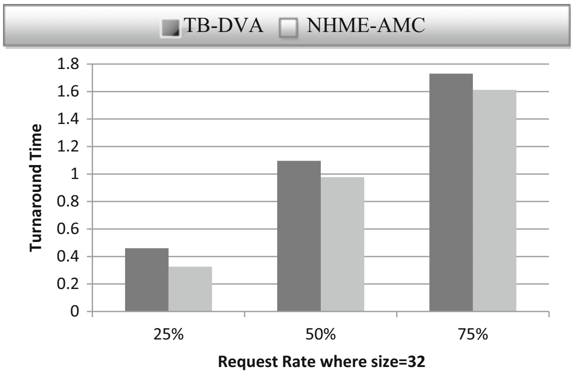


Fig. 5.5 Turnaround times for TB-DVA and NHME-AMC algorithms

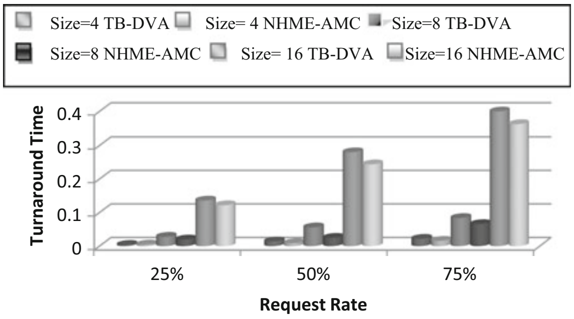


Fig. 5.6 Execution time of TB-DVA versus NHME-AMC for increasing network size

networks of all sizes. The turnaround time for the NHME-AMC algorithm is observed to be less than that of TB-DVA algorithm for all the cases.

Message Complexity of NHME-AMC Algorithm as Compared to RA Algorithm

The simulations setting for comparative performance of the NHME-AMC algorithm with analysis with Raymond's algorithm is very similar to that for TB-DVA.

However, some variations of these settings such as different number of nodes have been used in these experiments. These variations are described while explaining respective results (Table 5.2).

In Fig. 5.7, plots are generated for different network sizes from 4 nodes to 16 nodes. In each of the cases, we consider that 25, 50 and 75% of nodes have requested to enter the CS. The point to be noted here is that although Ricart–Agrawala algorithm assumes that the candidate node with the lowest time stamp receives consent from all of the remaining nodes, this often does not happen in reality. This is due to message loss or partitioning of the network. In Fig. 5.7, we observe that the message complexity of the NHME-AMC algorithm is less than that of Ricart-Agrawala for all the cases.

Table 5.2 Performance comparison of permission-based algorithms

Algorithm	Evaluation measures				Description
	Message complexity		Synchronization delay	Decision theory	
	Heavy load	Light load			
Lamport's	$3(N-1)$	$3(N-1)$	T	Static	Prioritize with time stamp
Ricart–Agrawala's [8]	$2(N-1)$	$2(N-1)$	T	Dynamic	Get $n-1$ permissions
Quorum dynamic	$O(Q)$	$O(Q)$	3T	Dynamic	Generate dynamic quorum
TB-DVA [7]	$5(N-1)$	$3(N-1)$	2T	Two-phase voting	Fault-tolerance and security
NHME-AMC algorithm	$O(N)$	$O(N)$	2T	Two-phase voting	No majority consensus needed
Billiard quorum's [9]	$\sqrt{2}\sqrt{N}$	$\sqrt{2}\sqrt{N}$	T	Coterie-based	Multidimensional voting
T.H. Thomas's	$[(N+1)/2]$	$[(N+1)/2]$	2T	Majority voting	Introduce the concept of voting

Q is the number of quorum members

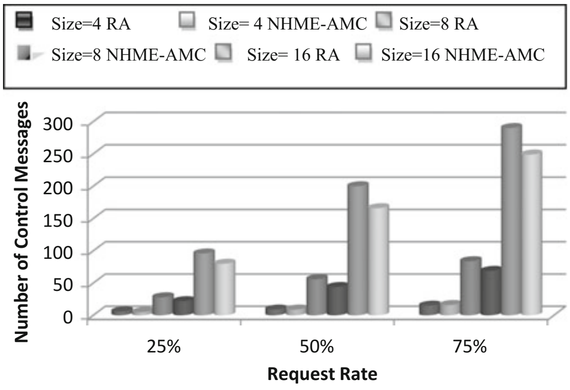


Fig. 5.7 Control messages for RA algorithm versus NHME-AMC algorithm

5.4 Concluding Remarks on Voting-Based ME Algorithms

Voting-based algorithms for mutual exclusion often cannot choose a candidate process when no single process earns majority voting. In this chapter, two new voting-based algorithms are discussed that work where majority consensus cannot be reached to elect the next process to enter critical section. BMaV, the first of the two algorithms, is discussed in Sect. 5.1. Lemma 5.3 in Sect. 5.3 establishes the theoretical soundness of BMaV. However, the algorithm requires knowledge about the resources allocated to different processes. This could be a considerable overhead in a run-time environment and shall worsen the overall performance of a distributed system.

