Design of Algorithms for Shared Resource Allocation and Certain Selected Problems in Distributed Systems

Thesis submitted
in partial fulfilment for
the award of the Degree of

DOCTOR OF PHILOSOPHY

by

R.Rajendra Prasath

Department of Computer Science
University of Madras
Chennai - 600 005

February 2004

CERTIFICATE

We certify that the thesis entitled, **"Design of algorithms for shared resource allocation and certain selected problems in distributed systems"** submitted for the Degree of Doctor of Philosophy by Mr. R.Rajendra Prasath is the record of research work carried out by him during the period from November 1998 to February 2004 under our guidance and supervision, and that this work has not formed the basis for the award of any degree, diploma, associateship, fellowship or other titles in this university or any other University or Institution of higher learning.

Chennai - 5
20.02.2004

Dr. P.Thangavel
Reader and Head - Supervisor

Dr. R.Sahadevan
Reader in Mathematics - Co-Guide

DECLARATION

I declare that the thesis entitled, **"Design of algorithms for shared resource allocation and certain selected problems in distributed systems"** submitted by me for the Degree of Doctor of Philosophy is the record of work carried out by me during the period from *November 1998* to *February 2004* under the guidance of **Dr. P. Thangavel**, Supervisor and Head, Department of Computer Science, University of Madras and **Dr. R. Sahadevan**, Co-Guide and Reader, The Ramanujan Institute for Advanced Study in Mathematics, University of Madras and has not formed the basis for the award of any degree, diploma, associateship, fellowship, titles in this or any other University or other similar institution of higher learning.

Chennai - 5
20.02.2004

R.Rajendra Prasath

Dedicated to:

Late K.Viswanathan Durai

Abstract

A distributed system refers to a collection of autonomous processors which communicate with each other exclusively by sending messages. In such a system, an effective management of distributed shared resources such as files and distributed objects is important. The problem is how to design a preferred method to accomplish mutual exclusion for the shared resource by restricting its use to only one processor at a time. In addition, adding the properties such as correctness, freedom from deadlock, freedom from starvation, fairness and fault-tolerance to the solution of distributed systems attracts more attention to this problem.

We have proposed distributed algorithms for single shared resource allocation, distributed sorting and prefix computation problems. For the allocation of single shared resource, we use request based token control strategy in which the token is circulated only if it receives a request for the shared resource. This strategy avoids an extensively large amount of communication that might be spent to manage a seldomly used resource in conventional hot-potato based token passing strategies.

In this dissertation, we have investigated single shared resource allocation using request based token control strategy in various interconnection networks. Throughout our investigation, we assume the lack of a global clock, lack of shared memory, lack of the source and destination identities of the processors and an unpredictable but finite message delay.

Chapter 1 presents the review of literature and a brief summary of the present research work. In Chapter 2, we have proposed an algorithm D' for single shared resource allocation in a bidirectional ring network which does not require an additional check tour and another algorithm $L1$ for single shared resource allocation in a linear array. In chapter 3, request based token control algorithms with their extensions for controlling the allocation of single shared resource in ring extension topologies like touching rings and intersecting rings have been proposed. Chapter 4 addresses the single shared resource allocation problem in interconnected rings, single side wrap around mesh and 2-dimensional regular meshes. Chapter 5 describes resource allocation problem in general networks. Here the given undirected network is converted into token based acyclic network and then the proposed algorithms for allocating the single shared resource is implemented using request based message passing strategy. Here, we have relaxed the assumption of knowing the identity of neighbours of Raymond's work(Raymond, 1989). Then we have extended the algorithm to the embedded linear array of networks. The proposed algorithms for single shared resource allocation in various interconnection networks prevent starvation and the request messages are served within a finite time.

Next we have investigated sorting and prefix computation problems in distributed contexts. In chapter 6, we have proposed a distributed sorting algorithm for sorting n elements which are distributed over n processors arranged in a line network. This algorithm prevents the creation of copies of elements at intermediate processors. Then we have proposed algorithms for distributed sorting and prefix computation in a static ad hoc mobile network. Concluding remarks and future work are described in chapter 7.

Acknowledgements

It is an extensive search in identifying a suitable phrase to thank my beloved Supervisor Dr.P.Thangavel, Reader and Head, Department of Computer Science, University of Madras for his guidance through valuable suggestions and constant support in all aspects during this research work. I am thankful to my co-guide Dr.R.Sahadevan, Reader in Mathematics, The Ramanujan Institute for Advanced Study in Mathematics, University of Madras for his guidance throughout my research period, Dr.G.Gopal, Professor in Statistics and Dr. V.Thangaraj, Professor in Mathematics for their kind support.

This is a unique opportunity to express my sincere thanks to Prof. Assaf Schuster, Prof. Le Lann, Prof. Suzuki, Prof. Kasami, Prof. Ajay Kshemkalyani, Prof. Nagabhushan and Dr. Xin Yuan who provided reprints which really motivated me to share with their knowledge and provided friendly encouragement. Also the timely efforts extended by Prof. Lei Li, Prof. Yamashita, Prof. Chalamaiah and others, not listed here, are gratefully acknowledged.

Financial assistance, received from Council of Scientific and Industrial Research (CSIR), University of Madras and All India Council for Technical Education (AICTE) at various stages of my research work is gratefully acknowledged.

My special thanks are due to the officials in the Libraries of the Institute of Mathematical Sciences, Chennai, the Indian Institute of Technology Madras and

the Indian Institute of Science, Bangalore as well. Also my special sincere thanks are due to Mr. P.Thangaraj, Mrs. & Mr. Dr.N.Godhantaraman for their timely help in getting some of the reprints of recent research papers.

I am grateful to my parents for their unbounded *patience and love*. Without their affection, this work would never have come into existence.

I should mention my gratitude to all my well wishers in and around the Department of Computer Science, University of Madras who had shown their support to their extreme extent. Also I thank the authorities of the University of Madras for their natural treatment throughout my research period.

Finally, I wish to thank:, My sisters and my uncle (for their care); my philanthropist Thiru. S.Ramesh (for his algorithmic thoughts); my respected exponent Thiru. G.Narayanan (for his encouragement and support in all aspects since from my post graduation); our technical staff Thiru. M.Gopal and Thiru. L.Umapathy (for their technical treatment); evergreen supporter D.Sampath Kumar (for his practical estimations); my catalytic power: Late K.Viswanathan Durai (for his proud and love); and all my well wishers ... (for all good and bad times we had together);

Chennai - 5 R.Rajendra Prasath
20.02.2004

Additional Acknowledgement

I am very much grateful to the anonymous examiner for his valuable suggestions and comments, indicating the corrections to be carried out, which enriched our understanding further.

⊘ Communications related to this thesis ⊘

- ⊘ R.Rajendra Prasath and P.Thangavel, Shared resource allocation using token based control strategy in ring extension topologies, in: J.C.Misra and S.B.Sinha (Eds.) *Recent Trends in Mathematical Sciences*, Narosa Publishing House, New Delhi, 2000, pp. 53-63.

- ⊘ R.Rajendra Prasath and P.Thangavel, Token based message passing in bidirectional ring extensions, *Journal of the Madras University(WMY-2000 Special Issue)-Section B: Sciences*, Vol. 52 (2000) pp. 145-159.

- ⊘ P.Thangavel and R.Rajendra Prasath, A note on token based control in rings and linear arrays, *Journal of the Computer Society of India*, Vol. 32, No. 3, March 2002, pp. 62-65.

- ⊘ R.Rajendra Prasath and P.Thangavel, Token based control algorithm with central coordinators, in: P.Thangavel (Ed.), *Algorithms and Artificial Systems*, Allied Publishers, Chennai, India, 2003, pp. 25-40.

- ⊘ R.Rajendra Prasath and P.Thangavel, Shared resource allocation using token passing strategy in interconnected networks, *Information*, Vol.6, No.2 (2003) pp. 197-206.

- ⊘ R.Rajendra Prasath, and P.Thangavel, Token based control algorithms for shared resource allocation in general networks, in: Proc. 3rd Int. Conf. on Information, (Info2004), 2004, 520-523.

- ⊘ R.Rajendra Prasath and P.Thangavel, Distributed algorithms for sorting and prefix computation in message passing systems, *under preparation*.

Contents

Abstract i

Acknowledgement iii

Communications related to this thesis vi

List of Figures x

List of Tables xi

1 Introduction 1

1.1 General 1

1.2 An overview of distributed algorithms 3

1.3 Distributed systems: single shared resource allocation, sorting and prefix computation 6

1.4 Related works 11

1.4.1 Shared resource allocation 12

1.4.2 Sorting and prefix computation 15

1.5 Present work 16

2 Shared resource allocation in rings and linear arrays **21**
2.1 Introduction 21
2.2 Allocation of a shared resource in rings 26
2.3 Conclusion 35

3 Shared resource allocation in ring extension topologies **36**
3.1 Introduction 36
3.2 Allocation in touching rings network 38
3.3 Allocation in intersecting rings network 46
3.4 Conclusion 54

4 Allocation in interconnected networks **55**
4.1 Introduction 55
4.2 The model of communication networks 57
4.3 Allocation in interconnected rings 59
4.4 Allocation in single side wrap around mesh 64
4.5 Allocation in regular meshes 70
4.6 Conclusion 77

5 Allocation in general networks **78**
5.1 Introduction 78
5.2 The model of communication network 81
5.3 Token-based algorithm in general networks 82
5.3.1 Allocation algorithm using a queue: GNet 83

5.3.2 Differences between Raymond's algorithm(1989) and the proposed algorithm . . . 93
5.4 Allocation by embedding token oriented acyclic network into a linear array . . . 94
5.5 Conclusion . . . 96

6 Distributed algorithms for sorting and prefix computation 97
6.1 Introduction . . . 97
6.2 The problem and the computational model . . . 100
6.3 A time-optimal algorithm for sorting in line network . . . 101
6.4 Simulation Results . . . 110
6.5 Sorting on static ad hoc mobile networks . . . 112
6.6 Computing prefix sums on static ad hoc mobile networks . . . 118
6.7 Conclusion . . . 123

7 Conclusions and future work 124

Bibliography 129

List of Figures

2.1 A linear array of n=5 processors 24
2.2 Single shared resource allocation in a linear array 34

4.1 Single side wrap around mesh network 65

5.1 (a) A general network with n=12 processors; (b) a spanning tree; (c) an indexed tree 83
5.2 Token based control algorithm for shared resource allocation among leaf processors 85
5.3 Token based control algorithm for shared resource allocation among central processors 88

6.1 An example of the $(n-1)$ round algorithm with n=5 elements . . . 103
6.2 Performance comparison of the proposed algorithm with Sasaki's Algorithm 111
6.3 Proposed sorting algorithm for a static ad hoc mobile network. . . . 113

List of Tables

6.1 Simulation results for distributed sorting on line network 111

Chapter 1

Introduction

1.1 General

A distributed system consists of a collection of geographically dispersed autonomous processors which are connected by a communication network. In such a system, an effective management of distributed resources is important, but is complicated by the requirements of reliability and effectiveness. The schemes used to manage resources can be classified into centralized management and completely distributed management. In the centralized management scheme, access to the shared resource is controlled by a single authenticated processor where as in completely distributed management, access to each shared resource is controlled through the co-operation among all processors. This co-operation among processors can be achieved through the design of a distributed mutual exclusion algorithm that provides a mechanism to secure the integrity of distributed shared resources through serializing the concurrent accesses to them.

Completely distributed management scheme has many advantages compared with centralized systems. However, designing distributed algorithms to control

distributed systems is by no means easy because of the following reason: processors must send / receive messages to other processors to get enough information to do their tasks. Messages are delivered with unpredictable but finite delay and therefore, in principle, there is no process which controls the entire distributed system. Hence processors in a distributed system communicate only by message passing and there is no shared memory. Also the message passing style of programming is widely used in parallel computers. The primitives to *send* and *receive* messages hide low level architectural details and are ideal for programming many large applications. While message passing systems have been in use for over a decade, relatively a few results concerning the complexity of message passing protocols are available due to the lack of theoretical models that appropriately capture the issues related to communication.

Underneath the message passing strategy, a message passes through several intermediate processors before reaching the destination. During each phase, the message buffer goes through the network interface connecting the processor to the network. After occupying the buffer at the network interface of the destination, the message is either processed or removed. Whenever a message can not get the critical resource it needs, it must wait. When messages wait for a long time, the communication delay can cause processor idling, thereby reducing overall performance greatly (Liu *et* al. 2001).

1.2 An overview of distributed algorithms

An algorithm is said to be *distributed* if it operates on a physically or logically distributed computing architecture. The idea of distributed algorithms centers around messages, synchronizing shared use of scarce resources and casual dependencies of actions for some particular computing architecture. The decisive properties of each distributed algorithm include aspects of safety and liveness, intuitively characterized as *nothing bad will ever happen* and *eventually something good will happen*, respectively. Distributed systems often consist of subsystems which share scarce resources. Such a resource (e.g., a shared variable) is accessible by at most one component simultaneously (Lynch 1996).

To solve a problem, processors in a distributed computing system must communicate among themselves. For both large computer networks and Very Large Scale Integrated (VLSI) architectures, the inclusion of a shared memory to facilitate inter processor communication is usually infeasible (Loui 1984). An effective algorithm for distributed systems should minimize the message traffic in order to minimize the computation time. The communication model is assumed to be asynchronous, requires decentralized control, admits no shared memory and permits data transfers only on a communication network.

Distributed algorithms are designed to run on hardware consisting of many interconnected processors. Pieces of a distributed algorithm run concurrently and independently, each with only a limited amount of information. The algorithms are supposed to work correctly, even if the individual processors and communication

channels operate at different speeds and even if some of the components fail.

In distributed systems, the preferred method to accomplish mutual exclusion for the shared resource is by restricting its use to only one processor at a time. An effort to guarantee exclusive access to a shared resource is one of the first problems encountered in parallel programming. The term *Critical Section* is used to characterize a structural program abstraction for concurrent access to a shared resource (Maekawa 1985, Makki 1994-a, Makki *et* al. 1994-b). Regarding the critical section and its exclusive access to it, the following conditions must be satisfied:

i) Any request from a process to enter the critical section will be granted in finite time.

ii) Any process currently in the critical section will exit in finite time.

iii) At any given time, only one process can enter the critical section.

Any distributed algorithm has to satisfy correctness, deadlock - freeness, starvation - freeness, fairness and fault tolerance. A *Deadlock* is defined as a state in a distributed system where a processor, which desires to enter the critical section, is not allowed entry into the critical section as it has been held by some other processor which in turn waits for some other shared resource. *Starvation* occurs when a processor, which is requesting a resource, never receives the resource while other processors continue to receive the resource and use it. The fairness of the proposed algorithm can be caused by the order in which requesting processors are served with the token relative to the order in which the processors actually requested the

token. However due to lack of a global clock, there is no way for a distributed algorithm to know absolutely the order in which requests are generated. Also a system which exhibits fairness should receive service for requests in the bounded time. A fault may occur due to the failure of a processor or a link or both. Since processors or links are vulnerable to faults, the algorithm designed for the problem of mutual exclusion should tolerate processor or link failures.

Makki *et* al. (2000) have stated that the algorithms designed for distributed systems are characterized by the following properties (Makki *et* al. 2000):

i) All processors have equal amount of information.
ii) All processors make a decision based solely on local information.
iii) All processors bear equal responsibility for the final decision.
iv) All processors expand equal effort in affecting a final decision.
v) Failure of a processor, in general, does not disturb the performance of the whole system.

Usually the distributed algorithm designed to achieve greater performance does not satisfy all of the above properties. However these kind of algorithms are still regarded as *distributed*.

In recent years, two types of ad hoc mobile networks, namely static and dynamic ad hoc mobile networks, are of special interest in distributed computing environment. Past work on modifying the existing distributed algorithms for ad hoc mobile networks includes numerous routing protocols and channel allocation algorithms. Characteristics that distinguish ad hoc mobile networks from existing distributed networks include frequent and unpredictable topology changes and

highly variable message delays. These characteristics make ad hoc mobile networks challenging environments in which to implement distributed algorithms (Walter *et* al. 2001).

Problems that arise in distributed environment include telecommunications, distributed information processing, scientific computing and real time process control, etc. For example, the real world problems such as today's telephone systems, airline reservation systems, banking systems, global information systems, weather prediction systems and aircraft and nuclear power plant control systems depend critically on distributed algorithms. So it is important that the distributed algorithms must run correctly and efficiently in these environments.

In a distributed computing environment, the topology of interconnection network plays an important role. Initially, every processor knows only the links that involve it. Several interconnection topologies, like linear array, bidirectional ring and its extensions, regular mesh, single side wrap around mesh, general network, its embedding architectures in distributed systems and ad hoc mobile networks, are considered in this thesis.

1.3 Distributed systems: single shared resource allocation, sorting and prefix computation

Distributed systems commonly include exclusive resources such as distributed objects that must be managed so that no two processes access the resource at the same time. The scheduling of processors with various resource requirements in this type of system is generally known as resource allocation. One

of the fundamental problems in the area of distributed algorithms is the problem of mutual exclusion that is exclusive access to a shared resource. It is appropriate to mention that the first problem to receive theoretical study is the problem of mutual exclusion. Essentially, the problem involves in managing access to single, indivisible resource that can only support one user at a time. In other words, this can be viewed as the problem of ensuring mutually exclusive access in which certain portions of the program code are executed with in critical regions, where no two programs are permitted to be in critical regions at the same time. Also there exists some uncertainty about which users are going to request access for the resource. This problem of mutual exclusion arises in both centralized and distributed operating systems.

An efficient architecture of a distributed system inherits in the exploitation of parallel or concurrent computation. Concurrence can be achieved by the simultaneous execution of many relatively simple operation sequences. In multicomputer systems, processors memory units are interconnected in a network and computation-enabling data are passed using a message passing protocol. Although the interconnection is much simpler, messages may have to pass through a number of intermediate processors before reaching their destinations. However, request messages are forwarded towards their destination according to a routing scheme and the efficiency of a distributed system as a whole is evaluated by the performance of the routing scheme. In these schemes, the identities of source and destination processors are known. Thus devising such an efficient routing scheme is very important in distributed network architectures.

To our knowledge, the work on solving the problem of distributed mutual exclusion can be classified into two main categories: *Permission based algorithms* and *Token based algorithms*. We wish to note that the above two categories adhere to some interesting properties mentioned in the section 1.2 (Ricart and Agrawala 1981, Suzuki and Kasami 1985, Feuerstein *et* al. 1998, Makki *et* al. 2000).

In Permission based algorithms, message traffic is based on well defined subset of processors. This subset of processors forms a basic for the unified framework, developed by Sandlers (Sandlers 1987). In his framework, when a processor wishes to enter the critical section, it sends a request message to the set of processors indicated by its information structure. The requesting processor may enter critical section only after receiving permission in the form of reply message for each one of its requests. At the end of critical section, that processor must send a release message to each processor in the subset, allowing the requesting processor to proceed to enter the critical section. However, this technique permits possible unfair ordering of request messages (Lamport 1978, Ricart and Agrawala 1981, Lamport 1987, Velazquez 1993).

Non−requesting processors send their permission to requesting ones. Each processor may grant its permission to only one processor at a time. A priority or an order of events has to be established between competing processors so only one of them receives permission from all other processors in the set. Such a processor is allowed to enter critical section. This enforces the requirement for mutual exclusion (Lamport 1978, Ricart and Agrawala 1981, Carvalho and Roucairol 1983, Maekawa 1985, Lamport 1987, Raynal 1988, Agrawala and El Abbadi 1991, Singhal 1993).

In Token based algorithms, a unique token will be circulated from one processor to another processor in the network according to a predefined set of rules understood and adhered by all processors (Dijkstra 1965, LeLann 1977, Lamport 1978, Andrews and Schulz 1982, Suzuki and Kasami 1985, Raynal 1988, Raymond 1989-a, Feuerstein *et* al. 1998, Lodha and Kshemkalyani 2000, Wu and Shu 2002). Whenever a processor needs to access the shared resource by performing mutual exclusion algorithm, it should first obtain a free token and then convert it into a busy token which initiates the execution of critical section. When the shared resource is no longer needed at a processor, then the busy token is destroyed at that processor and a new free token is circulated in the network for further allocation. Then the token is circulated repeatedly even if no processor has requested it. Thus the number of messages exchanged may be unbounded even for a finite set of requests.

The *request message based* token control strategy was presented to avoid the unbounded token movements even for a finite set of requests (Suzuki and Kasami 1985, Feuerstein *et* al. 1998). In this strategy, the token is allowed to circulate only if it is informed of a processor asking the resource and avoids an extensively large amount of communication that might be spent to manage a seldomly used resource. In the literature, there exists a variety of token based algorithms which are distinguished by their method for determining how a processor obtains the token and where the processor sends the token immediately after finishing the critical section (assuming that there is a pending request).

In this thesis, we have proposed single shared resource allocation algorithms using request based token passing strategy in various interconnection networks. We

assume a single shared resource and single token for the token distribution problem. The requests generated for single shared resource are anonymous. Also we assume that a processor can ask again the resource only after that the previous request has been satisfied. Requests are generated in an online fashion: each processor can generate a request for the resource at any time and neither the origin nor the destination of the generated request is known in *priori*. Processors have neither a shared memory nor a common clock, and their speeds are unrelated. Source and destination processor identities are not included either with the request message or the token.

To study the efficiency of shared resource allocation algorithms, we use the following two distinct measures: *the average number of messages per request* and *service traffic* (Tarjan 1985, Feuerstein *et* al. 1998, Horowitz *et* al. 2001). The first measure is *the average number of messages per request* necessary to satisfy a sequence of requests, that is, the worst case ratio between total number of (token and request) messages and the number of requests in the sequence. Alternatively, we also use the worst case number message exchanges per request. The second measure is the *service traffic*, defined as the worst case number of messages that are exchanged in the network between the time in which a processor sends a request message and the time in which the processor receives the token. The service traffic must be bounded in order to prevent starvation.

Among all computational tasks studied by researchers over the past few decades, sorting problem appears to have received the most attention and hardly a month goes by without a new article appearing in a technical journal that provides an-

other facet of distributed sorting (Akl 1989). The distributed sorting problem can be stated as follows: At the initial state, each processor P_i has an element S_i for sorting. Then, the position of each element is rearranged to satisfy the condition, $S_i \leq S_{i+1}$ in each processor P_i, $1 \leq i < n$, at the final state. There exists several models and algorithms for distributed sorting (Loui 1984, Rotem *et* al. 1985, Zaks 1985, Marberg and Gafni 1987, Akl 1989, Leighton 1992, Gerstel and Zaks 1997, Leu *et* al. 2000). We propose distributed sorting algorithms in a static, reliable line network and general networks that form a basis for all sorting problems.

Another computational problem we deal with, in this dissertation is the prefix computation problem which is stated as follows: *Given n values $\{u_1, u_2, \cdots, u_n\}$ distributed over n processors and an associative binary operation $\oplus$, we compute i^{th} prefix sum as $u_1 \oplus u_2 \oplus \cdots \oplus u_i$, for $1 \leq i \leq n$.* There exists several algorithms for prefix computation problem (Lin 1996, Wang *et* al. 1996, Bruck *et* al. 1997, Akl 1997, Lin and Yeh 1999). We have proposed a prefix computation algorithm for token based static ad hoc mobile networks

1.4 Related works

In this section, we describe related algorithms, studied in the literature, proposed for shared resource allocation, distributed sorting and prefix computation in various interconnection networks. The algorithmic study reported are not exhaustive and as the literature in each area is enormous, only selected few algorithms are cited.

1.4.1 Shared resource allocation

Dijkstra (Dijkstra 1965) introduced mutual exclusion for N process system as the requirement that at any moment only one of these N cyclic processors is in critical section. This requirement is still the standard definition of distributed mutual exclusion (Lamport *et* al. 2000). Dijkstra (Dijkstra 1971) first modelled the resource allocation problem as a ring of 5 processors, each accesses the single shared resource. Then he has presented a mutual exclusion protocol for a ring of n processors where each processor can read the state of its anticlockwise neighbor (Dijkstra 1974, Dijkstra 1982). Next Lynch (Lynch 1981) generalized the problem to an arbitrary conflict graph where a node represents a processor and an edge represents the sharing of resources between two processors.

Chandy and Misra (Chandy and Misra 1984) presented a dining philosophers algorithm using an acyclic directed version of the conflict graph. Then Ginat *et* al. (Ginat *et* al. 1989) proposed a drinking philosophers algorithm that solves the problem directly without using a dining philosophers subroutine. As a result, it is more message efficient, but it requires unbounded counters. Welch and Lynch (Welch and Lynch 1993) generalized the modular construction of Chandy and Misra's drinking philosophers algorithm to come up with a drinking philosophers algorithm which uses, as a subroutine, any dining philosophers algorithm.