The second voting based approach NHME-AMC discusses in Sect. 5.2 is free from such issues. It has been proved that the NHME-AMC and BMaV algorithms maintain progress condition and also ensures safeness and liveness. The solution is compatible with any majority voting based approach that may be used in phase 1 of the NHME-AMCalgorithm. Phase 2 essentially selects the candidate process from a group of processes that get votes above a system-defined threshold τ based on time stamp of the original request for entering into the critical section. The solution maintains correctness in tandem with an appropriately selected algorithm for voting in phase 1. The simulation results and the theoretical analysis establish that the message complexity and execution time for the NHME-AMC solution is better than the existing solutions compared.

References

1. Thomas, T.H.: A majority consensus approach to concurrency control for multiple copy databases. ACM Trans. Database Syst. **4**(2), 180–209 (1979)
2. Stoica, I., et al.: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of ACM SIGCOMM 2001 (2001)
3. Ricart, G., Agrawala, A.K.: An optimal algorithm for mutual exclusion in computer networks. Commun. ACM **24**(1), 9–17 (1981)
4. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new voting-based mutual exclusion algorithm for distributed systems. In: 4th Nirma University International Conference on Engineering (NUiCONE-2013), pp. 1–5 (2013)
5. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new hybrid mutual exclusion algorithm in absence of majority consensus. In: Proceedings of the 2nd International Doctoral Symposium on Applied Computation and security System, ACSS (2015, in press)
6. Carvalho, O.S.F., Roucairol, G.: On mutual exclusion in computer network. Commun. ACM **26**(2), 146–147 (1983)
7. Suzuki, I., Kasami, T.: An optimality theory for mutual exclusion algorithms in computer science. In: Proceedings of IEEE International Conference on Dist. Comp. Syst., pp. 365–370 (1982)
8. Mittal, N., Mohan, P.K.: A priority-based distributed group mutual exclusion algorithm when group access is non-uniform. J. Parallel Distrib. Comput. **67**(7), 797–815 (2007)
9. Singhal, M.: A dynamic information structure mutual exclusion in distributed system. IEEE Trans. Parallel Distrib. Syst. **3**(1), 121–125 (1992)

Chapter 6
Conclusions

We divide this rather small chapter in two separate sections. We start with a summary of the new algorithms discussed in the book in Sect. 6.1. This is followed by some observations in Sect. 6.2 on the impact of the new algorithms toward opening up newer, exciting research directions for future.

6.1 Summary of the Works Described in the Book

In this book, we present a collection of new mutual exclusion algorithms that aim to improve state-of-the-art scenario as far as process synchronization is considered in a distributed system. MRA-P [1] and FAPP [2] are two token-based algorithms on an inverted tree topology. The primary motivation behind these algorithms has been to ensure fairness among equal priority processes while confirming that higher priority processes are allowed to access CS before lower priority jobs. The most significant improvement that has been achieved by in MRA-P, is that the competing processes are given token first considering their priorities and then following the order in which the requests occur. This is definitely an improvement over the conventional token-based ME algorithms like Raymond's solution [3]. However, the major performance bottleneck for MRA-P is that the algorithm uses large number of control messages. FAPP [2] overcomes this. The MRA-P algorithm has a message complexity O(n). In comparison, the total number of control messages used in FAPP for a N-node balanced binary tree is $2 * (H - 1) = 2 * (\log 2\ N - 1) \approx \mathrm{O}(\log N)$. FAPP maintains liveness for lowest priority process although higher priority processes are preferred by the algorithm. However, a stable, hierarchical topology like tree is often unrealistic for many of the applications on an underlying ad hoc network where topologies keep on changing due to frequent link failures and process mobility.

In Chap. 4 of the book, we have discussed a new token-based ME algorithm that works for directed graph topology, with or without cycles (LFRT) [4]. This solution overcomes problems that may occur due to the lack of alternate paths with tree-based topology. Besides maintaining the correctness in terms of liveness and safeness, the algorithm ensures fairness in allocating the token amongst the competing processes.