LeLann (LeLann 1977) has proposed a token based algorithm in which a token is circulated among processors in a logical ring. Any processor which desires to enter the critical section simply waits for the token arrival. After completing the critical section, the token is forwarded along the logical ring. Even though

this approach is simple, it is easy to notice the inefficiency of this method as it can unnecessarily utilize network bandwidth in a lightly loaded system by continuously and needlessly passing the token. This weak condition mandates no upper bound for the number of message exchanges per critical section. Also a major concern of token circulation is to guarantee that the control token is never lost and that there is only one token on the ring.

Ricart and Agrawala (1981) have proposed a mutual exclusion algorithm that requires $2(N-1)$ message exchanges for each critical section invocation in a computer network whose processors communicate only by messages and do not share memory. Then Suzuki and Kasami coined the concept of the "PRIVILEGE" that is used to control the exclusive access of the shared resource (Suzuki and Kasami 1985). This privilege based distributed mutual exclusion algorithm requires at most N messages per critical section. Maekawa's algorithm (Maekawa 1985), for mutual exclusion in a computer network, further reduces the number of message exchanges per critical section entry to $O(\sqrt{N})$. Then Raymond proposed a tree-based distributed mutual exclusion algorithm that depends on the precise topology of the network spanning tree used and the average number of messages required per critical section is $O(\log N)$ (Raymond 1989-a).

Feuerstein *et* al.(1998) have presented a solution to the token distribution problem by means of a permission(control) token in a circular unidirectional ring network in which token and requests are allowed in single direction. They have proposed two protocols for single shared resource allocation in rings in which every request generated will eventually reach the token and *push* it till the next proces-

sor with a pending request. This type of circular token based control mechanism was the primary motivation for the study of the election problem (LeLann 1978), mutual exclusion problem (Raynal 1988) and hub polling systems (Lynch 1996).

Makki *et* al. (Makki *et* al. 2000) have presented a token-based distributed mutual exclusion algorithm for a fully connected (complete) graph with N processors. All connections are assumed to be full duplex connections. Further it is assumed that the message transmission time among processors is unpredictable and no global clock exists. Their algorithm offers a higher degree of connectivity and requires $(N+1)/2$ message exchanges per request in the worst case. This characteristic is highly desirable as it does not unnecessarily burden the network with frivolous message traffic.

Recently Lodha and Kshemkalyani(Lodha and Kshemkalyani 2000) presented a fair distributed mutual exclusion algorithm for distributed systems in which processors communicate by asynchronous message passing. The algorithm requires between $(N-1)$ and $2(N-1)$ messages per critical section access where N is the number of processors in the system. It does not introduce any other overheads over Lamport's (Lamport 1978) and Ricat-Agrawala's (Ricart and Agrawala 1981) algorithms which are the only other decentralized algorithms that allow mutually exclusive access in the order of time stamps of requests. Examples of resource allocation using message passing model can be found in many applications including distributed database and file systems (Rhee 1988). Dhamdhere and Kulkarni (Dhamdhere and Kulkarni 1994) presented a token based $k-$resilent mutual exclusion algorithm that tolerates up to k site/link failures. Fu *et* al. (Fu *et* al. 2000)

1.4.2 Sorting and prefix computation

In the design and analysis of algorithms, sorting problem is one of the most fundamental problems. It has been extensively investigated in distributed contexts. Loui (Loui 1984) presented a simple sorting algorithm on rings. Rotem *et* al. (Rotem *et* al. 1985) deal with the sorting problem where each processor contains a subset of the elements, and static and dynamic versions of the problem are studied. Zaks (Zaks 1985) presented a sorting algorithm for a tree network, then extended it to general networks. Then Marberg and Gafni (Marberg and Gafni 1987) developed a sorting method for a multi-channel broadcast network. Then McMillin and Ni (McMillin and Ni 1992) described the distributed sorting problem with an unreliable network. Hofstee *et* al. have designed a time-optimal algorithm for distributed sorting problem on a line network with restricted local memory (Hofstee *et* al. 1990). However each processor has to store at least two elements and this algorithm fails when each processor has exactly one element. Therefore Sasaki has designed a time-optimal distributed sorting algorithm with a strict lower bound of $(n-1)$ rounds on a line network by creating the copies of elements at intermediate processors (Sasaki 2002). But Sasaki's algorithm does not ensure that each processor always has two elements of the same value at the final round. However, Sasaki's algorithm requires the lower bound of (n-1) rounds in the worst case.

Prefix computation has broad applications and many prefix circuits have been designed (Lin 1996, Wang *et* al. 1996, Bruck *et* al. 1997, Akl 1997, Lin and Yeh 1999). Wang *et* al. have proposed a strict lower bound for parallel prefix with resource constratints (Wang *et* al. 1996). Then Lin and Yeh have proposed parallel prefix algorithms on multiport message passing systems (Lin 1996). Bruck *et* al. have proposed prefix algorithms using all-to-all communications in multiport message passing systems (Bruck *et* al. 1997). Recently, Jana *et* al. have proposed parallel prefix algorithms on extended multi-mesh network (Jana *et* al. 2002).

1.5 Present work

In spite of the developments of the algorithmic studies on shared resource allocation, sorting and prefix computation problems in distributed systems, due to the importance of shared resource allocation problem and its close association with mutual exclusion property in real time applications, the token based control strategy for solving single shared resource allocation problem in various distributed systems attract more attention.

Motivated by the above considerations, in this dissertation, the problem of controlling the allocation of single shared resource, among a set of processors that are arranged in various interconnection networks, using request based token passing strategy, the problem of distributed sorting on line and general networks and the problem of token based prefix computation on a static ad hoc mobile network are investigated.

The organization of the dissertation is as follows.

In chapter 2, we present the token based control strategy for the allocation of single shared resource among n processors that are arranged in a bidirectional ring network and linear array. The first proposed algorithm D' deals with the allocation of single shared resource in a bidirectional ring network in which token moves in anticlockwise direction and requests move in clockwise direction. We wish to point out that the additional check tour is not required in D' unlike in the algorithm D of Feuerstein *et* al.(1998). It satisfies all requirements of processors, requires at most $2(n-1)+1$ messages per request and its service traffic is bounded by $3(n-1)$. The second proposed algorithm $L1$ deals with the allocation of single shared resource for shared resource in a linear array - a ring with a faulty processor. In $L1$, token starts serving the processors along one direction by serving all pending requests until there exists a pending request; otherwise, the token changes its direction if there is a pending request in the other direction. Thus the algorithm serves all processors; requires $2(n-1)+1$ messages per request and $3(n-1)$ service traffic.

In chapter 3, we consider request based token passing strategy to control single shared resource allocation in ring extension topologies. First we consider touching rings network in which two rings that touch at only one common processor through which the routing of request messages as well as token from one ring to another takes place either in unidirectional or in bidirectional way. Here the role of the common processor is significant as it can appropriately switch the token as well as request messages according to the traversal of the token as well as request messages. The algorithm for unidirectional touching rings requires at most $(n-1)$ messages per request and $\frac{(5n-2)}{2}$ service traffic. In bidirectional touching rings, the common

processor balances the traffic by keeping track of request messages received from both rings concurrently. The algorithm for bidirectional touching rings requires $2(n{-}1){+}1$ messages per request and $(3n - 2)$ service traffic in the worst case. Also the proposed algorithm can be generalized to multi-touching rings network. Then we deal with fault-free bidirectional intersecting rings in which processors are arranged in two rings with different total number of processors that intersect at any two arbitrary common processors through which the routing of messages takes place. This algorithm requires n messages per request and $2n$ service traffic.

In chapter 4, we deal with the single shared resource allocation in two interconnected rings, single side wrap around mesh and regular mesh. In an interconnected rings network, there are $\frac{n}{2}$ processors in each ring, where n is the total number of processors(assumed to be even). Here each processor $P_{(i+\frac{n}{2})}$ in second ring communicates with the processor P_i in the first ring. Whenever a request is generated for the shared resource by the processor in the second ring, it will be forwarded to the directly connected processor in the first ring and then the processor in the first ring forwards the received request to its adjacent processor. Token maintains a counter to recognize the receipt of request messages. The algorithm for interconnected rings requires $(n - 1)$ messages per request and service traffic is bounded by $(3n{-}2)$.

In a single side wrap around mesh, n processors are arranged in k rings each with m processors that are connected by a bidirectional channel. We assume unidirectional movements over the ring edges and bidirectional movements over the edges connecting processors in the adjacent rings. In this algorithm, token makes

a check tour in the base ring to learn and serve all pending requests. The proposed algorithm requires $(n-1)$ messages per request and service traffic is bounded by $(2n+m-1)$. Then we consider bidirectional token as well as request movements in a regular $2-$dimensional mesh for single shared resource allocation. In this network, token moves from base row processor to column processors. After serving the request in that column, the token is passed to a processor in the adjacent column. From this column processor, token reaches the base row processor only after serving all pending requests. In this protocol, we do not need the token counter. The algorithm for regular mesh needs $(n-1)$ messages per request and service traffic is $(2n+1)$, in the worst case.

In chapter 5, we have proposed two algorithms based on token passing strategy for the single shared resource allocation problem in general networks. We have first converted the given undirected, fault-free, connected network into a token oriented acyclic network that may be derived from either a minimal spanning tree or an ordered tree network. The first algorithm consists of two phases for shared resource allocation in the acyclic network. This algorithm requires no identities of adjacent processors. We hardly require $Holder$ variables as in (Raymond 1989-a) to maintain the token oriented acyclic network. This algorithm works with local indices assigned by each processor to links connected to it. The first PHASE of the algorithm deals the allocation of single shared resource in leaf processors. The second PHASE of the algorithm controls the movements of request and token in non-leaf processors. This algorithm is volatile to dynamic changes of the network topology and hence this algorithm outperforms Raymond's tree based algorithm

which is a static algorithm (Raymond 1989-a). Also we have proposed another algorithm based on the embedding of the processors in the token oriented acyclic network into a linear array of processors in which token and request messages are permitted to move on both directions. The proposed algorithms may also be useful for ad hoc networks provided the construction of the token oriented acyclic network is based on the self stabilizing approach of Huang and Chen (Huang 1993, Huang and Chen 1993).

In chapter 6, we have proposed a distributed sorting algorithm focused on *median based exchanges* to speed up the computational time for sorting n elements distributed over a set of processors on a line network. The algorithm reduces the number of elements needed for sorting to exactly n without creating the copies of elements at intermediate processors. This idea is entirely different from the approach of odd-even transposition sort and the simulation results show that the proposed algorithm is faster than the Sasaki's distributed sorting algorithm (Sasaki 2002). Even though the proposed algorithm takes $(n-1)$ rounds in the worst case, it is still robust. Then we have proposed a sorting algorithm on a static ad hoc network by keeping track of the token with its message at intermediate processors. Finally we have made an attempt to design an algorithm, for prefix computation in static ad hoc mobile network that can be extended for dynamic ad hoc mobile networks.

The final chapter ends up with concluding remarks and future directions.

Chapter 2

Shared resource allocation in rings and linear arrays

2.1 Introduction

Resource allocation is a fundamental problem in distributed systems consisting of n processors that have no shared memory in common and can communicate only by exchanging messages. The communication delay between processors is totally unpredictable and request messages are not guaranteed to be delivered in the same order in which they were sent. Our objective here is to design a distributed algorithm that realizes a mutual exclusion requirement. This mutual exclusion problem has a significant impact in local area networks. The term *local area networks* was introduced by Clark *et* al.(Clark *et* al. 1978) with ring topology. The processors are logically arranged in a ring in which each processor can communicate with the next processor around the ring. A unique privileged bit - *token* is used for conflict free channel access as contrasted with random access in the ring network. Normally each processor simply relays the received bit stream from the previous processor to the next. The processor having the token is allowed to access the

channel. Then, the token is passed on to the next processor. Thus conceptually, we visualize a *token* as the control for message transmission from one processor to another. This type of circular token based control was the primary motivation for the study of mutual exclusion problem. Also the token ring networks(Farker and NewHall 1969, Farker and Larson 1972) constitute a popular approach to local area networks. Although token passing protocols, based on IEEE 802.4 and IEEE 802.5 standards, have been designed for both bus and ring topologies (IEEE 1982), we are particularly interested in token based control algorithms for various popular interconnection networks.

In distributed systems, processors, interconnected according to the adopted topology, are normally loosely coupled machines. In such systems, processing, control and databases are distributed over distinct processors. In *token bus* based local area networks, there is no guarantee of how much waiting time is required to gain the access right. To remove the nondeterministic behavior, an alternative approach involves in passing a token among the processors in the network. The owner of the token has the right to access the bus. Upon completion of the transmission, the token is passed to the next device based on some scheduling discipline (Duato *et* al. 2003). By restricting the maximum token holding time, the waiting time of a processor for token can be guaranteed.

The idea of token ring is a natural extension of token bus as the passing of the token forms a ring structure. IBM token ring supports bandwidths of both 4 and

16 Mbits/second based on coaxial cable. Fiber Distributed Data Interface (FDDI) provides a bandwidth of 100 Mbits/second using fiber optics as the transmission medium (Ross 1986, Bertsekas and Gallager 1992, Zhang *et* al. 2001, Duato *et* al. 2003). Because of the high speed and relatively insensitivity to physical size, FDDI can be used both as a backbone net for slower local area networks or as a metropolitan area network. The token based control strategy is used in a wide variety of communication problems. For example, data transmission in *token rings* (Andrews and Schulz 1982, Ross 1986, Stallings 1997) and access to the transmission medium in *bus networks* (Hajek 1991, Bertsekas and Gallager 1992) are solved by arranging the processors in a logical ring. Although an extensive amount of literature exists on the token based control strategy, little is known about the efficiency of this mechanism. In fact, with in the context of distributed computing, the research has mostly focused on the detection of token loss and on self stabilization aspects (Dijkstra 1974, LeLann 1977, Misra 1983, Burns and Pachl 1989, Israeli and Jalfon 1990, Huang and Chen 1993, Kakugawa and Yamashita 1995).

A bidirectional ring network is formed by permitting requests and the token to move in opposite directions in a ring network. A linear array is the simplest model among fixed connection networks to interconnect a set of n processors as shown in Figure. 2.1 for n=5 processors. Here, processor P_i, $i = 2, 3, \cdots, n-1$ is connected to its *left neighbor* P_{i-1} and its *right neighbor* P_{i+1} through a two-way communication link. Processor P_1 is connected to processor P_2 and processor P_n is connected to P_{n-1}. Processors P_1 and P_n are known as *boundary* processors and each has only one neighbor (Akl 1989, Wu 1999, Horowitz *et* al. 2001). If two

boundary processors are connected, then the linear array structure forms a ring network. Similarly, a ring network with a faulty processor forms a linear array.

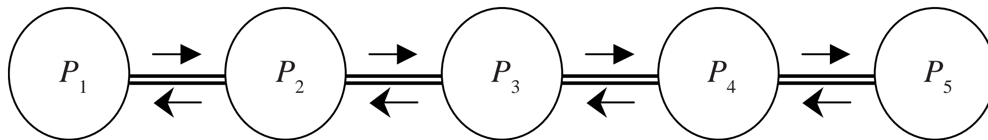

Figure 2.1: A linear array of n=5 processors

The token passing strategy has been widely used for solving the *mutual exclusion* problem (Maekawa 1985, Suzuki and Kasami 1985, Naimi and Trehel 1987, Raynal 1988, Raymond 1989-a, Raymond 1989-b, Makki 1994-a, Lynch 1996, Feuerstein *et* al. 1998, Reisig 1998, Wu 1999, Lodha and Kshemkalyani 2000, Sasaki 2001, Wu and Shu 2002), election problem (LeLann 1977, Huang 1993) and problems in hub polling systems (Bux 1984, Rego and Ni 1988, Bertsekas and Gallager 1992). All these problems use the same "hot-potato" procedure: *an entity holding token will pass it along the ring as soon as it no longer needs it* (IEEE 1982, Raynal 1988, Halsall 1992). According to this procedure, if a processor which did not request the token receives it, it will immediately forward it to the next processor. In this case, the unbounded number of token exchanges implies a negative consequence that an extensively large amount of communication (message exchanges) might be spent to manage a seldomly used resource. To overcome this negative consequence in which the request based token passing strategy, that allows the token to move only if it is informed of a processor asking the shared resource, was proposed (Suzuki and Kasami 1985, Feuerstein *et* al. 1998).

Feuerstein *et* al. (Feuerstein *et* al. 1998) have proposed two protocols Q and D for single shared resource allocation using request message based token control strategy in a fault-free unidirectional ring network in which requests and token move in single direction only. Algorithm Q requires n messages per request and service traffic $\frac{3n^2}{2}$. Algorithm D requires $2n$ messages per request and service traffic ($3n$ - 3). These algorithms do not require any piggyback information of the processors in the network. However message delivery is assumed to be guaranteed by underlying communication mechanism, but the transit time for messages is unpredictable. In contrast, when a request message carries routing information about source-destination identities, there are numerous token passing strategies that make use of shortest paths on which routing from source to destination processor could be made (Chalamaiah and Ramamurthy 1998, Chalamaiah 1999, Mukhopadhyaya and Sinha 1995).

The analysis of the proposed algorithms uses the two distinct measures namely *average number of messages per request* and *service traffic* described in the section 1.3.

In the next section, we present an algorithm D' for the allocation of a shared resource in a bidirectional ring network which is a powerful network structure suitable for high speed LANs. The message routing is reliable because of bi-directional movements over the processors. In this algorithm D', the additional check tour is not required as in Feuerstein *et* al (Feuerstein *et* al. 1998). Then, in section 2.2, the single shared resource allocation algorithm $L1$ presented for a linear array, provides better performance and improved service reliability for the processors to

access the single shared resource. This algorithm $L1$ provides a better lower bound with bi-directional message movements. We ensure that the proposed algorithms prevent starvation of the token and processors are served with in a maximum delay. At the end of this chapter, we brief the concluding remarks.

2.2 Allocation of a shared resource in rings

Let us consider n processors $P_1, P_2, \cdots, P_n$ that are arranged in a logically structured ring in which communications can be in either directions. For $i = 2, 3, \cdots, n-1$, processor P_i is connected to P_{i-1} and P_{i+1} and processor P_n is connected to P_1.

First we present the modified algorithm D' of Feuerstein *et* al.'s algorithm D. Here we assume that the token is allowed to pass through only in anticlockwise direction and requests are allowed to pass through clockwise direction. Also assume that each processor has a local variable M such that $M = 1$, if either a request has passed through the processor P_i or P_i has a pending request of its own and $M = 0$, otherwise. Processor P_i, willing the token, sends a request that passes through atmost (n - 1) processors to reach the token. Token may also be moving atmost (n - 1) processors to serve a request. Initially M value at each processor is assumed to be zero and the token is placed at any processor P_i. The behavior of a processor P_i is formally described as follows:

Algorithm - D'

/* This algorithm describes the single shared resource allocation in a bi-directional ring network. In this network, token moves in anticlockwise direction and requests move in clockwise direction */

int i // $1 \leq i \leq n$; the index of processor P_i

boolean M ; // takes value 0 or 1

(1) When P_i needs resource then

Begin

If P_i has token then it enters critical section;

Otherwise

if $M = 0$ then it sets $M = 1$ and sends the request

in clockwise direction to the next processor

else ($M = 1$) no message is sent

End

(2) When P_i receives a request then

Begin

If P_i has token then

it sets $M = 0$ and sends token to the next processor in

anticlockwise direction

If P_i does not have token then

if $M = 0$ then

it sets $M = 1$ and sends the request in clockwise direction

else ($M = 1$) no message is sent

End

(3) When P_i receives token then

Begin

If P_i has a pending request then it enters critical section in which

it sets $M = 0$. At the end of critical section, it passes token

to the next processor in anticlockwise direction

If P_i has no pending request then

if $M = 1$ then it sets $M = 0$ and sends token to the next processor

else ($M = 0$) it stops the token at P_i.

End

In the algorithm D of Feuerstein *et* al (1998), after serving all $(n-1)$ pending requests, token makes an additional check tour for honouring the requests generated later on. This check tour is compulsory even for a single request generation. But in the proposed algorithm D', token recognizes the requests in the processors on its way and serves them with one additional request movement. Hence the proposed algorithm D' does not require an additional check tour.

Theorem: 2.1

Algorithm D' satisfies all requirements of processors. It needs atmost $2(n-1)+1$ messages per request and its service traffic is $3(n-1)$.

Proof:

In algorithm D', request messages and token move in opposite directions.

First, we shall prove that the algorithm D' satisfies requests of all processors.

This algorithm consists of 3 steps.

In step (1), a processor, which is in need of shared resource, enters critical section if it has the token; otherwise, it generates a request message and forwards the request to the next processor in clockwise direction.

In step (2), as and when a processor receives a request, it checks the availability of the token. If the processor has token then, the token is sent to the next processor in anticlockwise direction. If the processor has no token then, the received request is forwarded to the next processor only if it has not already sent a request in clockwise direction; otherwise, the received request is stopped.

In step (3), whenever a processor receives token, it checks for a pending request.

If the processor has a pending request then, it enters critical section. At the end of critical section, token is passed to the next processor in anticlockwise direction. If the processor has no pending request of its own then token is passed to the next processor; otherwise token is stopped at that processor.

Therefore step (1) generates request messages; step (2) forwards the request messages and informs the token and step (3) makes the token to serve the processors with pending requests. This ensures that the algorithm serves all processors.

To prove that *the algorithm D' requires atmost* $2(n-1)+1$ *messages per request*, first consider the number of message exchanges per request:

In step (1), a generated request may advance over the next processor upto next $(n-1)$ processors. In step (2), the received request is forwarded only if it has not already sent a request; otherwise stopped if the processor has already sent a request.

Both steps (1) and (2) together require a total of atmost $(n-1)$ request message exchanges.

In step (3), token makes atmost $(n-1)$ moves to serve all pending requests and makes one move to come to rest. This requires atmost $(n-1)+1$ token movements.

Therefore, atmost $(n-1)+(n-1)+1 = 2(n-1)+1$ messages are exchanged per request.

Now we prove that *service traffic is* $3(n-1)$.

Let token be at processor P_i, and P_{i+1} generates the first request. Assume that all requests are generated only after the request generated by P_{i+1}. Now let us calculate the total number of messages exchanged before P_{i+1} gets the token.

In step (1), P_{i+1} generates request and forwards to P_{i+2}. In step (2), P_{i+2} receives the request of P_{i+1} and forwards it to P_{i+3}, if it has no pending request or stops if it has already sent a request. This implies a total of $(n-1)$ request message movements.

In step (3), token, on receiving a request, moves in anticlockwise direction and serves the processors which have pending requests. After serving, token moves to the next processor in anticlockwise direction. This implies that there are $(n-1)$ token movements. But as soon as the token leaves a processor, it may again generate a request message which may travel behind the token. This way atmost $(n-1)$ additional requests may be generated by the time token reaches P_{i+1}. This implies a total of $(n-1)+(n-1)$ message exchanges.

Thus the overall service traffic amounts to $(n-1)+(n-1)+(n-1) = 3(n-1)$ message exchanges in the worst case. ∎

2.3 Token-based control in linear arrays

Consider n processing elements arranged in the form of a linear array, which might result in from a ring with single faulty processor. Here also we use similar settings as stated in section 2.2, except that tokens move along one direction serving all pending requests until there is a pending request in that direction; otherwise, the token changes its direction if there is a pending request in the other direction. Request messages are sent using M value as above, by keeping track of the movement of token using a direction bit d. This direction bit d assumes 1, if token is passed along the right port and 0, if the token is passed along the left port. Initially, M and d assume 0 at all processors. Token has two flag bits T_l and T_r, which will be modified to record pending requests received from left or right respectively. The flag bit $T_{l(r)}$ assumes 1, if a request is received from left (right) port and 0, otherwise. Initially these flag bits T_l and T_r assume zero. The behavior of a processor P_i that executes algorithm $L1$ as follows:

Algorithm : $L1$

/* This algorithm allocates the single shared resource in a linear array - a ring with a faulty processor or link */

int i // $1 \leq i \leq n$; the index of processor P_i

boolean M, d, T_l, T_r; // each takes value 0 or 1

(1) When P_i needs resource then

Begin

If P_i has token then it enters critical section

If P_i does not have token then

if $M = 0$ then set $M = 1$ and

if $d = 0$ then send the request along the left port

else $(d = 1)$ then send the request along the right port

else$(M = 1)$ no message is sent.

End

(2) When P_i receives request then

Begin

If P_i has token then

If $i = 1$ (or $i = n$) then set $d = 1(0)$; $T_{r(l)} = 1$; $M = 0$ and

pass token along right (left) port of $P_1(P_n)$.

If $1 < i < n$ then

If request is from left port then set $T_l = 1$; otherwise set $T_r = 1$;

If $d = 0$ then

if $T_r = 1$ then set $d = 1$; send token along right port;

else$(T_r = 0)$ set $M = 0$ and pass token along left port of P_i.

else $(d = 1)$

if $T_l = 1$ then set $d = 0$ (if $T_r = 0$); $M = 0$ and

pass token along left port

else set $M = 0$ and pass token along right port of P_i.

else (if P_i has no token then)

if $M = 0$ then

if d=1 then

if request is received from right port then ignore the request;

else set $M = 1$ and send request to right port

else(if d=0 then)

if request is received from left port then ignore the request;

else set $M = 1$ and send request to left port

else (if $M = 1$ then)

stop the request at P_i itself.

End

(3) When P_i receives token then

Begin

If P_i has a pending request then it enters critical section;

at the end of critical section, pass token as follows:

if $i = 1$(or n) then set $T_{l(r)} = 0$; $M = 0$ and

if $T_{r(l)} = 1$ then set $d = 1(0)$ and pass token along right(left) port

otherwise token stays at $P_1(P_n)$.

else($1 < i < n$)

if $d = 0$ then set $d = 1$;

if $T_l = 0$ then set $M = 0$; else set $M = 1$;

and pass token along right port of P_i

else set $d = 0$;

if $T_r = 0$ then set $M = 0$; else set $M = 1$

and pass token along left port of P_i.

else (if P_i has no pending request then)

if $M = 0$ then

if $d = 0$ then set $T_r = 0$

if $T_l = 1$ then pass token along left port; else token stays at P_i

else ($d = 1$) set $T_l = 0$

if $T_r = 1$ then pass token along right port; else token stays at P_i

else ($M = 1$) if $d = 0$[or 1] then

if $T_{l[or\ r]} = 0$ then set $M = 0$; $d = 1$[or 0] and

send token along right[or left] port

End

Theorem: 2.2

Algorithm $L1$ serves all requests and requires $2(n-1)+1$ messages per request and service traffic is $3(n-1)$.

Proof:

First we prove that the algorithm $L1$ serves all requests.

This algorithm consists of 3 steps.

In step (1), a processor, which is in need of shared resource, enters critical section if it has token; otherwise, it generates a request and forwards the request only if it has not already forwarded a request.

In step (2), as and when a processor receives a request, it checks the availability of token. If the processor has token then, the token is sent along appropriate port based on the values of direction bit and token flag bits. If the processor has no token then the received request is stopped if it has already sent a request; otherwise, based on the value of the direction bit, the received request is either ignored or forwarded along the appropriate port.

In step (3), whenever a processor receives token, it checks for a pending request. If the processor has a pending request then, it enters critical section. At the end of critical section, token is either passed along appropriate port; otherwise token stops at that processor. If the processor has no pending request of its own and if it has a pending request of other processor then token is either forwarded along left or right port based on the values of direction bit and token flag bits, or token stops at that processor itself.

Thus, step (1) generates a request; step (2) forwards the request and informs the token and step (3) passes the token to serve the processors with pending requests.

These three steps ensure that the algorithm $L1$ serves all requests.

Now we prove that the number of message exchanges per request amounts to $2(n-1)+1$ in the worst case.

As per step (1), in the absence of token, a message generated for shared resource may be forwarded over atmost $(n-1)$ processors before it reaches the token. In step (2), the received request is either forwarded or stopped as already a request has been sent.

Both step (1) and (2) takes atmost $(n-1)$ message exchanges.

Once the token is informed, the token may travel in its direction or in reverse direction as per step (3). In any case, the total number of token exchanges for a request shall be atmost $(n-1)+1$. This is due to the fact that if request message requires less than $(n-1)$ exchanges to inform the token, the token may have to travel in the same direction as long as there is pending request. This may lead to token reaching the end processor in the worst case and then changing its direction. This proves that in the worst case, a request requires atmost $2(n-1)+1$ message exchanges.

Next we prove that *service traffic is* $3(n-1)$.

Let token be at any one of the end processors. Now by step (1) and step (2), atmost $(n-1)$ messages are exchanged. Then token starts serving the processors with pending request in one direction and stops either at the other end processor or at the intermediate processor, as there is no request pending in that direction as well as in the reverse direction. This requires atmost $(n-1)$ token movements.

In the mean time, serviced processors may again generate request messages. This implies atmost $(n-1)$ request movements.

Thus the overall service traffic amounts to $(n-1)+(n-1)+(n-1) = 3(n-1)$ message exchanges per request. ∎

Example:

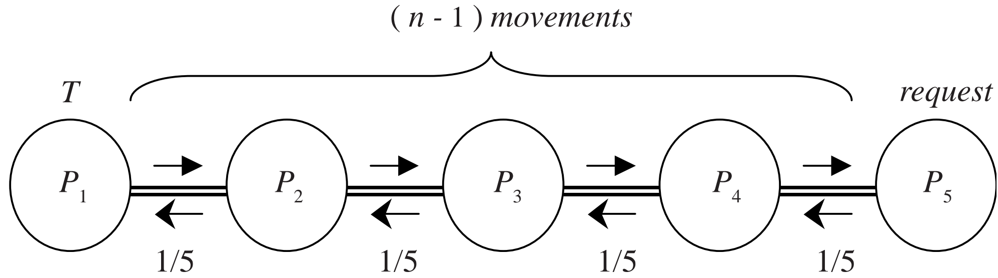

Figure 2.2: Single shared resource allocation in a linear array

To illustrate the performance of the proposed algorithm $L1$, consider a linear array of $n = 5$ processors as shown in Figure 2.2. We assume that the token resides at processor P_1 and in the worst case, the processor P_5 generates a request to access the single shared resource. The generated request from the processor P_5 advances over processors only if M value of that processor is zero; otherwise the request would be stopped as already a request had been sent. So the generated request from processor P_5 requires $(n - 1) = 4$ movements to inform the token in the worst case. Similarly as the token moves from the processor P_1 to P_5, it serves all pending requests in its way, if any request is generated in the mean time. This type of movements require $(n - 1) = 4$ movements. Finally additional $(n - 1)$ message movements are required as processors may generate request messages after the token had left that processor. Thus, the algorithm serves the request of the processor P_5 and requires 8 message exchanges. Hence, in the worst case, the service traffic amounts to 12 message exchanges.

2.3 Conclusion

We have proposed two algorithms, one for a bidirectional ring network and another for a linear array, using request based token passing strategy. The first algorithm D' for single shared resource allocation in a bidirectional ring network does not require an additional check tour as in the algorithm D of Feuerstein *et* al. (Feuerstein *et* al. 1998). The second algorithm $L1$ describes the bidirectional message and token passing strategy for shared resource allocation in the linear array. A skewed tree (Horowitz *et* al. 2001), it may be skewed to either left or right, is similar to a linear array structure and thus the algorithm $L1$ could be applicable to a binary tree skewed to either left or right regardless of its degree.

Chapter 3

Shared resource allocation in ring extension topologies

3.1 Introduction

A simple ring network is highly vulnerable to faults, eventhough it has several advantages and can easily be applicable for Local Area Networks. There is no way to bypass the faults unless they have to be removed or additional links across the processors should be included. Thus variations of ring networks had been introduced by suitably modifying the connectivity of links across the processors. In multicomputer systems, regular networks of low degree having small maximum message path lengths are of interest. Also a processor with two buses or degree 2 limits the regular networks to cycles (Liu 1978).

There are regular graphs of degree 3 in which the graph diameter is less in comparison to degree 2 case. Chordal rings and distributed loops are examples of the derivatives of a ring network. The chordal ring network offers increased reliability and fault tolerance compared to the simple ring network due to its multiple routing paths. One obvious way of finding a shortest path is simply to

traverse through the chords whenever necessary. In fact, there is more than one shortest path for most pairs of processors in a chordal ring network. Arden and Lee (Arder and Lee 1981) proposed a message passing algorithm with a routing record which is initially computed at the source processor. This routing record retrieves all the shortest composite paths between the source and destination processors. Then several shortest path algorithms with known source and destination identities are proposed for distributed loops(Mukhopadhyaya and Sinha 1995, Chalamaiah and Ramamurthy 1998, Chalamaiah 1999).

In this chapter, we consider the allocation problem of a single shared resource among a set of asynchronous processors that are arranged in the derivatives of a ring network like touching rings and intersecting ring networks in which one or more processors are assumed to be of degree more than two. As we assumed earlier, the identities of the processors are not included to request messages or the token. We propose algorithms based on *request-message-based* token passing strategy, as described in section 1.3, in which token movements are controlled by the requests generated for the single shared resource.

The analysis of the proposed algorithms uses the two distinct measures namely *average number of messages per request* and *service traffic* described in the section 1.3. In the next section, we present algorithms for the derivatives of a ring network. Then, in section 3.3, we present an algorithm for fault-free intersecting rings. Then section 3.4 carries concluding remarks.

3.2 Allocation in touching rings network

Consider a touching ring network in which n processors are logically structured in two subrings that touch at a common processor through which routing of the requests as well as token from one subring to another subring takes place either in an unidirectional or in a bidirectional way. In this network, the processor next to the common touching processor is indexed as 1 and the indexing runs over the remaining processors in clockwise direction; crossing the common touching processor; then again running over the processors in the next subring in clockwise direction and finally ends up with the processor before the common processor. The role of the common processor is significant as it has a special mechanism to switch the token as well as request messages into the subrings appropriately according to the receipt of request messages as well as token from the subrings.

We assume the common touching processor in unidirectional touching rings network as $P_{\lceil \frac{n}{2} \rceil}$. Incoming links to the common processor are from $P_{\lceil \frac{n}{2} \rceil - 1}$ and P_n and outgoing links are to P_1 and $P_{\lceil \frac{n}{2} \rceil + 1}$. Each processor P_i $[i = 1, 2, 3, ..., n]$ has a local variable M_p which is used to stop further movements of the following request messages. M_p assumes the value 1 or 0 as defined in section 2.2. Now assume that the common processor $P_{\lceil \frac{n}{2} \rceil}$ has a token locator T_l to trace the token location in the subrings and a request recorder V_r that helps to record the pending requests of one subring when token has passed through the other subring. The value for T_l is assigned to 0, if token has passed to the processor in the first subring from the common processor and 1, otherwise.

V_r is assigned to 1, if a request is received from P_n to $P_{\lceil \frac{n}{2} \rceil}$ [similarly from $P_{\lceil \frac{n}{2} \rceil - 1}$ to $P_{\lceil \frac{n}{2} \rceil}$] when $T_l = 0(1)$ and 0, otherwise. Initially the values of M_p and V_r are assumed to be zero and token is placed at the common processor. It can be observed that whenever $T_l = 0(1)$ then V_r indicates the receipt of a request message received from second (respectively first) subring. Token maintains a two cell flag bit structure and sets as and when it recognizes either V_r=1 at $P_{\lceil \frac{n}{2} \rceil}$ or the receipt of a request message. The cells are indexed as T_1 and T_2 which takes 1, if a request is received from first and second subrings respectively and 0, otherwise. The value of the two cell flag bit helps token to move through the processors in the next subring whose M_p might be zero. The token is directed to move from the common processor according to the value of M_p and V_r. Initially the the token is placed at the common processor and M_p, V_r, T_l, T_1 and T_2 are assumed to be zero. Token stops only if the values of M_p, T_1 and T_2 are zero. The algorithm is described below:

Algorithm : UTRN

(a) When P_i needs resource then

If P_i has token then it enters the critical section.

If P_i has no token then

if P_i is the common processor then

if T_l =0 (1) then

if $M_p = 0$ then set $M_p = 1$ and

send the request message to P_1 ($P_{\lceil \frac{n}{2} \rceil + 1}$)

else($M_p = 1$) stop the request at P_i

else (if P_i is not the common processor then)

if $M_p = 0$ then

set $M_p = 1$ and send the request message to the next

processor in the clockwise direction

else no request message is sent.

(b1) When P_i receives a request message

If P_i has token then

if P_i is the common processor then

if the request message is received from $P_{\lceil \frac{n}{2} \rceil -1}$ (P_n) then

set $T_l = 0(1)$; set $T_1(T_2) = 1$; set $M_p = 0$ and

send token to the processor $P_1(P_{\lceil \frac{n}{2} \rceil +1})$ in the current subring

else(if P_i is not the common processor then)

if $1 \leq i < \lceil \frac{n}{2} \rceil$ then set $T_1 = 1$; else ($\lceil \frac{n}{2} \rceil < i \leq n$) set $T_2 = 1$;

send token to the processor P_{i+1} in clockwise direction.

(b2) When P_i receives a request message and has no token then

If P_i is the common processor then

if the request message is received from $P_{\lceil \frac{n}{2} \rceil -1}$ (P_n) then

if $T_l = 0(1)$ then set $M_p = 1$;

if $V_r = 1$ then stop the request at P_i;

else($V_r = 0$) send request to P_1 ($P_{\lceil \frac{n}{2} \rceil +1}$) in the current subring.

else(if $T_l = 1(0)$ then) set $V_r = 1$.

if $M_p = 1$ then stop the request message at P_i;

else (if $M_p = 0$ then)

send request to next processor $P_{\lceil \frac{n}{2} \rceil + 1}$ (P_1) in the next subring;

else (if P_i is not the common processor then)

if $M_p = 0$ then set $M_p = 1$ and send the request to P_{i+1};

else stop the request at P_i.

(c1) When P_i receives token then

If P_i has a pending request message then

if P_i is the common processor then

if $V_r = 0$ then enter critical section; At the end of critical

section, set $M_p = 0$; set $T_{1(2)} = 0$; set $T_l = 0(1)$;

and send token to P_1 ($P_{\lceil \frac{n}{2} \rceil + 1}$) in the current subring

else(if $V_r = 1$ then)

if token is received from $P_{\lceil \frac{n}{2} \rceil - 1}$ (P_n) then

enter critical section; At the end of critical section,

reset $M_p = 0$; set $T_{2(1)} = 1$; set $T_l = 1(0)$; set $V_r = 0$ and

send token to $P_{\lceil \frac{n}{2} \rceil + 1}$ (P_1) in the next subring

else (if P_i is not the common processor then)

enter critical section; reset $M_P = 0$;

if $1 \leq i < \lceil \frac{n}{2} \rceil$ then set $T_1 = 0$; otherwise set $T_2 = 0$;

at the end of critical section, send token to the next

processor P_{i+1} in the current subring.

(c2) When P_i receives token and has no pending request then

if P_i is the common processor then

if $V_r = 1$ then

if token is received from $P_{\lceil \frac{n}{2} \rceil - 1}$ (P_n) then

set $T_l = 1(0)$; reset $V_r = 0$;

if $M_p = 1$ then set T_1=$T_2 = 1$; otherwise set $T_2(T_1) = 1$;

and send token to $P_{\lceil \frac{n}{2} \rceil + 1}$ (P_1) the next subring

else (if $V_r = 0$ then)

If $M_p = 1$ then set $M_p = 0$; and send token

to the next processor in the current subring

else (if $M_p = 0$ then)

if T_1 AND $T_2 = 0$ then stop token at P_i

else send token to the next processor in the current subring

else (if P_i is not the common processor then)

if M_p OR T_1 OR T_2= 1 then reset $M_p = 0$ and pass token to P_{i+1}

else stop the token at P_i itself.

Theorem : 3.1

The algorithm UTRN serves all the request messages.

Proof:

We assume that token is initially placed at the common processor. Each request message will eventually reach the token and informs it to access the shared resource. On receipt of a request message, token moves through the appropriate subring by setting T_l value at the common processor accordingly. Meantime V_r and M_p are maintained at the common processor to record the request messages received from

another subring and the current subring respectively.

Now let token be at any one of the processors in the subrings. Then the common processor when it receives a request message checks the values of T_l, V_r and M_p and sets them as in steps (b1) and (b2) and then the request message is sent to the corresponding subring based on the availability of the token. Whenever token reaches the common processor, it checks the values of V_r and M_p. If there is a pending request message of the next subring then token is forwarded to the next subring; otherwise token is forwarded to the next processor in the current subring. As and when token switches over subrings, V_r and M_p are maintained appropriately. Token moves through the processors until T_1 AND $T_2 = 0$ and $M_p = 0$. Thus request messages are not lost and token is aware of all request messages. Thus the algorithm UTRN serves all the request messages. ∎

Theorem : 3.2

The algorithm UTRN requires atmost $(2n-1)$ messages per request and service traffic is bounded by $(5n-2)/2$

Proof:

Assume that the token is at the common processor. Now suppose that a request message has been originated at some processor in a subring. In fact, regardless of n, the total number of processors, atmost $(\frac{n}{2}-1)$ movements, for each subring, are required to inform the token in the worst case. If token has passed through the first subring then the request coming from second subring moves to the first subring if $M_p = 0$ at the common processor, else the request message stops. Meantime,

atmost $(n-1)$ processors may generate request messages. Thus in addition, token may have to move over at most n processors. Thus the number of messages per request amounts $(2n-1)$ in the worst case.

To prove the second measure, two types of request messages may be sent between the time in which P_i generates a request message and the time in which it has been served. The first type includes the request messages that require atmost $(n-1)$ movements to inform the token. In response to this request, token makes atmost $\frac{n}{2}$ movements for switching it into the next subring; $\frac{n}{2}$ movements for crossing that subring and $\frac{n}{2}$ movements for a new request generated later on. Thus, in total $\frac{3n}{2}$ messages are needed for the token to reach the processor with a pending request. The last $\frac{n}{2}$ movements guarantee that token is aware of all the request messages of one subring when token is passing through another subring. Thus service traffic in the worst case amounts to $\frac{5n-2}{2}$. ∎

Instead of assuming unidirectional movement over the touching ring network, we can assume opposite directional movement for both the token as well as request messages. Such a touching ring network is termed as a *Bidirectional touching ring network* in which token can move in anticlockwise and request messages flow in clockwise direction. The algorithm UTRN still holds for a bidirectional touching ring network, provided if we keep track of the change in the direction of request messages that are allowed to move only in the clockwise direction. The resulting algorithm with bidirectional movements, we call it as BTRN, ensures the early service for the requests even in the worst case.

An immediate result follows in the sequel.

Theorem : 3.3

Algorithm BTRN for a bidirectional touching ring network serves all the request messages, requires atmost $2(n-1)+1$ messages per request and service traffic is bounded by $(3n-2)$.

Proof:

As the request messages move in anticlockwise direction and token in clockwise direction, no request message would be skipped by the token and hence all request messages are served. Now to prove the first measure, we consider the opposite movements of the token as well as request messages. Here the worst case number of message exchanges will be atmost $2(n-1)+1$ as the processor holding the token and the processor initiating the request may be adjacent. Now we prove the second measure. As the token moves in clockwise direction and request messages in anticlockwise direction, it can easily sensor and serve all pending requests in its way to the destination. Token stops moving only when it recognizes a processor with no pending request message. In the mean time, the processors which are served recently may also generate requests for the shared resource. There could be atmost $(n-1)$ such requests in the worst case. Hence, in the worst case, the service traffic amounts to $(3n-2)$. ∎

The algorithm UTRN can also be generalized for n-touching rings network $(n, m[k])$ where n is the number of processors in the common ring and $m[k]$ is the number of subrings each with k processors each and $k \geq 2$ [excluding the touching

processor]. The n-touching rings network is formed in such a way that each subring with k processors is connected with one processor in the common ring network. Thus there are totally $(n + mk)$ processors.

We assume a fault-free n-touching ring network with $(n+mk)$ prcoessors. Now we can extend the proposed algorithm UTRN to the n-touching rings network by considering each subring with the remaining part of the network and for each processor in the common ring network. By this way, each processor in the common ring assumes all the variables as in UTRN and operates in parallel. Still we follow the basic assumptions that they have neither a shared memory nor a common clock and their speeds are not related. Thus the generalized algorithm requires $(n+mk)$ messages per request and $(3n+2mk-1)$ service traffic in the worst case. It can be quaranteed that this generalized algorithm works perfectly for variable k also. But as n and k grow, the transmission delay for serving the pending requests would be unavoided.

3.3 Allocation in intersecting rings network

In this section, we consider an intersecting rings network in which $N[= (n+m)]$ processors are logically arranged in two rings each with n and m processors that intersect at two common processors through which routing of the request messages as well as token from one subarc of one ring to another subarc of another ring takes place. In this network, the processor next to the common processor is indexed as P_1^2[or P_1^1], then indexing runs over the remaining processors in the clockwise direction, crossing a common processor; running over the processors in the next

subarc of the same ring in clockwise direction and finally ends up with the processor before the next common processor. Similarly the second ring is numbered starting from the processor next to the common processor which is part of the first ring. In this scheme, one common processor is counted in the first ring and the other in the second ring.

We consider two common processors say P_f^1 and P_s^2 in a fault-free intersecting rings network. The network is assumed to be bidirectional. The first ring consists of an arc l_1 with processors P_1^1, P_2^1, P_3^1 ,...,P_{f-1}^1; the common processor P_f^1 and an arc l_2 with processors $P_{f+1}^1, P_{f+2}^1, ..., P_n^1$ $((f-1) > (n-f))$. Similarly the second ring consists of an arc r_1 with processors $P_1^2, P_2^2, P_3^2, ..., P_{s-1}^2$; the common processor P_s^2 and an arc r_2 with processors $P_{s+1}^2, P_{s+2}^2, ..., P_m^2$ $((s-1) > (m-s))$. The superscripts 1 and 2 denote the first and second ring respectively. We assume that the request messages move in the clockwise direction and token moves in the anticlockwise direction. Each processor P_i has a local variable M_p, as defined in section 3.2, to record the movements of request messages through it.

Each common processor has a token locator $L_{f[\text{or } s]}$ (L_f for P_f^1 and L_s for P_s^2) which is used to trace the arc through which the token has been sent and request indicators $V1_f$ and $V2_f$ [or $V1_s$ and $V2_s$] which are used to record the request messages received from P_{f-1}^1 and P_m^2 [or P_{s-1}^2 and P_n^1]. The token indicator $L_{f[\text{or } s]}$ is set to 1, if token has been passed through P_{f-1}^1 [or P_{s-1}^2] and 0, if token has been passed through P_m^2 [or P_n^1]. $V1_f(V2_f)$ assumes 1, if a request is received from P_{f-1}^1 (P_m^2) and 0, otherwise. Similarly $V1_s(V2_s)$ assumes 1, if a request is received from $P_{s-1}^2(P_n^1)$ and 0, otherwise. Token maintains a four cell flag bit structure, say

T_{l_1}, T_{l_2}, T_{r_1} and T_{r_2}. When token arrives at one of the common processors, the value of $T_{l_1(r_2)}$ [or $T_{r_1(l_2)}$] is set to 1, if there is a pending request received from P^1_{f-1} (P^2_m)[or P^2_{s-1} (P^1_n)] and 0, otherwise. The values of the these flag bit cells help the token to move through the processors in the subarcs. Initially the values of M_p, $L_{f[\text{or } s]}$, $V1_{f[\text{or } s]}(V2_{f[\text{or } s]})$ and $T_{l_1(r_2)}$[or $T_{r_1(l_2)}$] are all assumed to be zero and token is placed at any one common processor. Token stops moving only if the values of M_p, T_{l_1}, T_{l_2}, T_{r_1} and T_{r_2} are zero. The description of the algorithm : InteRN - B is as follows:

Algorithm : InteRN - B

(a) When P^1_i[or P^2_i] needs resource then

  If P^1_i[or P^2_i] has token then enter the critical section.

  If P^1_i[or P^2_i] has no token then

    If $i = f$[or s] then

      if $M_p = 0$ then set $M_p = 1$ and send the request message

        to the processor P^1_{f+1}[or P^2_{s+1}] if $L_{f[\text{or } s]} = 1(0)$

      else stop the request at P_i

    else if $M_p = 0$ then set $M_p = 1$ and send the request to P^1_{i+1}[or P^2_{i+1}]

      else (if $M_p = 1$ then) no request message is sent.