We further discussed (LFRT-P) [5] which is a major improvement over LFRT in the sense that the later approach maintains fairness while considering a dynamic process priority. LFRT-P is also resilient to link failure. This makes it ready for deployment where the underlying network is not quite stable in terms of connectivity. Besides, LFRT-P also takes into consideration priority of participating processes. This was not in the scope of LFRT.

Voting-based algorithms for ME often cannot choose a candidate process when no single process earns majority voting. This could be due to communication failure leading to loss of one or more voting responses. The proposed BMaV [6] and NHME-AMC [7] algorithms described in Chap. 5 both aim to solve the problem when majority consensus cannot be reached to elect the next process to enter critical section (CS). The correctness criteria for both the algorithms are proved. NHME-AMC [7] is a two-phase solution. It is compatible to any existing priority-computing approach. The phase two selects the candidate process from a group of processes that get votes above a predefined threshold τ. The solution also uses aging to ensure that no process continue to starve for an indefinite period. The message complexity and execution time for the presented solution are lower than the existing solutions compared. Simulation results and the theoretical analysis both confirm this.

6.2 Impact of New Algorithms on Future Research for Process Synchronization in Distributed Systems

The work documented in this book is done since 2005. It is satisfying to note that some of the new algorithms described in the preceding chapters are cited and extended in significant manner by other researchers addressing similar research domains. In this section, we mention couple of instances to demonstrate that the new algorithms are indeed relevant in the context of future research directions for process synchronization in distributed systems. MRA-P [1] is cited by seven other researchers. In [8], the authors use the same approach like MRA-P [1] is good for solving the unfairness issue by adding extensions to original Raymond's algorithm. Therefore, the algorithm proposed by [8] claims that the unfairness issue can be solved.

FAPP [2] is cited by as many as eight researchers in last few years. Authors of [9] proposed to add some heuristics on our work in FAPP [2] to slow down the frequency with which priority of pending requests is increased. They claim that the algorithm is an extension of FAPP [2] that aims to minimize the number of priority violations without starvation. Works in [10] demonstrate the potential of our work in FAPP [2] on cloud computing. Here, the authors aimed to develop a tree-based distributed mutual exclusion algorithm that supports the service level agreement (SLA). FAPP [2] is modified by adding a deadline is associated with every request. They have claimed to reduce the number of SLA violations with this modification of our work in FAPP.

References

1. Karnar, S., Chaki, N.: Modified Raymond's algorithm for priority (MRA-P) based mutual exclusion in distributed systems. In: Proceedings of ICDCIT 2006. LNCS 4317, pp. 325–332 (2006)
2. Kanrar, S., Chaki, N.: FAPP: A new fairness algorithm for priority process mutual exclusion in distributed systems. Special issue on recent advances in network and parallel computing. Int. J. Networks **5**(1), 11–18 (2010). ISSN: 1796-2056
3. Raymond, K.: A tree-based algorithm for distributed mutual exclusion. ACM Trans. Comput. Syst. **7**, 61–77 (1989)
4. Kanrar, S., Choudhury, S., Chaki, N.: A link-failure resilient token based mutual exclusion algorithm for directed graph topology. In: Proceedings of the 7th International Symposium on Parallel and Distributed Computing—ISPDC 2008 (2008)
5. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new link failure resilient priority based fair mutual exclusion algorithm for distributed systems. J. Network. Syst. Manage. (JONS) **21**(1), 1–24 (2013). ISSN 1064-7570
6. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new voting-based mutual exclusion algorithm for distributed systems. In: 4th Nirma University International Conference on Engineering (NUiCONE-2013), pp. 1–5 (2013)
7. Kanrar, S., Chaki, N., Chattopadhyay, S.: A new hybrid mutual exclusion algorithm in absence of majority consensus. In: Proceedings of the 2nd International Doctoral Symposium on Applied Computation and Security System, ACSS (2015, in press)
8. Challenger, M., Haytaoglu, E., Tokatli, G., Dagdeviren, O., Erciyes, K.: A hybrid distributed mutual exclusion algorithm for cluster-based systems. Math. Probl. Eng. **2013**, 1–15 (2013)
9. Lejeune, J., Arantes, L., Sopena, J., Sens, P.: Agreement for distributed mutual exclusion in cloud computing works on cloud computing. In: 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid, pp. 180–187 (2012)
10. Swaroop, A., Singh, A.K.: A distributed group mutual exclusion algorithm for soft real time systems. World Acad. Sci. Eng. Technol. Int. J. Comput. Control Quantum Inf. Eng. **1**(8) (2007)