(b1) When P^1_i[or P^2_i] receives a request message then

  If P^1_i[or P^2_i] has token then

    If $i = f$[or s] then

      if the request is received from P^1_{f-1} (P^2_m) [or $P^2_{s-1}(P^1_n)$] then

set $L_{f[\text{or } s]} = 1(0)$; $M_p = 0$; $T_{l_1(r_2)}$[or $T_{r_1(l_2)}$] $= 1$

and send token to $P^1_{f-1}(P^2_m)$[or $P^2_{s-1}(P^1_n)$]

else (if i is neither f nor s then)

if P^1_i[or P^2_i] $\in l_1(r_2)$[or $r_1(l_2)$] then

set $T_{l_1(r_2)}$ [or $T_{r_1(l_2)}$] $= 1$ and

send token to $P^1_{i-1}(P^2_{i-1})$[or $P^2_{i-1}(P^1_{i-1})$]

(b2) When P^1_i[or P^2_i] receives a request message then

If P^1_i[or P^2_i] has no token then

if $i = f$[or s] then

if the request is received from P^1_{f-1} (P^2_m) [or $P^2_{s-1}(P^1_n)$] then

set $V1_f(V2_f)$[or $V1_s(V2_s)$] $= 1$;

if M_p OR $V2_f(V1_f)$[or $V2_s(V1_s)$] $= 1$ then

stop the request message at P^1_i[or P^2_i]

else (M_p AND $V2_f(V1_f)$[or $V2_s(V1_s)$] $= 0$ then)

send the request message to P^1_{f+1}[or P^2_{s+1}]

else (if i is neither f nor s then)

if $M_p = 0$ then set $M_p = 1$ and send the request to P^1_{i+1}[or P^2_{i+1}]

else ($M_p = 1$) stop the request message at P^1_i[or P^2_i].

(c1) When P^1_i[or P^2_i] receives token and has a pending request then

enter the critical section; reset $M_p = 0$;

if $i = f$[or s] then

if token has been received from P^2_1 [or P^1_1] then

set $V1_s$[or $V1_f$] $= 0$;

if $V2_s$[or $V2_f$] $= 1$ then set T_{r_2}[or T_{l_2}] $= 1$; $V2_s$[or $V2_f$] $= 0$;

else (if token has been received from P^1_{f+1} (P^2_{s+1}) then)

reset $V2_f$[or$V2_s$]=0;

if $V1_f$[or $V1_s$] $= 1$ then set $T_{l_1[\text{or } r_1]} = 1$; $V1_f$[or $V1_s$] $= 0$;

and send token according to selectPath()

else (if i is neither f nor s then)

if P^1_i[or P^2_i] $\in l_1(l_2)$[or $r_1(r_2)$] then reset $T_{l_1(l_2)}$[or $T_{r_1(r_2)}$] $= 0$

and send token to P^1_{i-1}[or P^2_{i-1}] in anticlockwise direction.

(c2) When P^1_i[or P^2_i] receives token then

If P^1_i[or P^2_i] has no pending request message then

if $i = f$[or s] then

if token is received from P^2_1[or P^1_1] then

reset $V1_s$[or $V1_f$] $= 0$;

if $V2_f$[or $V2_s$] $= 1$ then reset $T_{r_2[\text{or } l_2]} = 1$; $M_p = 0$;

send token according to selectPath();

else [if $V2_f$[or $V2_s$] $= 0$ then]

if T_{l_1} AND T_{l_2} AND T_{r_1} AND $T_{r_2} = 0$ then

stop token at P^1_i[or P^2_i]

else send token according to selectPath();

else (if token is received from P^1_{f+1}[or P^2_{s+1}] then)

reset $V2_f$[or $V2_s$] $= 0$;

if $V1_f$[or $V1_s$] $= 1$ then reset T_{l_1}[or T_{r_1}] $= 1$; $M_p = 0$ and

send token according to selectPath();

else [if $V1_f$[or $V1_s$] $= 0$ then]

if T_{l_1} AND T_{l_2} AND T_{r_1} AND $T_{r_2} = 0$ then

stop token at P_i^1[or P_i^2]

else send token according to selectPath();

else (if $i \neq f[or\ s]$ then)

if $M_p = 1$ then reset $M_p = 0$; and send token

to the processor P_{i-1}^1[or P_{i-1}^2] in anticlockwise direction.

else (if $M_p = 0$ then)

if T_{l_1} AND T_{l_2} AND T_{r_1} AND $T_{r_2} = 0$ then

stop the token at P_i^1[or P_i^2] itself

else send the token to the next processor P_{i-1}^1[or P_{i-1}^2].

Procedure selectPath();

/* This procedure selectPath() sends the token through an appropriate outgoing link alternatively if there is a pending request.*/

if $L_{f[\text{or } s]} = 0$ then

if $T_{l_1[\text{or } r_1]} = 1$ then set $L_{f[\text{or } s]} = 1$ and send token to P_{f-1}^1[or P_{s-1}^2]

else send token to P_m^2[or P_n^1]

else (if $L_{f[\text{or } s]} = 1$ then)

if $T_{r_2[\text{or } l_2]} = 1$ then set $L_{f[\text{or } s]} = 0$ and send token to P_m^2[or P_n^1]

else if $T_{l_1[\text{or } r_1]} = 1$ then send token to P_{f-1}^1[or P_{s-1}^2]

else set $L_{f[\text{or } s]} = 0$ send token to P_m^2[or P_n^1]

Theorem : 3.4

The algorithm InteRN - B serves all request messages generated for the single shared resource.

Proof:

Let the token be initially at any one of the common processors [either at P_f^1 or at P_s^2]. Each request message, moving in clockwise direction will eventually reach the token and informs it to access the single shared resource. After receiving a request message, token moves in anticlockwise direction through the appropriate outgoing subarc of any one of the rings by setting the token locator L_f[or L_s] value in the common processor. Meantime $V1_f$, $V2_f$, $V1_s$, $V2_s$ and M_p values are maintained at the common processors to record the request messages received from two arcs of the different rings respectively. This would be achieved by a switching subsystem of the respective common processor. Whenever token receives a request message at a common processor or reaches a common processor, it will reset $T_{l_1}(T_{r_2})$[or $T_{r_1}(T_{l_2})$] according to the values of local variables $V1_f$, $V2_f$, $V1_s$, $V2_s$ and M_p. The switching subsystem records the receipt of request messages from two subarcs and hence the receipt of request messages are identified.

On the other hand, assume that token is at any one of the processors in any one of the subarcs of a ring. The common processor, when it receives a request message, checks the values of $L_{f[\text{or } s]}$, $V1_f$, $V2_f$, $V1_s$, $V2_s$ and M_p and sets them accordingly as in steps $(b1)$ and $(b2)$. Then the switching subsystem of the respective common processor forwards the request message in clockwise direction to the corresponding subarc based on the direction through which the token has

already been sent. In contrast, whenever token reaches the common processor in anticlockwise direction, it immediately checks the values of $V1_f$, $V2_f$, $V1_s$, $V2_s$ and M_p. After checking these values, procedure selectPath() is performed for selecting an appropriate outgoing link. As and when token switches over the subarcs, $V1_f$, $V2_f$, $V1_s$, $V2_s$ and M_p are dynamically maintained. Token moves through the processors until all cell entries of the flag bit cells and M_p are zero. Thus the request messages generated are not lost and token is aware of all request messages. Thus the algorithm InteRN - B serves all requests. ∎

Theorem : 3.5

The algorithm InteRN-B requires atmost N messages per request and service traffic is bounded by $2N$.

Proof:

To prove the first measure, we look at the opposite movements of the token as well as request messages. The associated variables are maintained at each processor. The worst case, in which the token is placed at a processor in the largest subarc of the first ring and the request message is generated at a processor in the largest subarc of the second ring, requires atmost $(n + m) = N$ movements necessary to satisfy a sequence of requests. These N movements include request messages generated by the rest of the processors. Hence the average number of messages per request amounts to N in the worst case. Next we prove the second measure, service traffic. As the token moves in anticlockwise direction and request messages in clockwise direction, token can easily sensor and serve all pending requests in its way to the destination. These request movements amount to N. In

the worst case, token additionally makes $(n + m) = N$ movements. Hence service traffic amounts to $2N$, which is a tight bound in the worst case. ∎

3.4 Conclusion

In this chapter, we have considered the problem of controlling the allocation of single shared resource in ring extension topologies like touching rings network and intersecting rings network using request based token control strategy. We have first presented an algorithm UTRN for the allocation of a shared resource in an unidirectional touching ring network in which communication takes place between two rings only at the common touching processor. This could be extended to a bidirectional touching ring network as well as multiple touching rings network. Finally we have presented an algorithm InteRN - B for bidirectional intersecting rings network with two arbitrarily intersecting rings. Also this algorithm could be extended to the possible architectures of more intersecting rings. But the communication delay would be higher as the total number of the intersecting rings increases. It would be quite interesting to look for similar problems in other augmented rings.

Chapter 4

Allocation in interconnected networks

4.1 Introduction

Many problems involving replicated data, atomic commitment, distributed shared memory and others require that a resource be allocated to a single process at a time in an interconnection network (Agrawala and El Abbadi 1991). Ring and mesh topologies are commonly used simple interconnection networks. A mesh structure provides lower message complexity and delay complexity because of more adjacent communication links in the network.

Chaudhuri and Karatta proposed a distributed mutual exclusion algorithm among n processors in a three dimensional mesh and the message complexity of this algorithm is $O(n^{\frac{1}{3}})$ per mutual exclusion invocation (Chaudhuri and Karatta 1998). Saxena and Gupta presented a token based mutual exclusion algorithm for arbitrary topologies (Saxena and Gupta 1999). This algorithm makes use of the network topology and site information held by each node to achieve an optimal performance. Also this algorithm reduces the delay in accessing the critical

section by allowing the token to serve the requesting processors which fall-on-route to its destination. In heavy load systems with n processors, the worst case scenario occurs when $(n - 1)$ processors are simultaneously requesting access to the critical section. The selection process, that points the next processor to be served, should be done in some fair manner if the response time is to be minimized. Greenwood (Greenwood 1995) showed a selection technique, using counters (Ricart and Agrawala 1983) and time stamps (Raynal 1986), always hardly achieves equity. He has also proposed a priority based selection method, in which a higher priority is given to processors that need frequent access to the critical section.

Aiello *et* al. have studied three augmentations of ring networks, namely chordal rings, express rings and multi-rings that are intended to decrease ring's diameter significantly while increasing its structural complexity only modestly (Aiello *et* al. 2001). A chordal ring is obtained from a ring network by adding non-crossing "shortcut" edges between a pair of nodes. An express ring is obtained from a chordal ring by orienting its shortcut chords in either the clockwise or counter clockwise sense, thereby turning the chords into arcs of the ring. Similarly a multi-ring is obtained from a ring network by appending subsidiary rings to edges of the ring and recursively, to the edges of subsidiary subrings. Multi-rings are a variation on the theme of multi-level ring interconnection networks such as one finds in the KSR1 multiprocessor (Burkhardt *et* al. 1992); they are also identical with the SONET (Synchronous Optical NETwork) multi-rings (Cosares 1992) that are so important in the realm of communication networks.

In this chapter, we have proposed single shared resource allocation algorithm for two interconnected rings network using request based token passing strategy. Then we have proposed the resource allocation algorithm for single side wrap around mesh. Then we have proposed single shared resource allocation algorithms among processors in single side wrap around mesh and regular mesh (Prasath and Thangavel 2003).

In the next section, we present the topological structure of the interconnection networks. Then section 4.3 presents the algorithm for shared resource allocation in interconnected rings. In section 4.4, we present an algorithm for single shared resource allocation in single side wrap around mesh network. In section 4.5, a bidirectional token passing algorithm for shared resource allocation in a regular mesh is presented. Finally, section 4.6 carries concluding remarks.

4.2 The model of communication networks

In this section, we describe the network structure of interconnected rings network, single side wrap around mesh and regular mesh. In an interconnected rings network, processors are interconnected in such a way that processor P_i in the first subring is connected with processor $P_{i+\frac{n}{2}}$ by a bidirectional link. We assume unidirectional movements for requests and token, over the ring edges and bidirectional movements over the edges connecting the two subrings. In an interconnected rings network, each processor in the second subring has a special mechanism to sensor the direction from which the token has been received so as to send the token from the processor in the second subring to the processor in the first subring.

Then we consider a single side wrap around mesh in which $n[= mk]$ processors are arranged in k rings each with m processors that are interconnected by a bidirectional channel in such a way that processor P_i^j $(i = 1, 2, 3, ..., k$ and $j = 1, 2, 3, ..., m)$, of a ring is connected to P_{i-1}^j and P_{i+1}^j apart from P_i^{j+1} and P_i^{j-1} and processor P_i^m is connected to P_i^1. We assume unidirectional movements for token and requests over the ring edges and bidirectional movements for token over the edges connecting the processors in adjacent rings. Each processor in this network has a switching subsystem to sensor the direction from which the request or token has been received. Also we assume that the token has a flag bit to control its movement over the processors in the adjacent columns of single side wrap around mesh.

Next we consider the popular interconnection network, regular mesh without wrap around connections in which n processors are arranged in two dimensions. A two-dimensional mesh network is obtained by arranging n processors in $m \times m$ array, where $m = \sqrt{n}$. Processors on the boundary rows and columns have fewer than four neighbors and hence fewer connections. In this network, token starts searching pending requests in one direction. When token reaches the end processor in that direction, then its direction is reversed and serves requests in other direction. Meantime it also serves all pending requests of column processors.

The analysis of the proposed algorithms uses two distinct measures namely *average number of messages per request* and *service traffic* described in section 1.3.

4.3 Allocation in interconnected rings

In this section, we describe request based token control mechanism for shared resource allocation in an interconnected rings network. In this architecture, there are two subrings with $\frac{n}{2}$ processors each, where n is the total number of processors(assumed to be even). The interconnections between two subrings are made in such a way that processor P_i (i=1,2,3,$\cdots$,$\frac{n}{2}$) in the first subring is connected to the processor $P_{i+\frac{n}{2}}$ in the second subring by a bidirectional channel in which the token as well as request messages can pass through in either directions.

Each processor has a local variable M_p that takes 1, if it has a pending request(may be of its own or of other processor's) and 0, otherwise. Each processor in the first subring has a flag variable V_p whose value is set to 1, if a request has been received from $P_{i+\frac{n}{2}}$ by P_i. The value of V_p will be checked by the token to identify the request messages received from the second subring. Token has a counter to record the receipt of request messages and token stops moving only when the values of the token counter and M_p are zero. Initially M_p, V_p and token counter are assumed to be aero and token is placed at any P_i in the first ring.

Algorithm : ICRN

(a) When P_i needs resource then

If P_i has token then it enters the critical section.

If P_i has no token then

If $1 \leq i \leq \frac{n}{2}$ then

if $M_p = 0$ then set $M_p = 1$;

if $V_p = 1$ then stop the request at P_i

else ($V_p = 0$) send the request to P_{i+1}.

else($M_p = 1$) stop the request at P_i.

else($\frac{n}{2} + 1 \leq i \leq n$) set $M_p = 1$ and send the request from P_i to $P_{i-\frac{n}{2}}$.

(b1) When P_i receives a request message then

If P_i has token then increase the token counter by one and

if $V_p = 1$ then set $V_p = 0$ and send token to the processor $P_{i+\frac{n}{2}}$

else set $M_p = 0$ and send token to the next processor P_{i+1}.

If P_i has no token then

if the request is received from $P_{i+\frac{n}{2}}$ then set $V_p = 1$

if $M_p = 1$ then stop the request at P_i

else send the request to P_{i+1}.

else (if the request is received from P_{i-1} then)

if $M_p = 1$ then stop the request at P_i

else set $M_p = 1$;

if $V_p = 1$ then stop the request at P_i

else send the request to P_{i+1}.

(c1) When P_i receives token then

If P_i has a pending request message then

if $1 \leq i \leq \frac{n}{2}$ then enter the critical section;

decrease the token counter by 1; reset $M_p = 0$;

if token is received from P_{i-1} then

if $V_p = 1$ then increase the counter by 1; set $V_p = 0$;

exit the critical section and send token to $P_{i+\frac{n}{2}}$;

else send token to the next processor P_{i+1}.

else (if token is received from $P_{i+\frac{n}{2}}$ then)

reset $V_p = 0$; if token counter is bigger than zero then,

send token to P_{i+1}; otherwise stop token at P_i.

else ($\frac{n}{2} + 1 \leq i \leq n$)

enter the critical section; decrease the token counter by one.

if token is received from P_{i-1} then send token to $P_{i-\frac{n}{2}}$

else send token to P_{i+1}.

(c2) When P_i receives token then

If P_i has no pending request message then

if $1 \leq i \leq \frac{n}{2}$ then

if token is received from P_{i-1} then

if $M_p = 1$ then set $M_p = 0$;

if $V_p = 1$ then increase the token counter by one;

set $V_p = 0$ and send token to $P_{i+\frac{n}{2}}$.

else send token to P_{i+1}.

else ($M_p = 0$)

if $V_p = 1$ then increase the token counter by one; set $V_p = 0$

and send token to $P_{i+\frac{n}{2}}$.

else if token counter is greater than zero then send token to P_{i+1};

otherwise stop the token at P_i.

else (if token is received from $P_{i+\frac{n}{2}}$ then)

if $M_p = 1$ then

reset $M_p = 0$; $V_p = 0$; if token counter is greater than zero then,

send token to P_{i+1}; otherwise stop the token at P_i.

else ($M_p = 0$) send token to the next

processor only if token counter is non-zero;

otherwise the token stops at P_i

else ($\frac{n}{2} + 1 \leq i \leq n$)

send token to the processor $P_{i-\frac{n}{2}}$.

Theorem : 4.1

The algorithm ICRN serves all request messages.

Proof:

We consider an interconnected rings network with two subrings as described in section 4.2. If there is no request message then token resides at any one of the processors in the first subring. So each originated request message from the processor in the first subring will eventually find out the token and push it till the next processor with a pending request. In fact each request generated at the processor in the first subring needs atmost $(n-1)$ movements to reach the processor with the token in the worst case. Meanwhile each request from the processor in the second subring requires just one movement from processor P_i to $P_{i-\frac{n}{2}}$ and then moves over the processors in the first subring. In the worst case, as all processors may initiate request messages for the token, the clockwise and zig-zag movements

of the token between two subrings serve all pending request messages. Thus the algorithm ICRN is aware of all requests and hence token serves all processors. ∎

Theorem : 4.2

The algorithm ICRN requires $(n-1)$ messages per request and service traffic is bounded by $(3n-2)$.

Proof:

To prove the first measure, let the token be initially at any one processor P_i in the first subring. Now a request generated at P_{i+1} has to traverse $(\frac{n}{2}-1)$ processors before it reaches the token or it has been stopped by the preceding request message. Meanwhile each processor in the second subring may originate and send requests for token. Thus in the worst case, request messages generated at the second subring require one move to reach their corresponding processors in the first subring and then moves $(\frac{n}{2}-1)$ processors in the first subring. These exchanges result in $\frac{n}{2}+(\frac{n}{2}-1)$=$(n-1)$ movements required to inform the token. Hence in the worst case, the average number of messages per request amounts to $(n-1)$.

In order to prove the second measure, service traffic, consider two types of movements (request as well as token) between two subrings. The first type includes requests from the first subring and in the absence of the requests from the second subring, atmost $(\frac{n}{2}-1)$ messages may be sent to inform the token. But by the interconnection of the second subring, each request originated from the second subring requires one move ahead per processor to reach their corresponding

interconnected processor in the first subring resulting in $\frac{n}{2}$ movements in total and then moves over the processors in the first subring provided $M_p = 0$. Thus, in total, $(n-1)$ messages are needed to inform the token in the worst case. Token starts from the processor in the first subring according to the value of V_p. The zig-zag movements of the token guarantee that the token serves all requests by visiting each processor atmost once. Thus atmost n token movements are required to visit each processor once. So atmost $(2n-1)$ messages are needed to service the processors with a pending request, regardless of the position of processors in the subrings. Additionally $(n-1)$ movements are needed, as each processor is again eligible to generate a request message after its previous request message has been served. Hence the service traffic, in the worst case, amounts to $(3n-2)$ and it is guaranteed that token skips no requests in its way to the destination. ∎

4.4 Allocation in single side wrap around mesh

In this section, we describe the allocation of single shared resource using token passing approach in a single side wrap around mesh which is depicted in Figure 4.1. In this architecture, we term the ring for which $i = 1$ (with processors P_1^1, P_1^2, $\cdots$, P_1^m) as *base ring*. Initially token resides at a processor in the base ring. It is assumed that if token is in a moving state over processors in the column of adjacent rings, no request from neighboring processors would be registered by that processor.

Each processor has local variables M_p and V_r. M_p assumes 1, if the processor has a pending request of its own or a request has been passed through it and 0,

Figure 4.1: Single side wrap around mesh network

otherwise. V_r assumes 1, if P_i receives a request from the processor P_{i+1}^{j} and 0, otherwise. Token has a flag bit L to control the token movement over processors in the adjacent columns. This flag bit L assumes 1, whenever token is received from processor P_i^{j-1} $(i > 1)$ by P_i^j. The value of L will be reset to zero as and when token reaches the processor in the base ring. When token recognizes both the values of M_p and V_r as zero at any one of the processors in the base ring, then it starts making a *check tour* to trace all remaining pending requests. For

this purpose, we assume that token additionally has a variable C for recording the number of token movements in the check tour over the processors in the base ring. During the check tour, the token recognizes V_r of any one of the processors as 1, then the check tour will be terminated; the value of C will be reset to zero and the corresponding pending request(s) from the column processor(s) will be served. Token stops moving when it recognizes the value of C as m and the values of M_p and V_r as zero.

The description of the algorithm - MICRN is as follows:

Algorithm - MICRN

// Left arrow($\leftarrow$) indicates the processor from which request or token is received.

(a) When P_i^j needs the shared resource then

$\star$ If P_i^j has token then enter the critical section.

$\star$ If P_i^j has no token then

if ($i = 1$; $M_p = 0$) then set $M_p = 1$ and send request to P_i^{j+1} if $V_r = 0$;

otherwise (either M_p= 1 or $V_r = 1$) stop the requests at P_i^j

else (if $i > 1$ then) set $M_p = 1$ and send the request to P_{i-1}^j if V_r=0;

otherwise stop the request at P_i^j

(b) When P_i^j receives a request then

$\star$ If P_i^j has token then reset $C = 0$, $M_p = 0$;

if (request $\leftarrow P_i^{j-1}$) then send token to P_i^{j+1}

else (request $\leftarrow P_{i+1}^j$) send the token to P_{i+1}^j.

⋆ If P_i^j has no token then

if $i = 1$ then

if (request $\leftarrow P_i^{j-1}$) then

if $M_p = 1$ then stop the request at P_i^j

else ($M_p = 0$) set $M_p = 1$;

if $V_r = 0$ then send the request to P_i^{j+1}; else stop the request at P_i^j

else (request $\leftarrow P_{i+1}^j$) set $V_r = 1$;

if $M_p = 1$ then stop the request at P_i^j;

else set $M_p = 1$ and send the request to P_i^{j+1}

else ($i > 1$) set $V_r = 1$;

if $M_p = 1$ then stop the request at P_i^j

else set $M_p = 1$ and send the request to P_{i-1}^j.

(c1) When P_i^j receives token and has a pending request then

enter the critical section; set $M_p = 0$;

if $i = 1$ then reset $C = 0$;

if (token $\leftarrow P_i^{j-1}$) then send token to P_i^{j+1} if V_r=0;

otherwise set $V_r = 0$ and send token to P_{i+1}^j

else (token $\leftarrow P_{i+1}^j$) reset $V_r = 0$; set $L = 0$ and send token to P_i^{j+1}

else($i > 1$)

if (token $\leftarrow P_{i-1}^j$) then

if $V_r = 1$ then reset $V_r = 0$ and send token to P_{i+1}^j

else if $L = 0$ then send token to P_i^{j+1}; else send token to P_{i-1}^j

else if (token $\leftarrow P_i^{j-1}$) then set $L = 1$;

if $V_r = 1$ then reset $V_r = 0$ and send token to P^j_{i+1}

else send token to P^j_{i-1}

else (token $\leftarrow P^j_{i+1}$) reset $V_r = 0$ and send token to P^j_{i-1}

(c2) When P^j_i receives token and has no pending request then

if (i=1; token $\leftarrow P^{j-1}_i$) then

if $V_r = 1$ then reset $M_p = 0$; $V_r = 0$; $C = 0$; and send token to P^j_{i+1}

else if $M_p = 1$ then reset M_p=0; and send token to P^{j+1}_i

else if $C =$ m then stop the token at P^j_i

else set $C = C + 1$ and send token to P^{j+1}_i

else (i=1; token $\leftarrow P^j_{i+1}$)

set $L = 0$; $V_r = 0$; $M_p = 0$ and send token to P^{j+1}_i

else ($i > 1$) set $M_p = 0$;

if (token $\leftarrow P^{j-1}_i$) then set $L = 1$;

if $V_r = 1$ then reset $V_r = 0$ and send token to P^j_{i+1};

else send token to P^j_{i-1}

else (token $\leftarrow P^j_{i-1}$) set $V_r = 0$ and send token to P^j_{i+1}

else (token $\leftarrow P^j_{i+1}$) set $V_r = 0$ and send token to P^j_{i-1}.

Theorem : 4.3

Algorithm - MICRN serves all requests generated for single shared resource.

Proof:

Initially let the token be at any one of the processors in the base ring. All processors other than those in the base ring are allowed to send requests towards the

corresponding processor in the base ring and processors in the base ring can send requests only in clockwise direction. Therefore a request generated by processor P_i^j in the ith ring is projected towards the corresponding processor P_1^j in the base ring. If the token is not available then, it will be passed to the next processor in the clockwise direction by setting the variables appropriately. In the meantime, other processors may also generate and project requests to inform token for accessing the shared resource.

The generated request will be forwarded only if it recognizes the values of M_p and V_r as zero. Otherwise, it will be stopped as another request has already been sent through that processor. Thus the generated request reaches the token and informs. Now token starts moving to serve the pending requests. Token searches for all pending requests as in steps (c1) and (c2) and stops moving after completing a check tour in the base ring. Thus in the algorithm MICRN, the token is aware of all requests and hence serves all processors. ∎

Theorem : 4.4

The algorithm MICRN requires $(n-1)$ messages per request and service traffic is bounded by $(2n + m - 1)$.

Proof:

We assume that the token is initially placed at any one of the processors in the base ring. Now each request originated at P_i^{j+1} has to traverse atmost $(m - 1)$ processors before it reaches the token or it has been stopped by a preceding request. Meanwhile processors in adjacent rings may also generate and send requests for the

token. Thus in the worst case, a request generated in the adjacent ring processor requires $(k-1)$ messages to reach their corresponding processor in the base ring and then makes $(m-1)$ moves over processors in the base ring. In the mean time, $m(k-1)$ messages may be generated by the column processors. Hence in the worst case, the total number of message exchanges per request amounts to $(m(k-1)+m-1)=(n-1)$.

To find the service traffic, we consider both the request and the token movements. The request originated from the base ring processor requires atmost $(m-1)$ messages to inform the token. Now each request originated from column processors may require atmost $(k-1)$ moves ahead per column to reach their corresponding processor in the base ring and then moves over the processors in the base ring provided $M_p = 0$. Thus in total, atmost $(n-1)$ messages are required in the worst case. To perform the check tour, Token starts from the processor in the base ring in which $V_r = M_p = 0$. This check tour requires m token movements in the base ring. So atmost $(n+m)$ token movements are required to service the processors with a pending request. In the meantime, atmost n message exchanges may be required for the newly generated requests. Hence in the worst case, service traffic amounts to $(2n+m-1)$ and token skips no requests. ∎

4.5 Allocation in regular meshes

Next we consider a regular mesh network in which bidirectional movements for both token and requests are assumed. Here token needs two flag bits T_l and T_r to record the pending requests from the left and right end processors of the

base row respectively and T_d to regulate the direction of the token over processors in the ith $(1 < i \leq k)$ row. Token maintains T_l (T_r) as 1, if request is received from the left port (right port) processor and 0, otherwise. In the base row (i=1), we require another local variable $M_p^{l(r)}$ that stores 1, if a request is received from processor P_i^{j-1} (P_i^{j+1}). Each processor in the base row (i=1) has a direction bit d that takes value 0(1) if token is sent to left (right) processor. Whenever the token moves from the base ring processor to the ith row, T_d value is set according to the current direction using d. We assume all other variables and assumptions as in the algorithm-MICRN except the token counter C, used in the check tour, which is not needed in the mesh without wrap around connections. In this algorithm, token moves from the base row processor to column processors and serves all requests in that column. Then, token is sent to the next processor in the adjacent column. From this column processor, token reaches the processor in the base row only after serving all processors in that column. Similarly the request from column processors first reaches the base row and then forwarded towards left or right port appropriately. At the beginning, the local variables $M_p^{l(r)}$, T_l, T_r and d are initialized to zero. Whenever there is no pending request, then token is idle in any one of the base row processors. The description of the algorithm is as follows:

Algorithm : SRA_Mesh

// Left arrow(←) indicates the processor from which request or token is received.

(a) When P_i^j needs resource then

If P_i^j has token then it enters the critical section.

If P_i^j has no token then

If $i = 1$ then if $M_p = 0$ then set M_p =1;

if $V_r = 0$ then send the request along the left port if d= 0;

otherwise through the right port

else (either $V_r = 1$ or $M_p = 1$) no request is sent

else(if $i > 1$ then) set $M_p = 1$ and send the request to P_{i-1}^j if $V_r = 0$;

otherwise stop the request at P_i^j.

(b1) When P_i^j receives request and has token then set $M_p = 0$;

If $j = 1[or\ m]$ then set $d = 1[or\ 0]$; reset $M_p^{l[or\ r]} = 0$;

if (request $\leftarrow P_i^{j+1}$[or P_i^{j-1}]) then

set $T_{r[or\ l]} = 1$ and send token to P_i^{j+1}[or P_i^{j-1}]

else (request $\leftarrow P_{i+1}^j$) reset $V_r = 0$ and send token to P_{i+1}^j

else $(1 < j < m)$

if (request $\leftarrow P_i^{j-1}$[or P_i^{j+1}]) then set $T_{l[or\ r]} = 1$; $M_p^{l[or\ r]} = 0$;

if $d = 0$ [or 1] then

if $T_{r[or\ l]} = 1$ then set $d = 1$[or 0] and send token to P_i^{j+1} [or P_i^{j-1}]

else send token to P_i^{j-1}[or P_i^{j+1}]

else (request $\leftarrow P_{i+1}^j$) set $V_r = 0$ and send token to P_{i+1}^j

(b2) When P_i^j receives request and has no token then

if ($i = 1$ and M_p=0) then

if (request $\leftarrow P_{i+1}^j$) then set $V_r = 1$;

else (request $\leftarrow P_i^{j-1}$[or P_i^{j+1}]) set $M_p^{l[or\ r]} = 1$; $M_p = 1$

if $j = 1$[or m] then send the request to P_i^{j+1}[or P_i^{j-1}]

else $(1 < j < m)$ send the request to P_i^{j-1} if d=0; else to P_i^{j+1}

else(i=1 and M_p=1) stop the request at P_i^j

else $(i > 1)$ set $V_r = 1$ and stop the request at P_i^j if M_p=1;

otherwise set $M_p = 1$ and send the request to P_{i-1}^j

(c1) When P_i^j receives token and has a pending request then

it enters the critical section; reset $M_p = 0$; at the end of critical section,

if $(i = 1;\ j = 1[or\ m])$ then set $T_{l[or\ r]} = 0$

if (token $\leftarrow P_2^{1[or\ m]}$) then reset $L = 0$; $V_r = 0$;

if $T_{r[or\ l]} = 1$ then set $d = 1$ [or 0] and send token to P_1^2[or P_1^{m-1}]

else if (token $\leftarrow P_i^{j+1}$[or P_i^{j-1}]) then

if $V_r = 1$ then set $V_r = 0$; $T_d = 1$[or 0] and send token to P_{i+1}^j

else if $T_{r[or\ l]}$=1 then set d=1[or 0] and send token to P_i^{j+1}[or P_i^{j-1}]

else token stays at P_i^j

else (i= 1; $1 < j < m$)

if (token $\leftarrow P_i^{j-1}$ [or P_i^{j+1}]) then

if $V_r = 1$ then set $d = 1$ [or 0]; $T_d = 1$ [or 0];

$V_r = 0$ and send token to P_{i+1}^j

else set d=1[or 0] send token to P_i^{j+1}[or P_i^{j-1}]

else(token $\leftarrow P_{i+1}^j$) reset V_r= 0; L=0;

if $d = 0$ [or 1] then if $M_p^{l[or\ r]}$=1 then reset $T_{l[or\ r]}$=1

if $T_{r[or\ l]} = 1$ then set $d = 1$[or 0] and send token to P_i^{j+1} [or P_i^{j-1}]

else if $T_{l[or\ r]} = 1$ then send token to P_i^{j-1}[or P_i^{j+1}]

else ($T_{l[or\ r]} = 0$) token stops at P_i^j

if ($i > 1$; j=1[or m]) then

if (token $\leftarrow P_{i-1}^{j}$) then

if $V_r = 1$ then reset $V_r = 0$ and send token to P_{i+1}^{j}

else if $L = 0$ then send token to P_i^{j+1} [or P_i^{j-1}]

else send token to P_{i-1}^{j}

else if (token $\leftarrow P_i^{j+1}$ [or P_i^{j-1}]) then set $L = 1$;

send token to P_{i+1}^{j} if $V_r = 1$; otherwise send token to P_{i-1}^{j}

else (token $\leftarrow P_{i+1}^{j}$) reset $V_r = 0$ and send token to P_{i-1}^{j}

else ($i > 1$; $1 < j < m$)

if (token $\leftarrow P_{i-1}^{j}$) then

if $V_r = 1$ then reset $V_r = 0$ and send token to P_{i+1}^{j}

else if $L = 0$ then send token to P_i^{j-1}[or P_i^{j+1}] if $T_d = 0$ [or 1]

else send token to P_{i-1}^{j}

else if (token $\leftarrow P_i^{j+1}$ [or P_i^{j-1}]) then set $L = 1$;

if $V_r = 1$ then set $V_r = 0$ and send token to P_{i+1}^{j}

else send token to P_{i-1}^{j}

else if (token $\leftarrow P_{i+1}^{j}$) then reset V_r= 0 and send token to P_{i-1}^{j}

(c2) When P_i^j receives token and has no pending request then

if ($i = 1$; $j = 1$[or m]) then

if (token $\leftarrow P_i^{j+1}$[or P_i^{j-1}]) then set $T_{l[or\ r]}$=0;

if $V_r = 1$ then set $M_p^{r[or\ l]} = 0$; M_p=V_r=0 and send token to P_{i+1}^{j}

else if $T_{r[or\ l]} = 1$ then set $d = 1$[or 0] and send token to P_i^{j+1}[or P_i^{j-1}]

else token stays at P_i^j

else(token $\leftarrow P_2^{1[or\ m]}$)

if $T_{r[or\ l]} = 1$ then reset $M_p^{r[or\ l]}$ =0; M_p=0; V_r=0; d=1[or 0]

and send token to P_1^2[or P_1^{m-1}]

else token stays at $P_1^{1[or\ m]}$

else ($i = 1$; $1 < j < m$)

if (token $\leftarrow P_{i+1}^j$) then

if d=0[or 1] then if $M_p^{l[or\ r]} = 1$ then set $T_{l[or\ r]} = 1$;

if $T_{r[or\ l]} = 1$ then set $d = 1$[or 0] and send token to P_i^{j+1}[or P_i^{j-1}]

else if $T_{l[or\ r]} = 1$ then send token to P_i^{j-1}[or P_i^{j+1}]

else ($T_r = 0$; $T_l = 0$) token stays at P_i^j

else if (token $\leftarrow P_i^{j-1}$[or P_i^{j+1}]) then

if $V_r = 1$ then set $V_r = 0$; $d = 1$[or 0]; $T_d = d$

and send token to the processor P_{i+1}^j

else if $T_{r[or\ l]} = 1$ then reset $M_p = 0$; $M_p^{r[or\ l]}$=0; d=1[or 0]

and send token to P_i^{j+1}[or P_i^{j-1}]

else if $T_{l[or\ r]} = 1$ then send token to P_i^{j-1}[or P_i^{j+1}]

else (T_l=T_r=0) token stays at P_i^j

If ($i > 1$; $j = 1$[or m]) then

if (token $\leftarrow P_{i-1}^j$) then if $V_r = 1$ then reset $V_r = 0$ and send token to P_{i+1}^j

else(V_r=0) if L=0 then send token to P_i^{j+1}[or P_i^{j-1}]

else send token to P_{i-1}^j

else if (token $\leftarrow P_i^{j+1}$[or P_i^{j-1}]) then set $L = 1$; if $V_r = 1$ then set V_r=0

and send token to P_{i+1}^j; otherwise send token to P_{i-1}^j

else if (token $\leftarrow P^j_{i+1}$) then reset $V_r = 0$ and send token to P^j_{i-1}

else ($i > 1$; $1 < j < m$)

if (token $\leftarrow P^{j-1}_i$[or P^{j+1}_i]) then set $L = 1$; if $V_r = 1$ then set $V_r = 0$

and send token to P^j_{i+1}; otherwise(V_r=0) send token to P^j_{i-1}

else(token $\leftarrow P^j_{i+1}$) reset $V_r = 0$ and send token to P^j_{i-1}

else(token $\leftarrow P^j_{i-1}$)

if V_r=1 then set V_r=0; M_p=0 and send token to P^j_{i+1}

else if L=0 then send token to P^{j+1}_i[or P^{j-1}_i] if T_d=1[or 0];

else(L=1) send token to P^j_{i-1}

Theorem : 4.5

The algorithm **SRA_Mesh** serves all requests generated for the single shared resource and requires $(n-1)$ messages per request and service traffic is bounded by $(2n+1)$.

Proof:

In algorithm **SRA_Mesh**, token serves all requests in one direction and then its direction is reversed. Token serves requests according to the value of flag bits T_l and T_r. Token, from an intermediate processor, takes diversion to the next column as per the value of T_d. Token, while reaching the base row processor, it checks $M_p^{l[or\ r]}$ value to store the pending request received from the left[or right] processor. All other arguments are similar to the proof of Theorems 4.3 and 4.4. ∎

In last two algorithms, as processors are aware of incoming and outgoing requests and token movements, we can detect the faulty nodes or links that occur

other than the base row (which is assumed to be fault-free) before starting the execution of the algorithms.

4.6 Conclusion

In this chapter, we have considered the problem of controlling the allocation of single shared resource among processors in two interconnected rings network, single side wrap around mesh and regular 2-dimensional mesh using request based token passing strategy. First we have proposed an algorithm ICRN for controlling the allocation of single shared resource in the network, in which processors in two rings are interconnected appropriately. In this algorithm, we additionally require a counter for performing the check tour in which the token is aware of the pending requests generated later on. Then we have proposed a request based token control algorithm MICRN that allocates the single shared resource in a single side wrap around mesh. Finally we have proposed a shared resource allocation algorithm SRA_Mesh for regular mesh with atmost $(n-1)$ messages per request and $(2n+1)$ service traffic. In this algorithm, we do not require an additional check tour, as the processors in the base row is aware of the direction through which the token has been sent.

Chapter 5

Allocation in general networks

5.1 Introduction

A variety of token based algorithms use the idea of requesting the token from only one processor, for example, to the processor that last requested the critical section (Naimi and Trehel 1987) or to one designated processor that would ultimately receive a circulating token and append requests to this circulating token (Kumar *et* al. 1991). Then Helary *et* al. proposed an algorithm that indicates the importance of considering the network topology on a distributed mutual exclusion algorithm and it does not take the advantage of dynamic local information (Helary *et* al. 1988). This algorithm just blindly searches the whole spanning tree rooted at a requesting processor. Still this distributed mutual exclusion algorithm has the message complexity of $O(n^2)$ per critical section invocation.

Raymond(1989) proposed a token based mutual exclusion algorithm that requires $\log(n)$ messages per critical section invocation, but the algorithm works for tree connected networks (Raymond 1989-a). The $O(\log n)$ complexity is achieved only for star-shaped tree networks. In the worst case, for a chain-shaped tree net-

work, the number of messages would be $O(n)$. Thambu and Wong (Thambu and Wong 1995) proposed an efficient token based mutual exclusion algorithm in the distributed system using the idea of projective planes. This algorithm assumes an error-free fully connected network and uses a time-out mechanism to avoid infinite delay. Also it requires two arbitrator sets called $superiors_i$ and $inferiors_i$. If a requesting processor j does not know any of its inferiors holding the token, then requesting all of j's superiors will definitely lead to the holder of the token. Therefore there will always be a path from a requesting processor to the processor that holds the token. This algorithm requires sending token request to a set of $\sqrt{n}$ processors at most.

Yan *et* al. presented a simple and efficient mutual exclusion algorithm in arbitrary network topologies (Yan *et* al. 1996). This algorithm shortens the request delay by fully taking advantage of the network dynamic status information. In this algorithm, each processor sends out only one request message to chase the token. While chasing the token, a request message dynamically adjusts its chasing path based on the local information at intermediate processors. Then Gregory *et* al. studied the problem of efficiently scheduling by evolving binary-tree-structured computations on a ring-structured parallel computer (Gregory *et* al. 1996). Next Chang presented a hybrid approach in which processors are divided into groups (Chang 1996). Barrows *et* al. studied the empirical aspects of dynamic scheduling on rings of processors (Barrows *et* al. 1999). The processors use one local algorithm to solve conflicts with processors in the same group and use a different global algorithm to solve conflicts with processors in other groups.

Datta *et* al. presented a deterministic distributed depth-first token passing protocol on a general network with a distinguished root (Datta *et* al. 2000). This protocol uses neither the processor identities nor the size of the network, but assumes the existence of a distinguished processor, called the root of the network. This protocol is self-stabilizing in which it is guaranteed to reach a state with no more than one token in the network. The depth-first token circulation scheme has many applications in the distributed systems. The solution to this problem can be used to solve the mutual exclusion, spanning tree construction, synchronization and many other important tasks.

Petit and Villain proposed depth-first search algorithms for tree structured networks using the static model in which the existence of a spanning tree is not assumed (Petit and Villain 1999). Housni and Trehel presented a token based management protocol for shared resources (Housni and Trehel 2001). They used the example of a teleconference with a point-to-point communication between speakers. Here the power of speech is the resource and the member who is speaking is the token holder. Every member must send a request to have right to speak. This request moves up from one member to another one going toward the token holder. At a given time, a logical tree expresses different path requests and must follow the members to reach the token holder and reciprocally the token going toward each requester. Thus a single request queue, including all requests, moves from one requester to another with the token, following the logical tree.

Chen *et* al. have proposed a self-stabilizing algorithm for constructing a spanning tree in a distributed network (Chen *et* al. 1991). Huang and Chen have

proposed a self stabilizing protocol that circulated a token on a connected network in nondeterministic depth first search order, rooted at a special processor (Huang and Chen 1993). Then Wu and Shu proposed an efficient distributed centralized token-based mutual exclusion algorithm with central coordinator in which the token is informed of the identity of the processor which is in need of the shared resource (Wu and Shu 2002).

In this chapter, we propose two algorithms based on request message based token control strategy for shared resource allocation in a general network. In the next section, we present the model of communication network. Section 5.3 presents a two phase algorithm for the token distribution problem in general networks. In section 5.4, we have proposed an algorithm for the single shared resource allocation using token oriented acyclic network in the form of a linear array. Section 5.5 concludes this chapter with future remarks.

5.2 The model of communication network

The underlying topology of the distributed system is abstracted as an undirected fault-free network $G = (V, E)$ with $|V| = n$ asynchronous processing elements. The links are assumed to be bidirectional. We denote $deg(P_i)$ as the number links incident on a processor(P_i) and $m_i = deg(P_i)$ and assume that each processor knows this information. The characteristics of each processor is assumed as described in section 1.3. If any one of the processors P_k ($P_k \in \{adj(P_i)\}$) among m adjacent processors of P_i, has no other adjacent processor(i.e., $deg(P_k) = 1$), then the processor P_k is termed as a *leaf* processor. A processor P_i is termed as a

central processor only if it has $m_i = deg(P_i) > 1$ adjacent processors and however, these central processors impose no centralized management. Still these processors follow completely distributed management. Each processor P_i knows the existence of links and assigns a unique local index to each link incident on it. Hence the movements of either request or token is controlled only by these local links and not by the actual indices of the adjacent processors. We assume that the message passing in any link is reliable and the communication delay in a link is finite and unpredictable, determined by network contention (Yan *et* al. 1996).

The analysis of the proposed algorithms also uses two distinct measures namely *average number of messages per request* and *service traffic* described in section 1.3.

5.3 Token-based algorithm in general networks

Consider a general network $G = (V, E)$ with n processors as depicted in the Figure 5.1(a). From G, we can construct a spanning tree G' with $|V| = n$ vertices and $|E'| = n - 1$ links, as in the Figure 5.1(b) and 5.1(c), using any well known self-stabilizing algorithm (Lakshmanan *et* al. 1987, Chen *et* al. 1991, Huang and Chen 1992, Huang and Chen 1993). We assign indices to the processors in the order of their traversals. Here, we retain the parent link for each processor, say $l = parent(P_k)$. In G', as and when token passes over the processors, each processor P_i maintains the identity of the link through which the token has been routed. The request generated in future, by processor P_i is directed towards that link through which the token has been sent. The current state of such a spanning tree is referred to as *token oriented acyclic network*. The processor P_i

$(i = 1)$ is indexed as the root processor and rest of the processors are indexed as $\{P_2, P_3, \cdots, P_n\}$ in the token oriented acyclic network G'.

Lakshmanan *et* al. have proposed a time optimal algorithm to construct a depth-first-search tree for an asynchronous communication network. This construction requires less than $4m - (n - 1)$ messages, where m and n denote the number of links and processors respectively (Lakshmanan *et* al. 1987).

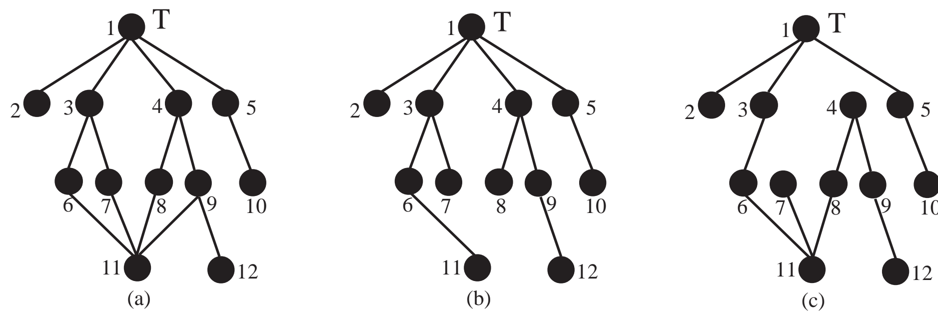

Figure 5.1: (a) A general network with n=12 processors; (b) a spanning tree; (c) an indexed tree

5.3.1 Allocation algorithm using a queue: GNet

The proposed algorithm GNet consists of two phases. The first phase describes the allocation of the shared resource in a leaf processor. The second phase keeps track of the token as well as request movements at central processors. Here each leaf processor P_i has a local variable M_p to control the movement of its own request and assumes the value 1, if P_i has its own pending request and 0, otherwise. Each central processor P_i maintains a *queue* q_i to record the status of the receipt of requests received from its links and also of its own request.

Next we assume that token moves with a request indicator T_r which moves along the token to indicate the existence of additional pending requests other than the $firstEl()$ - *the first entry*, in the queue q_i. T_r assumes 1, if there is a pending request other than the first entry of q_i and 0, otherwise. Each central processor has a direction indicator d - a variable which identifies the link through which token had been sent. d assumes l, if the token is passed through that link l that connects the adjacent processor P_j with P_i and 0, if token has not yet passed through the processor P_i. At the beginning, M_p, T_r and d are initialized to zero; q_i is assumed to be empty and initially token is placed at P_1.

PHASE I

This section deals with the allocation of single shared resource among leaf processors in the token oriented acyclic network. When a request is generated at a leaf processor P_i, it accesses the single shared resource only if it has the token; otherwise the generated request is passed to its neighbor. Whenever P_i receives a request then it must posses the token, because P_i is a leaf processor. Similarly whenever P_i receives token, it must have its own pending request. However, if the request indicator T_r=1 then, the processor P_i resets T_r=0 and sends token to P_j after serving P_i; otherwise the token stops at P_i itself.

The description of PHASE - I is given in Figure 5.2.

Variables kept at each leaf processor P_i:

boolean M_p, T_r

On demand of the shared resource at P_i:

if P_i has token then it enters critical section;

else sets $M_p = 1$ and sends request through the link j

On receipt of a request at P_i from the link j: // P_i must possess token

processor P_i sends token through the link j

On receipt of the token at P_i from the link j: // P_i must have a pending request

P_i enters critical section; sets $M_p = 0$ and at the end of critical section,

if T_r=1 then set T_r=0 and send token through the link j;

else stop token at P_i itself.

Figure 5.2: Token based control algorithm for shared resource allocation among leaf processors

PHASE II

In this section, we describe the algorithm for controlling the movements of requests and token at central processors. Whenever a central processor P_i needs a resource and has no token then it adds the request into its queue and then forwards the request to its neighbor, if it has not sent a request earlier. The request messages generated for the shared resource keep track of the value of direction indicator d to identify the link leading to the processor through which the token is available. After getting informed, token starts serving the pending requests. On reaching a processor P_i from P_j, token scans the entries in the queue q_i using *firstEl()* - first entry of the queue, that identifies the first entry in the queue without removing it. If the index in *firstEl(q_i)* is i then, the processor P_i enters critical section.

Otherwise the processor P_i sets $d = firstEl(q_i)$. Then if T_r=1 then the queue is enqueued with the local index of the link from which token is received and dequeued. Before sending the token through the link d to the adjacent processor, P_i checks the status of the queue q_i; if q_i is still nonempty, then the request indicator T_r is set appropriately and then the token is passed through the selected outgoing link. Now token starts moving from P_i through the adjacent link as indicated by d. Finally token stops at a central processor P_i only if T_r=0 and queue q_i is empty. The value of d plays a vital role in directing the request received at P_i in the absence of the token.

It is clear that there could be atmost one request received from any adjacent link in the queue q_i. All requests received from adjacent processors are entered into the queue q_i. A request received from an adjacent processor will be forwarded to the next processor based on the value of d if the queue was already empty.

The detailed description of PHASE II is given in Figure 5.3.

Variables kept at each central coordinator P_i:

int i, j, k, l, d, q_i;

boolean T_r

queue q_i

On demand of the single shared resource at P_i:

if P_i has token then it enters critical section;

else (if P_i has no token then)

if $queue(q_i) = empty$ then *enqueue* the request status of P_i in q_i and

if d=0 then send request through the link l to the parent of P_i

else $(d = j)$ send request through the link j

else($queue(q_i) \neq empty$)

enqueue the request status of P_i in q_i and stop the request at P_i

On receipt of a request at P_i *from the link* j:

if P_i has token then set $d = j$ and send token through the link j

else(P_i has no token)

if $q_i = empty$ then *enqueue* the request status of the link j in q_i

if $d = 0$ then send the request through the link l - the parent of P_i

else($d = k$) send the request through the link k

else($q_i \neq empty$)

enqueue the request received from the link j in q_i and

stop the request at P_i.

On receipt of token at P_i *from the link* j:

if *firstEl*(q_i) $= i$ then enter critical section;

dequeue(q_i) and at the end of critical section, do:

if $q_i = empty$ then

if $T_r = 1$ then set $d = j$; reset $T_r = 0$ and send token through the link j

else stop token at P_i

else ($q_i \neq empty$) set $d = firstEl(q_i)$; $dequeue(q_i)$;

if $T_r = 1$ then $enqueue(q_i)$ with the local index of the link

through which token is received; set $T_r = 1$

and send token through the link d

else($T_r = 0$)

if $q_i \neq empty$ then set $T_r = 1$ and send token through the link d.

else ($q_i = empty$) send token through the link d

else($firstEl(q_i) \neq i$) set $d = firstEl(q_i)$; $dequeue(q_i)$;

if $T_r = 1$ then $enqueue(q_i)$ with the local index of the link j

and send token through the link d

else if $q_i \neq empty$ then set $T_r = 1$ and send token through the link d

else ($q_i = empty$) send token through the link d.

Figure 5.3: Token based control algorithm for shared resource allocation among central processors

Theorem: 5.1

The algorithm GNet serves all request messages.

Proof:

Consider a fault-free connected network $G = (V, E)$ with $|V|{=}n$ processors. The given network G is first converted into the token oriented acyclic network which in turn may be either a spanning tree or an ordered tree G' where n processors and $(n - 1)$ links. Placing the token initially at P_1 implies P_1 as the privileged processor and all non-privileged processors can generate requests. The generated request searches the token and the request generated first by P_i, $1 < i \leq n$, will be forwarded to the processor P_1 through the link to P_k (adjacent processor to P_i) only if P_k, $1 < k < n$ and $k \neq i$, has not already generated or forwarded a request towards P_1. When token received a request, it moves through the link from

which the request is received. Whenever token crosses a processor P_i, the processor P_i maintains d which assumes the appropriate index assigned locally to the link through which the token is forwarded from that processor P_i. Thus the processor P_i has no knowledge about the index of adjacent processor P_j. The queue q_i at each processor P_i maintains the links from which requests have been received. As and when token moves from one processor to another, it carries the boolean value T_r to indicate the existence of pending request(s). According to the entries of local queue q_i in each processor P_i, token moves to serve all processors until $T_r = 0$ and q_i is non-empty. Here in each processor, there is a local queue which allows requests generated to be served in order. This guarantees no starvation for the token and hence token serves all requests. ∎

Theorem: 5.2

The algorithm GNet for single shared resource allocation in general networks requires $(2n-1)$ messages per request

Proof:

To prove the bound on the number of messages per request, we use amortized complexity analysis (Feuerstein *et* al. 1998, Tarjan 1985). First we assume that R is the set of pending requests spread over processors in the network; M is the set of waiting request messages for the token and C is the current value of the request indicator T_r. Clearly each processor P_i can have atmost $m = deg(P_i)$ waiting requests in the queue q_i. Thus, in total, there could be atmost $(2n-1)$ enqueued waiting requests in the worst case. It is also to be noticed that each request generated at any processor P_i will be enqueued at more than one intermediate

processor only if that intermediate processor has forwarded no other request. The request generated / received first will be forwarded through the link that leads to the processor where token is available.

Now each pending request message $r \in R$, let $d_1(r)$ denotes the distance from the processor with the token to the processor with a pending request r. Similarly for each waiting request $m \in M$, $d_2(m)$ denotes the distance from the processor currently possessing a waiting request m to the processor currently having the token T.

Let Φ be a potential function,

$$\Phi = \sum_{r \in R} d_1(r) + \sum_{m \in M} d_2(m)$$

The formerly generated/received request, from each branch of the token oriented acyclic network, only reaches and informs the token and all other requests generated/received are enqueued in their respective queues and stop as already a request had been passed through it. The amortized cost of a message is defined as $1 + \Delta\Phi$, where 1 is the cost for exchanging the message or token due to the algorithm and $\Delta\Phi$ is the variation of potential function due to exchanges.

We study the amortized cost of any exchanged message:

- The token is passed through a link from P_i:

  In this case, as per the description of the algorithm, q_i is non-empty, i.e., there exist atleast one entry in the queue and hence $|q_i| > 0$ which implies $C = 1$. In the worst case, there could be $(2n - 1)$ enqueued waiting requests. Therefore

the variation in the potential function, when token carries no waiting request from previous processor, is decreased by 2 by reducing 1 each from $d_1(r)$ and $d_2(m)$. Also if the token carries a pending request from earlier processor then the variation in the potential function is decreased by one.

- A request is passed through a link from P_i:

  The variation in the potential function Φ increases by 2 since the generated request message has increased $d_1(r)$ and $d_2(m)$ each by 1 in the worst case.

Each new request increments Φ by atmost $(2n-1)$ in the worst case (queue at all P_i are non empty and $T_r > 0$). For a sequence of k message / token exchanges, the amortized cost is $k*(2n-1)$ in the worst case and $(2*k-1)$ in the best case. In the best case, the variation in Φ would be 2. Therefore the average number of message exchanges is $(2n-1)$, in the worst case and 2, in the best case. ∎

Theorem: 5.3

The algorithm GNet for single shared resource allocation in general networks requires $3(n-1)$ service traffic in the worst case.

Proof:

We consider two types of request movements to prove the service traffic. In the first case, we consider the network in which each central processor except the root is assumed to have exactly two neighbors(linear array). In this case the generated request at a leaf processor reaches the processor P_i, where token is available, in (n-1) movements (the worst case). In the meantime the intermediate processors may generate their own requests and these requests will be either forwarded towards P_i

or stopped as already a request had been sent. Thus the generated request informs the token within atmost (n-1) movements. Now token moves to serve the pending requests. Token needs atmost (n-1) movements to reach the end processor, which is possibly having a pending request. Therefore the number of message exchanges will be (2n-2). In the second case, we consider a network in which each central processor may have more than 2 adjacent neighbors. The requests, received from all neighbors of a central processor, will be enqueued in the queue q_i and the first received request will be forwarded only if no request has yet passed through that central processor. The rest of the received requests will be stopped and their entries are added into the queue. Thus the request generated for the shared resource reaches the token within atmost (n-1) movements.

Now token traces the queue entries to check pending request(s) of its own or of its neighbors to select an outgoing edge leading to its neighbor P_i based on the first entry j in the queue. When it reaches the neighbor P_i through the selected outgoing edge from P_i then P_i's pending request carried by T_r is updated in the queue of P_i. Note that the remaining pending requests of P_i get chance only after serving all the pending requests of neighbors of P_i or of its own and also observe that after getting served once, P_i or its neighbors get access of the token only after servicing the remaining pending requests of P_i. This type of token traversal amounts to atmost (n-1) token exchanges. Subsequently the processors served by the token, may also generate requests again. But this type of requests will be updated in the queues of the intermediate processors. There is atleast one processor which may, in each round, generate a request for the shared resource. Thus in total, there would

be (n-1) processors with a pending request. Hence summing up of all these three type of movements implies a total of (n-1)+(n-1)+(n-1)=3(n-1) service traffic in the worst case. ∎

5.3.2 Differences between Raymond's algorithm(1989) and the proposed algorithm

- We have proposed an algorithm for single shared resource allocation in token oriented acyclic network. In the proposed algorithm, it is assumed that neither token nor request message carries information about the identities of source and destination processors.

- In Raymond's algorithm (1989), each processor knows the existence of its neighbors in the tree. But in our algorithm, each processor knows the existence of links connected to it only by their local indices. Hence we have relaxed the assumption, in Raymond(1989), that each processor knows the identities of the adjacent processors and monitors the movement of the token by keeping track of the variable $Holder_x = Y$. This $Holder$ variable in each processor knows the identity of adjacent processors and each adjacent processor knows their adjacent processors by their $Holder$ variables (Raymond 1989-a). In the proposed algorithm as each processor assigns a local index to each link connected to it and hence it has no knowledge about the identities of even its adjacent processors. Also as the token knows the existence of pending requests by means of the boolean flag T_r and q_i, the direction indicator d directs the token to select the next outgoing link. Thus

the variable $Holder_x = y$ in each processor is not required in our algorithm.

- In the proposed algorithm, even in the absence of *Holder* variables, it is possible to collectively maintain token oriented paths from each processor to the privileged processor.

- In Raymond's algorithm, the queue of each processor holds the index of adjacent processors from which requests had been received. Here we do maintain a queue of requests not by the indices of adjacent processors but by the local indices of links connected to it.

- The most important variation in the proposed algorithm is the updated value of the request indicator T_r that identifies the existence of additional pending requests other than the first entry of the queue q_i.

5.4 Allocation by embedding token oriented acyclic network into a linear array

In this section, we propose an allocation algorithm for the single shared resource using token oriented acyclic network in the form of a linear array. In chapter 2.2, we have presented an algorithm for the bidirectional linear array of n processors and $(n-1)$ links with $2(n-1)$ messages per request and service traffic of $3(n-1)$.

The token oriented acyclic network shall be embedded into a linear array in such a way that the indices of the processors in the acyclic network shall be mapped to the indices of the processors in the linear array. Now the links joining successive processors in the acyclic network shall be mapped to a single link joining the

corresponding successive processors in the linear array. This mapping shall be used for bidirectional communications among processors that are indexed from 1 to n, say P_1 to P_n. The number of links travelled during a visit from processor P_1 to processor P_n will be, in the worst case, $(2n-1)$ that includes the counting of backtracked links also. Note that the path from processor P_i to P_{i+1} must be stored which will be useful for token and request traversals.

As and when a request is initiated, it searches and informs the token and then token starts serving requests left to right in the linear array. In one traversal from left to right, all processors having pending requests will be served only once, that is, at the first visit to the processor but not during the backtracking towards the visit to P_n. Similarly when token moves from right to left, all pending requests during the first visit will be satisfied. The token does not change its direction as long as there are pending requests in the current direction. Therefore the algorithm prevents starvation of any request or token. This kind of token and request movements require two traversals: *one* for the request and *another* for the token. Hence the number of message exchanges per request is $2(2n-1)$.

For the second measure, the algorithm requires additional $(2n-1)$ message exchanges due to additional requests generated by the other processors. Therefore the service traffic of the algorithm is $3(2n-1)$. Thus we have,

Theorem: 5.4

The above algorithm satisfies all pending requests with in a finite time, requires $2(2n-1)$ messages per request and the service traffic is bounded by $3(2n-1)$. ∎

5.5 Conclusion

We have proposed algorithms based on token passing strategy for the single shared resource allocation problem in general networks. We have first converted the given undirected network into a token oriented acyclic network that may be derived from either a minimal spanning tree or an ordered tree network. The first algorithm consists of two phases for shared resource allocation in token oriented acyclic network. The first PHASE of the algorithm deals with shared resource allocation in leaf processors. The second PHASE of the algorithm controls the movements of request and token in non-leaf processors. Also we have proposed another algorithm based on the embedding of the token oriented acyclic network into a linear array of processors in which token and request messages are permitted to move on both directions. The proposed algorithms may also be useful for ad hoc networks if the token oriented acyclic network can be established based on the self stabilizing approach of Huang and Chen (Huang 1993, Huang and Chen 1993).

Chapter 6

Distributed algorithms for sorting and prefix computation

6.1 Introduction

In the design and analysis of algorithms, sorting problem is one of the most fundamental problems. We consider the sorting problem with a set of elements distributed over processors in a line network and in static ad hoc mobile networks. The traditional lower bound for distributed sorting problem has been considered to be n rounds because the number of disjoint comparison-exchange operations required n rounds for parallel sorting on a linear array (Akl 1989, Leighton 1992). Then it has been reduced to $(n-1)$ rounds by Sasaki's algorithm (Sasaki 2002). This problem has been extensively investigated in distributed contexts. The potential efficiency of a distributed system is inherent in the design of an effective algorithm that minimizes the number of message exchanges as well as the computation time (Luk and Ling 1989).

There are several algorithms for distributed sorting in various topologies. Loui (Loui 1984) presented a simple sorting algorithm on rings. Rotem *et* al. (Rotem

et al. 1985) have studied the static and dynamic versions of the sorting problem where each node contains a subset of elements. Zaks (Zaks 1985) presented a sorting algorithm for a tree network and then extended it to general networks. Then Marberg and Gafni (Marberg and Gafni 1987) developed a sorting method for a multi-channel broadcast network. Then McMillin and Ni (McMillin and Ni 1992) described the distributed sorting problem with an unreliable network. We deal with a static sorting problem with a reliable network which has been used by many researchers (Akl 1989, Gerstel and Zaks 1997, Horowitz *et* al. 2001, Leu *et* al. 2000, Loui 1984, Marberg and Gafni 1987, Rotem *et* al. 1985, Zaks 1985).

An important observation in all these results is to find a strategy that minimizes the amount of communication. For example, Gerstel and Zaks (Gerstel and Zaks 1997) showed that for every network with a tree topology T, every sorting algorithm must send at least $\Omega(\Delta_T \log(L/N))$ bits in the worst case, where $\{1, 2, \cdots, L\}$ is the set of possible initial values and Δ_T is the sum of distances from a median of T to all processors in it. Then Pan *et* al. (Pan *et* al. 1997/98) presented a parallel quick sort computational model on a linear array with a reconfigurable pipelined bus system. Hofstee *et* al. (Hofstee *et* al. 1990) designed a time-optimal sorting algorithm on a line network with restricted local memory. However in this algorithm, each processor has to have at least two elements and the algorithm fails when each processor has exactly one element. Recently Sasaki proposed a time-optimal distributed sorting algorithm with a strict lower bound of $(n-1)$ rounds on a line network by creating the copies of elements at intermediate processors (Sasaki 2002). Thus the number of elements used for sorting is $2(n-1)$. Even though

Sasaki's algorithm is based on the odd-even transportation sort, it does not ensure that each processor always has two elements of the same value at the final round. We propose an algorithm for distributed sorting using median based exchanges in a line network without creating the copies of elements at all processors.

Then we present algorithms for distributed sorting and prefix computation problems on static ad hoc mobile networks. A *mobile ad hoc network* is a network where in a pair of processors communicates by exchanging messages either over a direct wireless link or over a sequence of wireless links including one or more intermediate processors. Direct communication between a pair of processors is possible only if both processors fall in the same transmission radius. Wireless *link failures* occur when communicating processors exceeds the transmission range of each other. Similarly wireless *link formation* occurs when communicating processors, that were separated far apart from each other, comes within the transmission radius of each other. Thus the main feature that distinguish ad hoc mobile networks from existing distributed networks include frequent and unpredictable topology changes and highly variable message delays (Gafni and Bertsekas 1983, Walter *et* al. 2001). Here we consider a static ad hoc mobile network as the token oriented acyclic network. We present distributed sorting and prefix computation algorithms for a static ad hoc mobile network induced into the token oriented acyclic network that dynamically changes the logical structure to adopt the changing physical topology in the mobile ad hoc environment.

In next section 6.2, we have described the distributed sorting and prefix computation problems. Section 6.3 carries the proposed distributed sorting algorithm

for a line network. In section 6.4, simulation results of the proposed algorithm are presented in comparison with the recent work in distributed sorting on the line network. For a static ad hoc mobile network, section 6.5 describes the distributed sorting algorithm and section 6.6 describes an algorithm for prefix computation. Finally, concluding remarks in section 6.7 completes this chapter.

6.2 The problem and the computational model

The definition of the distributed sorting problem, considered in this chapter, is as follows: *At the initial state, each processor P_i has its element u_i for sorting. Then, the position of each element is rearranged to satisfy the condition, $\forall$ i, $0 \leq i < n-1$, $u_i \leq u_{i+1}$ at the final state.* The distributed prefix computation problem on static ad hoc mobile networks is as follows: *Given n values $\{u_1, u_2, \cdots, u_n\}$ distributed over n processors in an ad hoc mobile network and an associative binary operation $\oplus$, we compute i^{th} prefix sum of the processor P_i as $u_1 \oplus u_2 \oplus \cdots \oplus u_i$, for $1 \leq i \leq n$.*

Next we describe the assumptions in the *line network* and *ad hoc mobile network* - the underlying computational models used for distributed sorting and prefix computation. A line network is defined as a linear collection of n processors P_0, P_1, P_2, $\cdots$, P_{n-1} where each P_i, $0 < i < n-1$, is bidirectionally connected to P_{i-1} and P_{i+1}. Each processor can communicate with its direct neighbor(s) only. We assume that P_0 is the end processor on the left of the network and P_{n-1} is the end processor on the right of the network. Also we assume that each processor knows its neighbors only by local names of *left* and *right*, with the orientation consistent

along the line. Each processor P_i is equipped with a restricted local memory and it is capable of having a constant number of elements.

The static ad hoc mobile network is considered as a connected network $G = (V, E)$ where V denotes the set of processors and E denotes the set of wireless links. In ad hoc mobile network, each processor is capable of communicating with other processors via direct wireless links or a sequence of wireless links. If two processors fall within the transmission radius of each other, then direct communication between them is possible; otherwise the communication takes place via a sequence of wireless links. Then we can construct a token oriented acyclic network $G' = (V, E')$ where $|V| = n$ and $E' = |V| - 1$, n is the number of processors. Thus the network G' may either be a spanning tree or an ordered tree with n processors and $(n-1)$ edges. The analysis of the proposed algorithms uses two distinct measures namely *time complexity* which is measured in terms of the number of rounds and *communication complexity* which is measured in terms of the total number of message exchanges.

6.3 A time-optimal algorithm for sorting in line network

The distributed sorting problem is similar to a parallel sorting problem on a linear array, which can be solved by using the *odd-even transposition sort* on a synchronous model. At first, we brief the operations of the *odd-even transposition sort* (Akl 1989, Leighton 1992, Sasaki 2002) as follows: *At an odd-numbered step, a processor P_i whose suffix i is odd exchanges its element u_i with P_{i+1}'s element*

u_{i+1} if u_i is larger than u_{i+1}. At an even numbered step, P_i whose suffix is odd exchanges its element u_i with P_{i-1}'s element u_{i-1} if u_{i-1} is larger than u_i. Each processor executes the above operation n steps.

To apply the odd-even transposition sort, P_i has to know its global position, because the difference between the execution of odd-numbered step and even-numbered step depends on i. While adopting this type of communication, each processor always has two copies of the same element at the final state. Consequently, the time complexity is n rounds, since sorting is executed with $2n$ elements. Without additional $(n-1)$ overhead rounds to learn the global position, Sasaki (Sasaki 2002) adopted a strategy that does not require information on the global position, i.e., a strategy in which each processor communicates with both neighbors simultaneously as in Hofstee *et* al.'s algorithm (Hofstee *et* al. 1990). The improvement in the algorithm has been achieved by cancelling the creation of copies in the leftmost and rightmost processors, i.e., each P_0 and P_{n-1} has only one element in each round. This implied that the number of elements used for sorting is $2(n-1)$ and the time complexity is $(n-1)$ rounds. However, this improved algorithm does not ensure that each processor always has two copies of the same element at the final state (Leighton 1992, Sasaki 2002).

We have proposed an $(n-1)$ rounds algorithm for the distributed sorting problem on a line network. Sasaki's algorithm can be improved by implementing the median based exchanges and cancelling the creation of copies at intermediate processors, i.e., each intermediate processor P_i has always only one element in every round. This explicitly implies that the number of elements used for sorting

is exactly n. Accordingly, the time-complexity of $(n-1)$ rounds is derived from the exchanges of neighboring elements at the median processor and this exchange is entirely different from the conventional odd-even transposition sort.

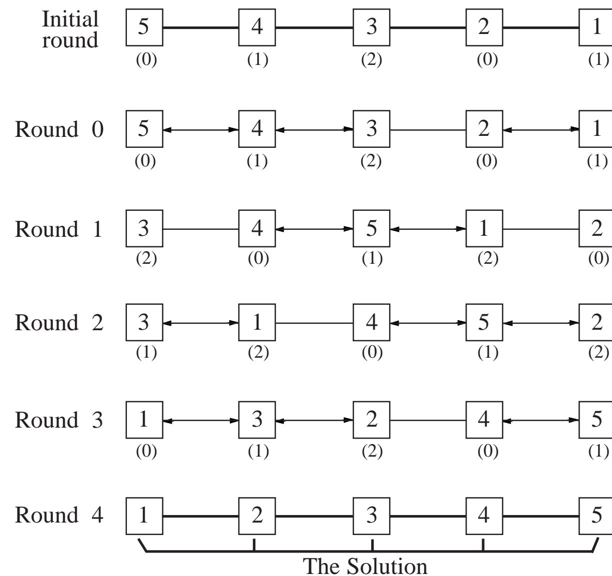

Figure 6.1: An example of the $(n-1)$ round algorithm with n=5 elements

First we explain the proposed algorithm with an example as shown in Figure 6.1. The initial state resembles with the one that is as in n round algorithm as well as in Sasaki's algorithm. But in the final state, we do not require a separate rule, as in (Sasaki 2002), for processor P_i to select an element as the solution. In fact, the element stored by the processor P_i after $(n-1)$ rounds, is itself the correctly sorted element even in the worst case.

Sasaki's sorting algorithm for a line network does not require information to learn the global position. But as in Hofstee *et* al.'s algorithm(Hofstee *et* al. 1990),

in which a processor communicates with both neighbors simultaneously, we do communicate with both neighbors simultaneously only at the median processor and not at other non-median processor. The non-median processors can communicate only with its corresponding neighboring median processor. There are efficient distributed algorithms for determining the medians (Frederickson 1983, Gallager *et* al. 1983). In the sequel, we describe the process of how to select an adjacent median processor in each round. The processor whose mark is 1 is the median processor and whose mark is either 0 or 2 are non-median processors. First we fix a mark T_i to each processor P_i, $0 \leq i \leq n-1$, in such way that $T_i \equiv i \pmod 3$ in the initial round. Here a processor neither creates nor holds a copy of the element in each round. Then for the subsequent rounds, the mark follows *two step forward* incremental operation to exchange the smallest element to the left processor and the largest element to the right processor as well. So after the initial round, the mark T_i of each processor P_i is set as $T_i \equiv (T_i + 2) \pmod 3$ to achieve two-step forward incremental operation. Thus the state for becoming as a median processor is determined by the value of the mark T_i. Also this mark helps a processor P_i to select the direction in which the element has to be exchanged with its median neighbor.

It is quite interesting to notice that among intermediate processors, only the median processors involve in performing two *receipt* and two *send* operations where as all other non-median intermediate processors as well as the end processors (may be median or non-median) need to perform only one *send* and one *receipt* operation simultaneously. But as in (Sasaki 2002), each intermediate processor needs com-

pulsorily two *receipt* and two *send* operations and the end processors need only one *receipt* and one *send* operation. The proposed distributed sorting algorithm greatly reduces the computational time. Thus, in each round, each non-median processor P_i, whose mark is 0, communicates its element for comparison with the median processor P_{i+1} and obtains the smallest element contributed by that median processor. Similarly, each non-median processor P_i, whose mark is 2, communicates its element with the median processor P_{i-1} and obtains the largest element contributed by that median processor. Now it is clear that each median processor P_i receives elements from its adjacent neighbors; exchanges the received elements along with the element of its own to preserve the partial order $<$ of three elements and then sends back the smallest one to the processor whose mark is 0 and the largest element to the processor whose mark is 2. The same process is repeated for $(n-1)$ rounds. After the execution of $(n-1)$ rounds, the resulting sequence is itself in the sorted order and as in Sasaki's algorithm (Sasaki 2002), we do not require a special rule for selecting the solution. The process of ensuring each processor to have two elements of the same value at the final round is not required. This new median based algorithm is different from traditional odd-even transposition sort, for distributed sorting problem in which each processor always has only one element at the end of each round. Instead of using two-step forward incremental operation, if we use $T_i \equiv (T_i + 1)(\text{mod } 3)$, then the worst case number of rounds exceeds $(n - 1)$ rounds in some cases. Next we describe the practical operations necessary for the simulation of the proposed distributed sorting algorithm on a line network. Each processor has to know n for termination detection, which is ini-

tially unknown. The actual algorithm, operations to compute n should be executed simultaneously with the following.

An $(n-1)$ round algorithm:

1. Definitions of the basic primitive internal operations for P_i:

   x:= the message that contains the element.

   send(x,P) - sends the message x to processor P whose mark is either *left* or *right*.

   receive(x,P) - receives the message x from processor P whose mark is either *left* or *right*.

   exchange(a,b,c) - exchanges the elements a,b and c in the order $small < medium < large$.

   swap(a,b) - swaps the elements a and b.

   $STOP$ - completes the execution.

2. Local variables at P_i:

   u_i - a variable or an element to be sorted.

   v_i - variable or an element for sorting, initially an initial element.

   s,m,l - temporary variables used for exchanging the elements during sorting, initially undefined.

   rd - time (round), initially 0.

T_i - mark of the processor P_i. This mark decides the median processor whose mark is 1; Initially $T_i \equiv (T_i + 2)$ (mod 3)

n - number of processors in the line network.

3. Operations for P_i in round 0:

Begin

$u_i = v_i$; $T_i = i(mod\ 3)$;

if $T_i = 0$ then *send(*u_i*,right)*;

else if $T_i = 2$ then *send(*u_i*,left)*;

else ($T_i = 1$)

s=receive(u_{i-1}*,left)*; $m = u_i$; *l=receive(*u_{i+1}*,right)*;

endif

endif

$rd = rd + 1$;

$T_i = (T_i + 2)(mod\ 3)$;

End

4. Operations for P_i after the initial round:

Begin

if $rd < n - 1$ then

if $i = 0$ then s=0;

if $T_i = 0$ then $send(u_i, right)$;

else if $T_i = 1$ then m=u_i;

$l = receive(u_{i+1}, right);$

if $m > l$ then

$swap(m, l);$ $u_i = m;$

$send(l, right);$

endif

endif

endif

$T_i = (T_i + 2)(mod\ 3);$

else if $i = n - 1$ then l=0;

if $T_i = 2$ then $send(u_i, left)$

else if $T_i = 1$ then

$s = receive(u_{i-1}, left);$ $m = u_i;$

if $s > m$ then

$swap(s, m);$ $u_i = m;$

$send(s, left);$

endif

endif

endif

$T_i = (T_i + 2)(mod\ 3);$

else $(0 < i < n - 1)$

if $T_i = 0$ then $send(u_i, right)$

else if $T_i = 2$ then $send(u_i, left)$

else $s = receive(u_{i-1}, left)$;

$m = u_i$; $l = receive(u_{i+1}, right)$;

$exchange(s, m, l)$;

$send(s, left)$; $u_i = m$; $send(l, right)$;

endif

endif

$T_i = (T_i + 2)(mod\ 3)$;

endif

endif

$rd = rd + 1$;

else //prints the solution and no special rule is needed

$v_i = u_i$;

endif

STOP

End

It shall be noted that the proposed algorithm for the distributed sorting problem can be generalized to an embedded linear array that is derived from a general network. The general network shall be embedded into a linear array in such a way that the indices of the processors in the general network shall be embedded into the indices of the linear array. Now the links joining the successive processors in the general network shall be mapped to a single link joining the corresponding successive processor in the linear array. The links in the path between two successive

processors in the network are grouped into a single link having bidirectional communication capability with path record. This is the idea behind the embedding of a general network into a linear array. Note that the path stored from processor P_i to P_{i+1} will be useful for message traversals.

After embedding the general network into a linear array, we compute all path records between two successive processors. One path record consists of the details of all intermediate processors and it is assumed that within a time round, element of one processor will be exchanged with the element of the successive processor via the computed path record. Then we can implement the proposed algorithm on the embedded linear array. The proposed distributed sorting algorithm takes $(n-1)$ rounds to complete distributed sorting.

6.4 Simulation Results

The simulation results show that the computational time of the proposed algorithm has been reduced to nearly one third of the time required for Sasaki's algorithm(Sasaki 2002). We have also noticed that the amount of communication to be spent in each round for exchanging the elements in the proposed algorithm is 4*Num(m_i) + 2*($n-$Num(m_i)), where Num(m_i) is the number of intermediate median processor(s) ($\approx \lceil n/3 \rceil$) and ($n-$Num($m_i$)) is the number of remaining processors including the end processors [which may either be a median processor or a non-median processor]. The amount of communication needed in Sasaki's algorithm amounts to 4*Num(P_i) + 2*Num($n-P_i$), where Num(P_i) is the number of intermediate processors and Num($n-P_i$) is the number of end processors.

Number of nodes (n)	Execution time of Sasaki's algorithm(Sasaki 2002) (in sec.)	Execution time of the proposed algorithm (in sec.)
1000	0	0
2000	1	0
5000	6	2
10000	23	10
15000	49	23
20000	136	41
25000	225	71
30000	355	99
32000	393	111

Table 6.1: Simulation results for distributed sorting on line network

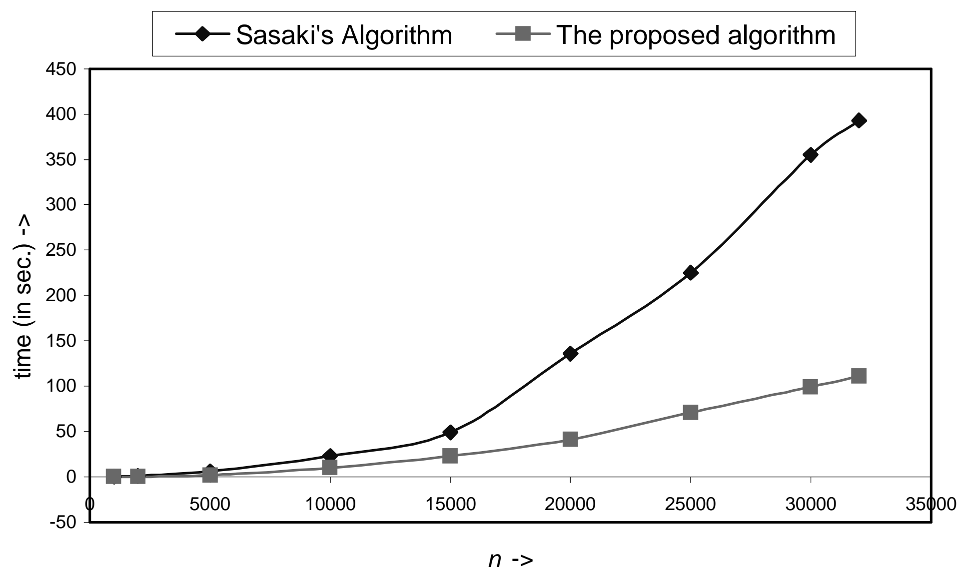

Figure 6.2: Performance comparison of the proposed algorithm with Sasaki's Algorithm

While comparing the amount of communication to be needed in each round for the Sasaki's algorithm which is based on the conventional odd-even transposition sort, the proposed algorithm is an improved result with less number of elements. We have tabulated the simulation results of the proposed algorithm with the performance comparison of Sasaki's algorithm (Sasaki 2002). From simulation results (Table 6.1 and Figure 6.2), it is clear that the proposed algorithm is robust for distributed environments. The proposed algorithm can be executed with both synchronous and asynchronous models by simple coping up with wakeup.

6.5 Sorting on static ad hoc mobile networks

In this section, we propose an algorithm for sorting n elements spread over a set of n processors on a static ad hoc mobile network in which processors/links do not fail. The proposed token based distributed sorting algorithm is event-driven. An event at processor P_i consists of receiving the token with a message from another processor $j \neq i$ and sending the token with the new sorted sequence to the other processor for entering into the critical section. The proposed sorting algorithm is described for a static ad hoc mobile network with $n = 8$ processors as shown in Figure. 6.3.

Data Structures

The following data structures are assumed for the execution of the proposed distributed sorting algorithm.

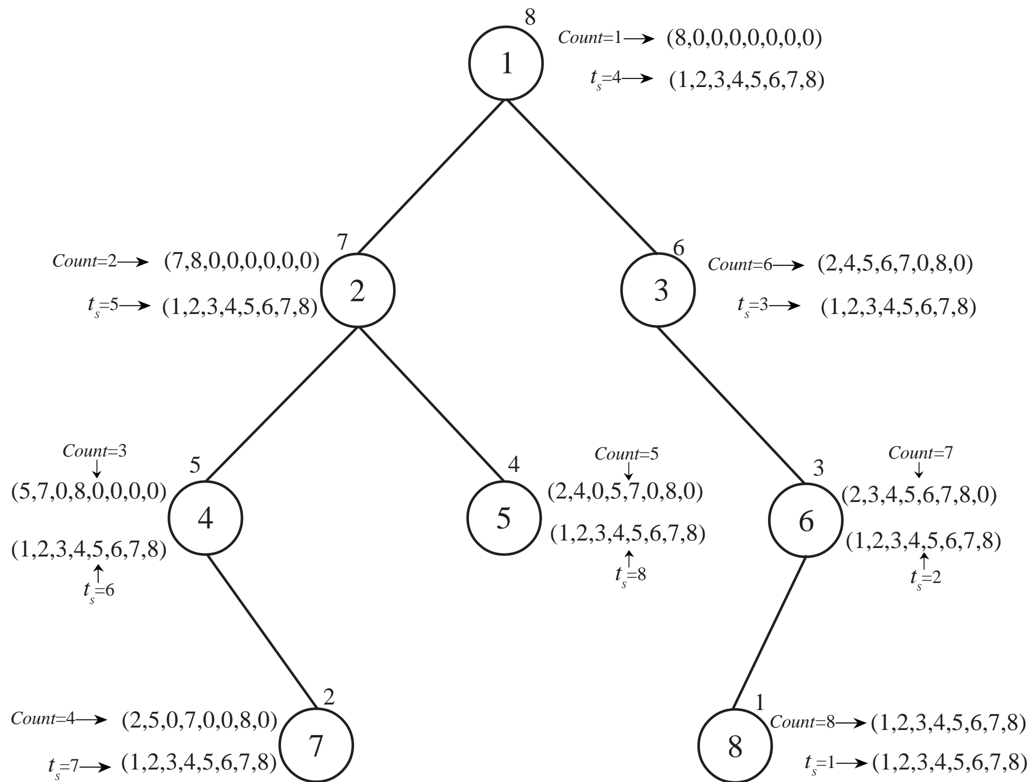

Figure 6.3: Proposed sorting algorithm for a static ad hoc mobile network.

- ◦ $S-$ the set, containing adjacent neighbors of a processor P_i, is arranged in the form of a circular linked list
- ◦ $n-$ the total number of processors, known to all processors in the network
- ◦ *count*– token counter which moves along the token and increased by 1 when $t_s = 0$; Initially *count* assumes 0
- ◦ t_s- token stopper, which moves along with the token, is used to stop the token when all the elements are sorted. This variable is incremented by 1

when *count* reaches n and when t_s reaches n, token stops at P_i; Initially t_s assumes 0

- T_c- token's carry which is assumed as an $n-$tuple that takes the elements of privileged processors at their respective positions in the $n-$tuple (0_1, 0_2, $\cdots$, 0_{i-1}, 0_i, 0_{i+1}, $\cdots$, 0_{n-1}, 0_n). The ith position holds the element of the processor P_i. Initially T_c assumes 0 at all positions.

- $A_i[n]-$ local array of the processor P_i. At first, $A_i[n]$ is initialized to zero at all positions of $n-$tuple except at the ith position. The ith position represents its own element.

- $Holder_i-$ a local variable of P_i that indicates the direction in which the token has been forwarded. Initially $Holder_i = 0$ at all processors except at P_1 where $Holder_i = self$; After the movement of the token at P_i, each processor maintains $Holder_i = j$ that represents a directed edge from processor i to j. Also if $Holder_i = self$ then the current processor i is itself the privileged processor

- Ind_i- a boolean flag bit that initially assumes zero at all processors. This flag bit assumes 1, if token has passed through the processor P_i and 0, otherwise

- $m,\ k-$ local variables to check whether all adjacent neighbors are served or not. Initially, m and k assume 0 at all processors.

Initially token is placed at P_1 and it is assumed that each processor P_i is equipped with sufficient memory to perform the local sorting of elements.

Description of the algorithm

As the proposed token based algorithm is fully distributed, each processor has to compare and exchange its element with the elements of other processors by token based message passing, so as to rearrange the elements in the sorted order. Also each processor has to be privileged at least once for comparing its elements with the elements of other processors. So, in the description of the algorithm, we have omitted the case of when P_i is not privileged. Hence the behavior of the proposed algorithm, when a processor P_i is privileged, is described as follows.

Begin

if ($count \neq n$) then set $k = 0$;

if $i = 1$ then $Holder_i = next \;\; l \in S$

if $m = 0$ then

copy $A_1[n]$ in T_c; set $count = count + 1$; set $m = m + 1$;

$Ind_i = 1$ and send token to the next processor $l \in S$

else ($m \neq 0$)

send token to the next processor $l \in S$

endif

else ($1 < i \leq n$) $Holder_i = next \;\; l \in S$

if $m = 0$ then

insert its element in T_c using binary search; set $Ind_i = 1$;

copy the sorted sequence T_c in $A_i[n]$;

set $count = count + 1$; $m = m + 1$;

if $count \neq n$ then

send token to the next processor $l \in S$

else $(count = n)$

set $t_s = t_s + 1$; $Ind_i = 0$; reset $m = 0$

and send token to the next processor $l \in S$

endif

else $(m \neq 0)$

send token to the next processor $l \in S$

endif

endif

else$(count = n)$ reset $m = 0$;

if $t_s \neq n$ then $Holder_i = next \;\; l \in S$;

if $k \neq deg(P_i) - 1$ then set $k = k + 1$;

if $Ind_i = 1$ then

copy T_c in $A_i[n]$; set $t_s = t_s + 1$; reset $Ind_i = 0$

and send token to the next processor $l \in S$

else $(Ind_i = 0)$

send token to the next processor $l \in S$

endif

else $(k = deg(P_i) - 1)$

copy T_c in $A_i[n]$;

if $Ind_i = 1$ then

set $t_s = t_s + 1$; reset $Ind_i = 0$;

and send token to the next processor $l \in S$

else ($Ind_i = 0$) send token to the next processor $l \in S$

endif

endif

else ($t_s = n$)

copy T_c in $A_i[n]$; reset $Ind_i = 0$; $Holder_i = self$

and stop token at P_i itself

endif

endif

End

Here the computational step is counted in terms of the number of movements of the token in the network. This token based sorting algorithm requires $2(n-1)$ movements for arriving at a sorted sequence and $2(n-1)$ movements for copying the sorted sequence at all processors. Each processor is assumed to have $O(n)$ space for their local computations and each processor is privileged at least twice. In figure 6.3, the token, from processor P_8 carries the sorted sequence and rests at P_2 after copying the sorted sequence at all processors. At P_2, *count* $= n$ and $t_s = n$. The worst case occurs when there exists $(n-1)$ leaf nodes with exactly one common processor, that is., the topology reduces to the star-shaped network. In this token oriented distributed sorting algorithm, we do not require a separate selection rule for selecting the solution as in (Sasaki 2002). Thus we have,

Theorem: 6.1

The above token based distributed sorting algorithm for token oriented ad hoc mobile network with n processors and $(n-1)$ wireless links requires $4(n-1)$ token movements in the worst case.

Proof:

Token is placed initially at P_i $(i = 1)$ and token's carry T_c is initialized to zero at all positions. Now the processor P_i, which is privileged, copies the element to be sorted in the ith position of the $n-$tuple (T_c) and *count* is incremented by 1. Then the modified and updated token's carry T_c is sent to processor P_j. Now, as and when P_j receives the updated T_c, it updates T_c by copying its element to be sorted in the jth position of T_c and *count* is incremented by one. The circular linked list (S) header is advanced to the next element of P_i from which the token has arrived at. Once this process is over, P_j sends the modified and updated T_c to another processor say P_k. This process is repeated until all positions in the T_c are modified (or all processors are visited at least once or *count* reaches n). As the token moves at most twice through an edge, it amounts to $2(n-1)$ token movements, for copying the elements in T_c.

When *count* reaches n, it implies that T_c has collected all elements to be sorted from all processors in the network. Now, T_c is sorted in an increasing order regardless of its original position in the $n-$tuple. After sorting, the element in the ith position in the $n-$tuple represents the sorted element to be posted to the processor P_i. Thus to post all sorted elements, token has to visit again all n processors atleast once. The posting of the sorted elements is controlled by a token stopper

T_s, which is incremented by one as and when a sorted element is copied at a processor. This process of copying the sorted element stops when T_s reaches n. This type of token passing with the sorted sequence needs $2(n-1)$ token movements to copy the sorted element at the respective processor.

Thus, the proposed distributed sorting algorithm requires $2(n-1)+2(n-1) = 4(n-1)$ token movements in the worst case. ∎

6.6 Computing prefix sums on static ad hoc mobile networks

In this section, we present an algorithm for the prefix computation problem on a static ad hoc mobile network. We implement the token based event-driven algorithm in the network in which processors / links do not fail. As and when the token is passed from one processor to another processor, $Holder_i$ is maintained at each processor P_i.

Data Structures

The following data structures are assumed for the execution of the proposed distributed algorithm for computing prefix sums.

- $S-$ the set, containing the indices of adjacent neighbors of processor P_i, is arranged in the form of a circular linked list
- $n-$ the total number of processors, which is known to all processors in the network
- u_i- the element at processor P_i; Initially an initial element
- v_i- the computed prefix sum at processor P_i, i.e., $v_i = u_1 \oplus u_2 \oplus \cdots \oplus u_i$

- $count-$ the token counter which moves along the token and increased by 1 when $t_s = 0$; Initially $count$ assumes 0

- t_s- token stopper, which moves along the token, is used to stop the token when all processors have performed prefix computation. This variable increases by 1 when $count$ reaches n and when t_s reaches n, token stops at P_i; Initially t_s assumes 0

- T_c- token's carry which is an $n-$tuple that takes the elements of the privileged processors at their respective positions in the $n-$tuple; Initially T_c assumes 0 at all positions; The ith position represents the element of processor P_i.

- $Holder_i-$ a local variable of P_i to indicate the processor through which the token has been sent. Initially $Holder_i = 0$ at all processors except at P_1 where $Holder_i = self$; After the movement of the token, each processor maintains the variable $Holder_i = j$ that represents a directed edge from processor P_i to P_j. Also if $Holder_i = self$ then the current node P_i is itself the privileged node

- Ind_i- a boolean flag bit. This flag bit assumes 1, if token has passed through the processor P_i and 0, otherwise; Initially Ind_i assumes 0 at all processors

- $m,\ k-$ the local variables to check whether all adjacent neighbors are served or token has performed prefix computation at all adjacent processors. Initially, m and k assume the value - 0 at all processors.

Initially token is placed at P_1 and it is assumed that each processor has sufficient memory to perform local operations with at most n elements.

Description of the algorithm

As the proposed token based algorithm is fully distributed, each processor P_i has to compute prefix sums with the elements of i processors by token based message passing strategy. Also each processor has to be privileged at least once for passing token to compute prefix sums with the elements of other processors. So, in the description of the algorithm, we have omitted the case of when P_i is not privileged. Hence the behavior of the proposed algorithm, when a processor P_i is privileged, is described as follows.

Begin

if $count \neq n$ then reset $k = 0$; $Holder_i = next \;\; l \in S$

if $m = 0$ then

if $i = 1$ then copy u_i $(i = 1)$ in the ith position of T_c;

set $count = count + 1$; $m = m + 1$; $Ind_i = 1$ and

send token to the next processor $l \in S$

else $(1 < i \leq n)$

copy u_i in the ith position of T_c;

set $count = count + 1$; $m = m + 1$; $Ind_i = 1$ and

if $count \neq n$ then

send token to the next processor $l \in S$

else $(count = n)$

compute $v_i = u_1 \oplus u_2 \oplus \cdots \oplus u_i$ using $u_i \in T_c$;

set $t_s = t_s + 1$; reset $Ind_i = 0$; $m = 0$ and

send token to the next processor $l \in S$

endif

endif

else$(m \neq 0)$

send token to the next processor $l \in S$

endif

else $(count = n)$ reset $m = 0$;

if $t_s \neq n$ then $Holder_i = next \;\; l \in S$

if $k \neq deg(P_i) - 1$ then set $k = k + 1$;

if $Ind_i = 1$ then

compute $v_i = u_1 \oplus u_2 \oplus \cdots \oplus u_i$ using $u_i \in T_c$; reset $Ind_i = 0$;

set $t_s = t_s + 1$ and send token to the next processor $l \in S$

else $(Ind_i = 0)$

send token to the next processor $l \in S$

endif

else $(k = deg(P_i) - 1)$

compute $v_i = u_1 \oplus u_2 \oplus \cdots \oplus u_i$ using $u_i \in T_c$;

if $Ind_i = 1$ then

set $t_s = t_s + 1$; reset $Ind_i = 0$ and

send token to the next processor $l \in S$

else($Ind_i = 0$)

send token to the next processor $l \in S$

endif

endif

else ($t_s = n$)

reset $Ind_i = 0$; $Holder_i = self$

and stop the token at P_i itself

endif

endif

End

The complexity of this algorithm, in terms of the number of movements of the token, resembles with the complexity of the distributed sorting algorithm on static ad hoc mobile networks. As in the earlier distributed sorting algorithm, the worst case occurs when the topology reduces to the star-shaped network. This token based prefix computation algorithm can be generalized to dynamic ad hoc mobile networks where the topology is frequently changing due to the movements of mobile hosts from one transmission radius to another transmission radius. During the generalization, we assume a single variable that could be broadcasted to all processors so that the change in n would be intimated to all processors. Here also we do not require a separate rule for selecting the solution as in (Sasaki 2002).

Theorem: 6.2

The above token based distributed prefix computation algorithm for token oriented ad hoc mobile network with n processors and $(n-1)$ wireless links requires $4(n-1)$ token movements in the worst case. ■

6.7 Conclusion

In this chapter, we have proposed a distributed sorting algorithm for sorting n elements distributed over a set of processors in a line network. The algorithm reduces the number of elements needed for sorting to exactly n elements without creating the copies of elements at intermediate processors. The simulation results show that the proposed algorithm is faster than the Sasaki's time-optimal distributed sorting algorithm. Even though the proposed algorithm takes $(n-1)$ rounds in the worst case, it is still robust. Next we have described a technique for distributed sorting on the linear embedding from a general network. Then we have proposed a distributed sorting algorithm on the static ad hoc network by keeping track of the token at intermediate processors. Finally we have described an algorithm, for prefix computation in the static ad hoc mobile networks that can suitably be modified and extended for dynamic ad hoc mobile networks.

Chapter 7

Conclusions and future work

In this dissertation, we have investigated the single shared resource allocation problem using request based token passing strategy in various interconnection networks. Then we have presented a median based algorithm for distributed sorting on a line network. Finally, we have proposed algorithms for sorting and prefix computation on a static ad hoc mobile network. A brief description of our investigation is given below.

In chapter 1, we have briefly outlined the recent developments in token based control strategies in distributed systems and the present work. In all single shared resource allocation algorithms, request message or token carries no identities of source and destination processors. Also we assume that there is no global clock, no common memory and each processor can again generate a request only if its previous request has been satisfied.

In chapter 2, we have presented request based token passing strategy for single shared resource allocation in a bidirectional ring network and a linear array. We have observed that the shared resource allocation algorithm D' proposed for a bidi-

rectional ring network does not require the additional check tour as in Feuerstein *et* al (Feuerstein *et* al. 1998). The next proposed algorithm $L1$ allocates the single shared resource in a linear array.

In chapter 3, we have discussed algorithms for single shared resource allocation in ring extensions. The algorithm UTRN proposed for single shared resource allocation in unidirectional touching rings requires atmost $(n-1)$ messages per request and $\frac{(5n-2)}{2}$ service traffic. The shared resource allocation algorithm proposed for bidirectional touching rings requires $2(n-1)+1$ messages per request and $(3n-2)$ service traffic in the worst case. The next algorithm InteRN-B allocates the single shared resource in bidirectional intersecting rings with n messages per request and $2n$ service traffic.

In chapter 4, we have presented single shared resource allocation algorithms in interconnected networks. The algorithm ICRN for shared resource allocation in interconnected rings assumes that the token maintains a counter to record the receipt of request messages. Then we have described the allocation of single shared resource in a single side wrap around mesh and in this algorithm MICRN, token makes a check tour in the base ring to test its awareness of all pending requests. Next we have presented an algorithm SRA_Mesh for single shared resource allocation in a regular 2-dimensional mesh.

In chapter 5, we have proposed single shared resource allocation algorithms for general networks. At first, the given graph is converted into a token oriented acyclic network in which there is a directed path from each processor to the processor with the token. The proposed algorithm GNet consists of two phases. The first

phase allocates the shared resource among leaf processors and the second phase controls the movements of the token and request messages at central processors. This algorithm GNet assumes no knowledge about the adjacent processor(s) and each processor traces incoming requests only by the indices locally assigned to links incident on it. The processors in the token oriented acyclic network shall be embedded into the processors in a linear array and hence the algorithm $L1$ stated in chapter 2 can be applied for single shared resource allocation in the embedded linear array.

In chapter 6, we have proposed an efficient algorithm with *median based exchanges* for distributed sorting on a line network. The proposed sorting algorithm improves the performance of each processor without creating copies of $(n-2)$ elements at intermediate processors. However, this algorithm assumes the knowledge of global position of each processor. The simulation results show that the proposed algorithm is robust. The proposed algorithm could be extended to the linear embedding of a general network. Then we have presented algorithms for distributed sorting and prefix computation on static ad hoc mobile networks. The elements are distributed over a set of mobile processors and then the network is converted into token oriented acyclic network in which the token visits each processor atleast twice. Here the number of token movements required for sorting (and prefix computation as well) is $4(n-1)$. These algorithms could also be extended to dynamic ad hoc mobile networks by keeping track of the mobility of processors from one transmission radius to another.

Future work

The token based control strategy used to design algorithms for shared resource allocation, sorting and prefix computation may be used to investigate some interesting issues in popular interconnection networks. The list of such issues include:

- Investigation of single shared resource allocation in augmented rings like chordal rings, distributed loops, multi-rings, etc (Arder and Lee 1981, Aiello *et* al. 2001).
- Investigation of processor or link failures during the allocation of single shared resource in various popular interconnection networks such as mesh, hypercube and their extensions
- Investigation with the performance analysis of the shared resource allocation algorithms in interconnection networks with virtual channels for other networks such as: *k-ary n-cubes, torus, etc.*

A simulation model to carry out the performance analysis of the proposed single shared resource allocation algorithm in a bidirectional ring network may be developed and further investigations can be made in the presence of one or more faulty processors in the bidirectional ring. This investigation may include the performance analysis of the algorithm proposed for single shared resource allocation in a linear array.

Similarly the design of the simulation model for measuring the performance of the shared resource allocation algorithms proposed for various ring extension

topologies would be exciting. The study could also be extended to other augmented rings like chordal rings, distributed loops which are well suitable for local area networks. The algorithm MICRN proposed for single side wrap around mesh can be generalized for allocating the single shared resource in torus which is a mesh with fully wrap around.

The algorithm proposed for general networks can be easily generalized to the $k-$port model which is a multiport message passing fully connected multicomputer (Lin 1996, Bruck *et* al. 1997). The design of the simulation model for the shared resource allocation algorithm in $k-$port model would be very interesting. The shared resource allocation algorithms presented for token oriented acyclic network may also be useful and effective for ad hoc networks provided the construction of the token oriented acyclic network is based on the self stabilizing approach of Huang and Chen (Huang 1993, Huang and Chen 1993).

The proposed median based sorting algorithm for a line network can be modified and implemented for sorting n elements in mesh and its related topologies. The prefix computation algorithm proposed for a static adhoc mobile network could be extended to dynamic adhoc mobile network in which the movements of the mobile hosts are carefully observed.

Bibliography

Agrawala. D., and El Abbadi. A., An efficient and fault tolerant solution for distributed mutual exclusion, *ACM Trans. on Computer Systems*, **9** (1991) 1-20.

Aiello. W., Bhatt. S.N., Cheng. F.R.K., Rosenberg. A.l., and Sitaraman. R.K., Augmented ring networks, *IEEE Trans. on Parallel and Distributed Systems*, **12** (2001) 598-609.

Akl. S.G., *The Design and Analysis of Parallel Algorithms*, Prentice-Hall, Englewood Cliffs, New Jersy, USA, 1989.

Akl. S.G., *Parallel Computation: Models and Methods*, Prentice-Hall, Upper Saddle River, New Jersy, USA, 1997.

Arden. B.W., and Lee. H., Analysis of chordal ring network, *IEEE Trans. on Computers*, **30** (1981) 291-295.

Andrews. D., and Schulz. G., A token-ring architecture for local area networks: An update, in : *Proc.IEEE COMPCON*, 1982.

Barrows. M.E., Gregory. D.E., Gao. L., Rosenberg. A.L., and Cohen. P.R., An emperical study of dynamic scheduling on rings of processors, *Parallel Computing*, **25** (1999) 1063-1079.

Ben-Dor. A., Haveli. S., and Schuster. A., Potential function analysis of greedy hot potato routing, *Theory of Computing Systems*, **31** (1998) 41-61.

Bertsekas. D., and Gallager. R., *Data Networks*, Prentice Hall, New Jersy, 1992.

Bruck. J., Ho. C.T., Kipnis. S., Upfal. E., and Weathersby. D., Efficient algorithm for all-to-all communications in multiport message passing systems, *IEEE Trans. on Parallel and Distributed Systems*, **8** (1997) 1143-1155.

Burkhardt. H. *et* al., Overview of the KSR1 Computer System, *Tech. Report*, KSR-TR9202001, Kendell Square Research, 1992.

Burns. J.E., and Pachl. J., Uniform self-stabilizing rings, *ACM Trans. on Programming Languages and Systems*, **11** (1989) 330-334.

Bux. W., Performance issues in local area networks, *IBM Journal*, **23** (1984).

Chandy. K., and Misra. J., The drinking philosophers problem, *ACM Trans. on Programming Languages and Systems*, **6** (1984) 632-646.

Carhalho. O., and Roucairol. G., On mutual exclusion in computer networks, *Communications of the ACM*, **26** (1983) 146-147.

Chalamaiah. N., and Ramamurthy. B., Finding shortest paths in distributed loop networks, *Information Processing Letters*, **67** (1998) 157-161.

Chalamaiah. N., *A study of issues in multiconnected distributed loop interconnected networks*, Ph.D. Dissertation, Indian Institute of Technology, Kharagpur, India, July 1999.

Chang. Y.-I., A hybrid distributed mutual exclusion algorithm, *Microprocessing and Microprogramming*, **41** (1996) 715-731.

Chang. Y.-I., Singhal. M., and Liu. M.T., A fault tolerant algorithm for distributed mutual exclusion, *in: Proc. of 9th Symposium on Reliable Distributed Systems,* IEEE Computer Society, (1990) 146-154.

Chaudhuri. P., Karatta. M.K., An $O(n^{\frac{1}{3}})$ algorithm for distributed mutual exclusion, *J. Systems Architecture*, **45** (1998) 409-420.

Chen. N.-S., Yu. H.P., and Huang. S.-T., A self-stabilizing algorithm for constructing spanning trees, *Information Processing Letters*, **39** (1991) 147-151.

Clark. D.D., Pogran. K.T., and Reed. D.P., An introduction to local area networks, *in: Proc. of IEEE*, **66** (1978) 1497-1517.

Cosares. S., Saniee. I., and Wasen. O., Network planning with the SONET tool kit, Bell Core Exchange, (1992) 8-15.

Datta. A.K., Johnen. V., Petit. F., and Villain. V., Self-stabilizing depth-first token circulation in arbitrary rooted networks, *Distributed Computing*, **13** (2000) 207-218.

Dhamdhere. D.M., and Kulkarni. S.S., A token based $k-$ resilent mutual exclusion algorithm for distributed systems, *Information Processing Letters*, **50** (1994) pp. 151-157.

Dijkstra. E.W., Solution of a problem in concurrent programming control, *Communications of the ACM*, **8** (1965) p. 569.

Dijkstra. E.W., Hierarchical ordering of sequential processes, *Acta Informatica*, **1** (1971) 115-138.

Dijkstra. E.W., Self stabilizing systems inspite of distribuited control, *Communications of the ACM*, **17** (1974) 643-644.

Dijkstra. E.W., Self stabilization in spite of distributed control, *in: Selected Writings on Computing: A Personal Perspective*, Springer-Verlag, New York, USA (1982) 41-46.

Duato. J., Yalamanchili. S., and Ni. L., *Interconnection Networks: An Engineering Approach*, Morgan Kaufmann, San Franscisco, USA, 2003.

Farkar. W.D., and NewHall. E.E., An experimental distributed switching system to handle bursty computer traffic, *in: Proc. of ACM Symposium on Prob. Optim. Data Comm. Syst.*, (1969) 1-33.

Farker. E.D., and Larson. K.C., The system architecture of the distributed computer system - The communications system, presented at: *The Symposium on Computer Networks*, Polytechnique Institute of Brooklyn, New York, USA, 1972.

Feuerstein. E., Leonardi. S., Marchetti - Spaccamela. A., and Santoro. N., Efficient token based control in rings, *Information Processing Letters*, **66** (1998) 175-180.

Frederickson. G.N., Tradeoffs for selection in distributed networks, *in: Proc. 2*nd *ACM Symposium on Principles of Distributed Computing*, (1983) 154-160, Association of Compututing Machinery, New York.

Fu. S., and Tzeng. N., Efficient token-based approach to mutual exclusion in distributed memory systems, Technical Report #TR-95-8-1, University of South Western Louisiana, (1995) 1-18.

Fu. S.S., Tzeng. N.F., and Chung. J.Y., Empirical evaluation of mutual exclusion algorithms for distributed systems, *J. of Parallel and Distributed Computing*, **60** (2000) pp. 785-806.

Gafni. E., and Bertsekas. D., Distributed algorithms for generating loop-free routes in networks with frequently changing topology, *IEEE Trans. on Communications*, **C-29** (1983) 11-18.

Gallager. R.G., Humblet. P.A., and Spira. P.M., A distributed algorithm for minimum-weight spanning trees, *ACM Trans. on Programming Languages and Systems*, **5** (1983) 66-77.

Gerstel. O., and Zaks. S., The bit complexity of distributed sorting, *Algorithmica*, **18** (1997) 405-416.

Ginat. D., Shankar. A.U., and Agrawala. A.K., An efficient solution to the drinking philosophers problem and its extensions, *in: Proc. of the 3*rd *International*

Workshop on Distributed Algorithms, Lecture Notes in Computer Science, Springer-Verlag, Heidelberg, Germany, **392** (1989) 83-93.

Greenwood. G.W., On the equity of mutual exclusion algorithms in distributed systems, *Information Processing Letters*, **56** (1995) 19-22.

Gregory. D.E., Gao. L., Rosenberg. A.L., and Cohen. P.R., An emperical study of dynamic scheduling on rings of processors, *in: Proc. of the 8th IEEE Symposium on Parallel and Distributed Processing*, (1996) 470-473.

Halsall. F., *Data Communications, Computer Networks and Open Systems*, Addison-Wesley, New York, USA, 1992.

Hajek. B., Bounds on evacuation time on deflection routing, *Distributed Computing*, **5** (1991) 1-6.

Helary. M., Plouzeau. N., and Raynal. M., A distributed algorithm for mutual exclusion in an arbitrary network, *The Computer Journal*, **31** (1988) 289-295.

Hofstee. H.P., Martin. A.J., and Van De Snepscheut. J.L.A., Distributed sorting, *Science and Computer Programming*, **15** (1990) 119-133.

Horowitz. E., Sahni. S., and Rajasekaran. S., *Computer Algorithms*, Galgotia Publications, New Delhi, India, 2001.

Housni. A., and Trehel. M., A new distributed mutual exclusion algorithm for two groups, *in: ACM Symposium on Applied Computing (SAC2001)*, Las Vegas, USA (2001) 531-538.

Huang. S.-T., Leader election in uniform networks, *ACM Trans. on Programming Languages and Systems*, **15** (1993) 563-573.

Huang. S.-T., and Chen. N.-S., A Self-stabilizing algorithm for constructing breadth-first trees, *Information Processing Letters*, **41** (1992) 109-117.

Huang. S.-T., and Chen. N.-S., Self-stabilizing depth first token circulation on networks, *Distributed Computing*, **7** (1993) 61-66.

IEEE Draft Standard 802.4: Token passing bus access methods and physical layer specifications: Draft D, December 1982.

Israeli. A., and Jalfon. M., Token management schemes and random walks yield self-stabilizing mutual exclusion, *in: Proc. of the 9th ACM Symposium on Principles of Distributed Computing*, (1990) 119-129.

Jana. P.K., Naidu. B., Kumar. S., Arora. M., and Sinha. B.P., Prallel prefix computation on extended multi-mesh network, *Information Processing Letters*, **84** (2002) 295-303.

Kakugawa. H., Fujita. S., Yamashita. M., and Ae. T., Availability of $k-$ Coterie, *IEEE Trans. on Computers*, **42** (1993) 553-558.

Kakugawa. H., and Yamashita. M., Uniform randomized seld-stabilizing mutual exclusion on unidirectional ring under unfair c-daemon, *in: Proc. of the 2nd Workshop on Self-Stabilizing Systems*, (1995) 14.1-14.13.

Kumar. V., Place. J., and Yang. G.-C., An efficient algorithm for mutual exclusion using queue migration in computer networks, *IEEE Trans. on Knowledge and Data Engineering*, **3** (1991) 380-384.

Lakshmanan. K.B., Meenakshi. N., and Thulasiraman. K., A time-optimal message-efficient distributed algorithm for depth-first-search, *Information Processing Letters*, **25** (1987) 103-109.

Lamport. L., Time, clocks, and ordering of events in distributed systems, *Communications of the ACM*, **21** (1978) 558-565.

Lamport. L., A fast mutual exclusion algorithm, *ACM Trans. on Computer Systems*, **5** (1987) 1-11.

Lamport. L., Perl. S., and Weihl. W., When does a correct mutual exclusion algorithm guarantee mutual exclusion?, *Information Processing Letters*, **76** (2000) 131-134.

Leighton. F.T., *Introduction to Parallel Algorithms and Architectures: Arrays, Trees, Hypercubes*, Morgan Kaufmann, San Mateo, USA, 1992.

Le Lann. G., Distributed systems - Towards a formal approach, in: B.Gilchrist(Ed.) *Information Processing*, **77** (1977) 155-160.

Le Lann. G., *Motivation, Objective and Characteristics of Distributed Systems*, Springer-Verlag, Heidelberg, Germany, **1** (1978) 1-9.

Leu. F-C., Tsai. Y-T., and Tang. C.Y., An efficient external sorting algorithm, *Information Processing Letters*, **75** (2000) 159-163.

Lin. Y.-C., Optimal prefix circuits with fan-out 2, in: *Proc. of the International Conference on Algorithms*, (1996) 175-181.

Lin. Y.-C., and Yeh. C.-S., Efficient parallel prefix algorithms on multiport message passing systems, *Information Processing Letters*, **71** (1999) 91-95.

Liu. M.T., Distributed loop computer networks, in: *Advances in Computers*, Academic Press, New York, **17** (1978) 155-160.

Liu. P., Aiello. W., and Bhatt. S., Tree search on an atomic model for message passing, *SIAM J. Computing*, **31** (2001) 67-85.

Lodha. S., and Kshemkalyani. A., A fair distributed mutual exclusion algorithm, *IEEE Trans. on Parallel and Distributed Systems*, **11** (2000) 537-549.

Loui. M.C., The complexity of sorting on distributed systems, *Information and Control*, **60** (1984) 70-85.

Luk. W.S., and Ling. F., An analytical/emperical study of distributed sorting on a local area network, *IEEE Trans. on Software Engineering*, **15** (1989) 575-586.

Lynch. N., Upper bounds for static resource allocation in a distributed system, *J. Computer and System Sciences*, **23**(1981) 254-278.

Lynch. N.A., *Distributed Algorithms*, Morgan Kaufmann, San Francisco, USA, 1996.

Maekawa. M., A $\sqrt{N}$ algorithm for mututal exclusion in decentralized systems, *ACM Trans. on Computer Systems*, **3**(1985) 145-149.

Makki. K., Banta. P., Been. K., and Ogawa. R., Two algorithms for mutual exclusion in s distributed system, in: *Proc. of the International Conference on Parallel Processing*, (1991) 460-466.

Makki. K., An efficient token based distributed mutual exclusion algorithm, *J. Computer and Software Engineering*, **2**(1994) 401-416.

Makki. K., Been. K., and Pissinou. N., A simulation study of token based mutual exclusion algorithms in distributed systems, *Int. J. in Computer Simulation*, **4** (1994) 65-88.

Makki. K., Pissinou. N., and Park. E., An efficient solution to the critical section problem, in: *Proc. of the International Conference on Parallel Processing*, St. Charles, IL, (1994) 77-80.

Makki. K., Delt. J., Pissinou. N., Melody Moh. W., and Xiaohua Jia., Using logical rings to solve the distributed mutual exclusion problem with fault tolerance issues, *J. Supercomputing*, **16** (2000) 117-132.

Marberg. J.M., and Gafni. E., Distributed sorting algorithms for multi-channel broadcast networks, *Theoretical Computer Science*, **52** (1987) 193-203.

McMillin. B.M., and Ni. L.M., Reliable distributed sorting through the application-oriented fault tolerance paradigm, *IEEE Trans. on Parallel and Distributed Systems*, **3** (1992) 411-420.

Misra. J., Detecting termination of distributed computations using makers, *in: Proc. of the 2nd ACM Symposium on principles of distributed Computing*, (1983) 290-294.

Mukhopadhyaya. K., and Sinha. S.B., Fault-tolerant routing in distributed loop networks, *IEEE Trans. on Computers*, **44** (1995) 1452-1456.

Naimi. M., and Trehel. M., An improvement of the log(n) distributed algorithm for mutual exclusion algorithm, in: *Proc. of the 7th International Conference on Distributed Computing Systems*, (1987) 371-375.

Nesterenko. M., and Mizuno. M., A quorum based self-stabilizing distributed mutual exclusion algorithm, *J. Parallel and Distributed Computing*, **62** (2002) 284-305.

Pan. Y., Hamdi. M., and Li. K., Efficient and scalable quicksort on a linear array with a reconfigurable pipelined bus system, *Future Generation Computer Systems*, **13** (1997/98) 501-513.

Petit. F., and Villain. V., Time and space optimality of distributed depth-first token circulation algorithms, in: *in: DIMACS Workshop on Distributed Data and Structures*, Carleton University Press, Ontario, Canada, (1999) 91-106.

Pissinou. N., Makki. K., Park. E., Hu. Z., and Wong. W., An efficient distributed mutual exclusion algorithm, in: *1996 International Conference on Parallel Processing*, (1996) 196-203.

Rajendra Prasath. R., and Thangavel. P., Token based message passing in bidirectional ring extensions, *J. Madras University (WMY-2000 Special Issue) - Section B: Sciences*, **52** (2000) 145-159.

Rajendra Prasath. R., and Thangavel. P., Shared resource allocation using token based control strategy in ring extension topologies, in: J.C.Misra and S.B.Sinha (Eds.), *Recent Trends in Mathematical Sciences*, Narosa Publishing House, New Delhi, India, (2000) 53-63.

R.Rajendra Prasath and P.Thangavel, Token based control strategy for shared resource allocation in chordal rings, in: *Proc. of the International Conference on Industrial Mathematics(ICIM 2001)*, Indian Institute of Technology, Madras on August 12-14, 2001.

R.Rajendra Prasath and P.Thangavel, Token based control algorithm with central coordinators, in: P.Thangavel (Eds.), *Algorithms and Artificial Systems*, Allied Publishers, Chennai, India, (2003) 25-40.

Rajendra Prasath. R., and Thangavel. P., Shared resource allocation using token passing strategy in interconnected networks, *Information*, **6** (2003) 197-206.

Raynal. M., *Algorithms for Mutual Exclusion*, MIT Press, Cambridge, USA, 1986.

Raynal. M., *Distributed Algorithms and Protocols*, John Wiley and Sons, New York, USA, 1988.

Raynal. M., Prime numbers as a tool to design distributed algorithms, *Information Processing Letters*, **33** (1989) 53-58.

Raymond. K., A tree based algorithm for distributed mutual exclusion, *ACM Trans. on Computer Systems*, **7** (1989) 61-77.

Raymond. K., A distributed algorithm for multiple entries to a critical section, *Information Processing Letters*, **30** (1989) 189-193.

Rego. V., and Ni. L.M., Analytic models of cyclic service systems and their applications to token-passing local networks, *IEEE Trans. on Computers*, **C-37** (1988) 1224-1234.

Reisig. W., *Elements of Distributed Algorithms*, Springer-Verlag, Heidelberg, Germany, 1998.

Rhee. I., A modular algorithm for resource allocation, *Distributed Computing*, **11** (1998) 157-168.

Ricart. G., and Agrawala. A., An optimal algorithm for mutual exclusion in computer networks, *Communications of the ACM*, **24**(1981) 9-17.

Ricart. G., and Agrawala. A., Author's response to "On Mutual Exclusion in Computer Networks" by Carvalho and Roucairol, *Communications of the ACM*, **26**(1983) 147-148.

Ross. F.E., FDDI-A tutorial, *IEEE Communication Magazine*, **24** (1986) 10-17.

Rotem. D., Santoro. N., and Sidney. J.B., Distributed sorting, *IEEE Trans. on Computers*, **C-34** (1985) 372-376.

Sandlers. B., The information structure of distributed mutual exclusion algorithms, *ACM Trans. on Computer Systems*, **5** (1987) 284-289.

Sasaki. A., A distributed resource allocation algorithm in a unidirectional ring, *IEICE Technical Report*, **101** (2001) 41-48.

Sasaki. A., A time-optimal distributed sorting algorithm on a line network, *Information Processing Letters*, **83** (2002) 21-26.

Saxena. P.C., and Gupta. S., A token-based delay optimal algorithm for mutual exclusion in distributed systems, *Computer Standards* & *Interfaces*, **21** (1999) 33-50.

Singhal. M., A taxonomy of distributed mutual exclusion, *J. Parallel and Distributed Computing*, **18** (1993) 94-101.

Stallings. W., *Data and Computer Networks*, Prentice Hall, New Delhi, India, 1997.

Stallings. W., *Operating Systems*, 4th Ed., Prentice Hall, New Delhi, India, 2002.

Suzuki. I., and Kasami. T., A distributed mutual exclusion algorithm, *ACM Trans. on Computer Systems*, **3** (1985) 344-349.

Tarjan. R.E., Amortized computational complexity, *SIAM J. Algorithms and Discrete Mathematics*, **6** (1985) 306-318.

Thambu. P., and Wong. J., An efficient token-based mutual exclusion algorithm in a distributed system, *J. Systems Software*, **28** (1995) 267-276.

Thangavel. P., and Rajendra Prasath. R., A note on token based control in rings and linear arrays, *J. of Computer Society of India*, **32** (2002) 62-65.

Velazquez. M.G., A survey of distributed mututal exclusion algorithms, *Tech. Report*, Colorado State University, Sep. 1993, 1-37.

Walter. J.E., Welch. J.L., and Vaidya. N.H, A mutual exclusion algorithm for ad hoc mobile networks, *Wireless networks*, **9** (2001) 585-600.

Wang. H., Nicolau. A., Siu. K.S., The strict time lower bound and optimal schedules for parallel prefix with resource constraints, *IEEE Trans. on Computers*, **45** (1996) 1257-1271.

Welch. J., and Lynch. N., A modular drinking philosophers algorithm, *Distributed Computing*, **6** (1993) 233-244.

Wu. J., *Distributed System Design*, CRC Press, Boca Raton, Florida, USA, 1999.

Wu. M.Y., and Shu. W., An efficient distributed token-based mutual exclusion algorithm with central coordinator, *J. Parallel and Distributed Computing*, **62** (2002) 1602-1613.

Yan. Y., Zhang. X., and Yang.H., A fast token-chasing mutual exclusion algorithm in arbitrary network topologies, *J. Parallel and Distributed Computing*, **35** (1996) 156-172.

Zaks. S., Optimal distributed algorithms for sorting and ranking, *IEEE Trans. on Computers*, **C-34** (1985) 376-379.

Zhang. S., Lee. E.S., and Burns. A., Determining the worst case synchronous message response time in FDDI networks, *The Computer Journal*, **44** (2001) 31-41.